MoDeST: A Compositional Modeling Formalism for Hard and Softly Timed Systems

Henrik Bohnenkamp[a] Pedro R. D'Argenio[b,a]
Holger Hermanns[c,a 1] Joost-Pieter Katoen[d,a]

[a]*Department of Computer Science*
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

[b] *CONICET – FaMAF. Universidad Nacional de Córdoba*
Ciudad Universitaria, 5000 – Córdoba, Argentina

[c]*Department of Computer Science*
Saarland University, D-66123 Saarbrücken, Germany

[d]*Department of Computer Science*
Technical University Aachen, D-52056 Aachen, Germany

Abstract

This paper presents MODEST (MOdeling and DEscription language for Stochastic Timed systems), a formalism that is aimed to support (i) the modular description of reactive system's behaviour while covering both (ii) functional and (iii) non-functional system aspects such as timing and quality-of-service constraints in a single specification. The language contains features such as simple and structured data types, structuring mechanisms like parallel composition and abstraction, means to control the granularity of assignments, exception handling, and non-deterministic and random branching and timing. MODEST can be viewed as an overarching notation for a wide spectrum of models, ranging from labeled transition systems, to timed automata (and probabilistic variants thereof) as well as prominent stochastic processes such as (generalized semi-)Markov chains and decision processes. The paper describes the design rationales and details of the syntax and semantics.

Key words: modeling formalism, compositionality, formal semantics, timed automata, stochastic processes

[1] Contact author. E-mail: `hermanns@cs.uni-sb.de`, tel.: `+49-681-3025631`, fax: `+49-681-3025636`

1 Introduction

Background and motivation. The prevailing paradigm in computer science to abstract from physical aspects is gradually being recognized to be too limited and too restricted. Instead, classical abstractions of software that leave out "non-functional" aspects such as cost, efficiency, and robustness need to be adapted to current needs. In particular this applies to the rapidly emerging field of *"embedded"* software [19,29,48].

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

Embedded software is an important motivation for the development of modeling techniques that on the one hand provide an easy migration path for design engineers and, on the other hand, support the description of quantitative system aspects. This has resulted in various extensions of light-weight formal notations such as SDL (System Description Language) and the UML (Unified Modeling Language), and in the development of a whole range of more rigorous formalisms based on e.g., stochastic process algebras [36,38], or appropriate extensions of automata such as timed automata [3], probabilistic automata [54] and hybrid automata [2]. Light-weight notations are typically closer to engineering techniques, but lack a formal semantics; rigorous formalisms do have such formal semantics, but their learning curve is typically too steep from a practitioner's perspective and they mostly have a restricted expressiveness.

In this paper, we propose a description language that is intended to have a rigid formal basis (i.e., semantics) and that incorporates several ingredients from light-weight notations such as exception handling [2], modularization, atomic

[2] Exception handling in formal specification languages has received scant attention.

assignments, iteration, and simple data types. The semantics enables formal analysis 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. First industrial case studies [8,10] with our tool environment [9] confirm that the rigid approach towards the semantics results in trustworthy analysis results, obtained via discrete-event simulation. Standard simulation environments are risky to use instead as they may yield contradictory results even in simple case studies [17].

Issues of concern. Important rationales behind the development of the description language, called MODEST (MOdeling and DEscription language for Stochastic Timed systems), have been:

- *Orthogonality.* 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 some other commonly used languages. The syntax resembles that of the programming language `C` and the modeling language Promela [39]. Data modularization concepts and exception handling mechanisms have been adopted from modern object-oriented programming languages such as Java. Process algebraic constructs have been strongly influenced by FSP (Finite State Processes [44]), a simple, elegant language that is aimed at educational purposes.
- *Practical considerations.* The design of the language and the development of accompanying prototypical tool support have taken place hand-in-hand. Considerations about the tool handling of language constructs have been one of the driving forces behind the language development.
- *Expressiveness.* Several concepts – all well studied and widely accepted in the fields of e.g., computer-aided verification and concurrency theory – have been considered:

(1) *Action nondeterminism* is often used in concurrent system design to leave parts of the description underspecified or to allow different reactions on stimuli from the embedding environment, and is an appropriate means to reflect that the order of events in concurrent executions is out of the control of a modeler.
(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 (e.g., coin flipping), and to represent (randomized) scheduling strategies.
(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.
(4) *Delay nondeterminism* allows one to leave the precise timing of events unspecified, and only indicate a lower- and upper-bound on their occurrence

Notable exceptions are e.g., Enhanced-LOTOS [32] and Esterel [7].

time.

(5) *Random variables* are used to quantify the likelihood of an 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. It is our belief that each of these concepts is indispensable when striving for an integrated consideration of quantitative system aspects during the entire system design trajectory.

Organization of the paper. Section 2 introduces the language ingredients of MODEST by presenting a compositional model of a soccer match. Section 3 defines stochastic timed automata, a model that allows for the symbolic (i.e., finite) representation of continuous-time stochastic phenomena. This model is used as the semantic basis for MODEST. Section 4 presents the syntax of MODEST and its operational semantics that associates with each MODEST process a stochastic timed automaton. Section 5 is rather technical and presents the formal interpretation of stochastic timed automata. Whereas these automata are mostly finite-state, their semantics is an infinite (probabilistic) transition system due to the continuous nature of time. Section 6 shows how some well-known constructs (like while-loops and location invariants of timed automata) can be modeled in MODEST. To conclude, Section 7 discusses the motivations for the design decisions that have been made while developing MODEST, and Section 8 concludes the paper.

This paper is an extended and revised version of [25].

2 A Gentle Language Primer

This section introduces the core language features of MODEST by modelling an abstract view on a soccer match between two teams. Although this example is not a typical software engineering example, it illustrates almost all ingredients of MODEST in a rather natural way.

A *Soccer* match is played by two teams of 11 players each. There is one ball to play with and a referee who occasionally blows the whistle and keeps track of the total playing time of 90 minutes. The team that has the lowest score at the end of the match or has no players left on the field *looses* the match. In the following, the potential evolution of a soccer match is described using MODEST. The description heavily uses its compositional features.

```
int score[2];
int players[2];
players[0] = 11;
players[1] = 11;
```

To start with, in order to keep track of the score and the number of players left on the field, two arrays of integers are introduced. The array *score* has dimension 2. $score[0]$ ($score[1]$) equals the number of goals made by team 0 (team 1). In MODEST, newly introduced integers are set to zero by default. $players[0]$ ($players[1]$) is the number of players of team 0 (1). Both fields are set explicitly to 11, the number of players initially. In the following, the teams are distinguished by the numbers 0 and 1.

```
process FoulPlay(){
    clock c;
    float delay;
    {= delay = UNIFORM(2,5), c = 0 =};
    when(c == delay)
        urgent(c == delay)
            throw foul
}
```

One of the main activities of players during a match is fouling other players. To describe this behaviour, the process *FoulPlay* is introduced. This process describes that a team tries to foul other players, but before the foul happens some time passes that is uniformly distributed over the interval $[2, 5]$. To measure this time, MODEST provides the concept of *clocks*, real-valued variables which increase *linearly* and *continuously* with time by a constant rate 1. First, the clock c is reset to zero, and a random sample from a uniform distribution function on the interval $[2, 5]$ is drawn and assigned to the float variable *delay*. Assignments that are enclosed in {= ... =} are executed atomically. To measure that *delay* units of time have passed, the conditional constructs when$(\cdot)$ and urgent$(\cdot)$ are used. The Boolean expression in a when$(\cdot)$ construct determines when the process following the construct is allowed to be executed. The Boolean expression in an urgent$(\cdot)$ construct describes when the process following the construct is *required* to be executed at the latest. Since Boolean expressions may refer to clocks, the evaluation of the expressions might change over time. In this example, the expressions in the when$(\cdot)$ and urgent$(\cdot)$ construct are the same: $c == delay$. This means that as soon as $c == delay$ holds, the following process has to be executed with no further delay. The process action to be executed is the construct throw *foul*, which *throws* the *exception* named *foul*. Exceptions signal certain exceptional conditions in the execution of the process. An exception may be caught outside the process it was thrown. Exception handling will be further discussed below.

The ball, once possessed by the team with number *team*, is kicked away, either towards another player or in the goal. This is described by the process *Pass*. *Pass* takes one parameter, the integer *team*, indicating which team is currently playing the ball. The ball is kicked away, as indicated by action *kick*, and is either passed to another player (of the own or the opponent team) or goes into a goal (of the own or the opponent team). Note that in absence of a when- and urgent-construct, the action *kick* is not restricted in any way, however, it is also not *required* to happen. The interpretation is that it happens sometime, but it is unspecified when this would be.

```
process Pass(int team){
    do {
        :: kick palt {
            :0.9 · players[team]:
                self
            :0.9 · players[1−team]:
                other; break
            :0.1 · players[team]:
                {= score[team] += 1 =};
                goal; other; break
            :0.1 · players[1−team]:
                {= score[1−team] += 1 =};
                goal; break
        }
    }
}
```

The four possibilities are described by means of the palt construct, which describes a *probabilistic* choice between alternatives. The branching probabilities are implicitly determined by so-called *weight expressions*, which are arithmetic expressions (embraced by colons) that evaluate to non-negative values. The probability of a branch to happen is given by the weight of this branch, divided by the sum of the weights of all branches of the palt construct. Since weight expressions are allowed to refer to variables, the weights, and therefore, the branching probabilities might change during the execution. Let $p_{all} = players[team] + players[1-team]$. In the example, with probability $0.9 \cdot players[team]/p_{all}$, the ball is passed to a player of the own team (indicated by action *self*), with probability $0.9 \cdot players[1-team]/p_{all}$ to a player of the opponent team (action *other*), and so on. Since the number of players on the field varies, the probabilities where the ball eventually ends up vary as well. In particular, the larger the difference between the number of players of the two teams get, the smaller the probability for the small team to keep the ball in possession and to score.

The described palt in process *Pass* is embedded in a do construct. The do construct has in general two purposes: expressing nondeterminism between different processes, and restarting itself once a chosen process has terminated. In case of process *Pass* the do indicates that the probabilistic choice should be repeated indefinitely. This infinite behaviour is aborted on executing the construct break, and resumed with the process following the do construct (if any). In our example, this occurs whenever either the ball is lost to the other team or a goal is scored.

```
process Play(int team){
    gotBall; Pass(team); Play(team)
}
```

A team can only pass if it possesses the ball. This is described by the process *Play*: whenever action *gotBall* is executed, process *Pass* is invoked. Subsequently, process *Play* is invoked recursively. In addition to the do construct, recursion is thus another way to specify infinite behaviour.

The behaviour of a complete team can be described now as the *parallel composition* of two processes, as done in process *Team*: the process *Play* describing the handling of the ball, and the process *FoulPlay* describing the handling of the opposite team. Using parallel composition to describe the behaviour of a team is justified, since usually only one player can handle the ball, whereas the others can still foul each other.

```
process Team(int team){
    par {
        :: Play(team)
        :: FoulPlay(team)
    }
}
```

The two different teams are instantiated as follows:

```
process Team0(){                          process Team1(){
    hide{kick, self, goal}                    hide{kick, self, goal}
        relabel  {other, gotBall, foul}           relabel  {other, gotBall, foul}
        by       {0to1, 1to0, foul0}              by       {1to0, 0to1, foul1}
            Team(0)                                   other; Team(1)
}                                         }
```

Basically, both *Team0* and *Team1* describe the same behaviour as *Team*, however, there are two important differences:

(1) Some actions are hidden, *i.e.,* they are renamed to the internal action τ. This is the case for actions *kick*, *self*, and *goal* in either process. The action τ is invisible to other (parallel) components and thus cannot be subject to synchronization.
(2) In both processes, actions are relabelled: in case of *Team*0, action *other* is renamed into *0to1*, and *gotBall* into *1to0*. Similarly, in *Team1*, *other* is renamed into *1to0*, and *gotBall* into *0to1*. The exception *foul* is renamed into *foul0* and *foul1*, respectively.

Both processes *Team0* and *Team1* can now be put together to describe a complete match as defined by the process *Match*. The processes *Team0* and *Team1* are put in parallel inside a par construct. Both processes run independently from each other, but are synchronizing on actions with the same name. In case of *Team0* and *Team1* these actions are *0to1* and *1to0*. This explains why action *other* in process *Team0* and action *gotBall* in *Team1* have been renamed to *0to1*: both processes synchronize on these actions, and model the passing of the ball from team 0 to team 1. The same holds for the reverse direction.

```
process Match() {
    try {
        par {
            :: Team0()
            :: Team1()
        }
    }
    catch(foul0){
        players[1] −= 1; Match()
    }
    catch(foul1){
        players[0] −= 1; Match()
    } }
```

The parallel composition forms the *try-block* of the enclosing try/catch construct, which is used for the handling of exceptions. Exceptions can be *caught* by an exception handler, which is introduced by the keyword catch. In the given example, there are two exception handlers, one for exception *foul0*, the other for *foul1*. In both handlers, the number of players of the respective opposite team is decremented by one, and the *Match* process is restarted.

Note that in the given specification, *Team0* always gets the ball first, since *Team1* is started unconditionally with action *other*. This is a simple way of avoiding that both teams wait for their competitor to pass the ball, although neither of them possesses it. This would yield a deadlock.

```
process Referee() {
    clock $x = 0$;
    par {
        :: when($x == 90$)
            urgent($x == 90$)
                throw gameover
        :: when($players[0] == 0 \lor players[1] == 0$)
            urgent($players[0] == 0 \lor players[1] == 0$)
                throw noplayers
    }
}
```

The process *Match* describes already a soccer match quite accurately, however, two things have to be taken care of: first, process *Match* describes a never-ending match, and second, it is possible for a team to have a negative number of players. Both aspects are unrealistic. To take care of these situations, a process *Referee* is introduced that monitors the time that has passed so far. It also ensures that there are always a non-negative number of players on the field. This is again done by means of exceptions: exception *noplayers* is thrown if a team has lost all its players, whereas exception *gameover* is raised when 90 minutes have passed. The use of the urgent() construct guarantees that the match is ended once one of these conditions hold.

```
try{
    par{
        :: Match()
        :: Referee()
    }
}
catch(gameover){
    swap_shirts
}
catch(noplayers){
    $\tau$
}
```

Finally, the complete specification of the soccer match is a parallel composition of the processes *Referee* and *Match*, nested inside a try-catch construct to take care of the exceptions *noplayers* and *gameover*. In case no players are left, the game simply stops. In case the game is played to its end, the remaining players exchange their shirts.

3 Stochastic Timed Automata

The semantics of MODEST is defined using an operational model which is based on timed automata [3,12], a well-studied and tool-supported symbolic model for real-time systems. Timed automata extend labelled transition systems with clocks to measure the elapse of time, guards (that possibly refer to clocks) to specify when an action is enabled, and urgency constraints to force actions to happen at some ultimate time instant. As timed automata do not have means to support probabilistic branching, such mechanisms have to be incorporated for our purposes. To accommodate for random delays, samples from probability distributions can be assigned to variables. By comparing clocks to such variables, actions can be delayed by a random amount of time. This section defines the operational model, called *stochastic timed automata*, and justifies the main differences with some existing models. We start by defining expressions.

Expressions and assignments. We distinguish the following syntactic categories:

- Var is the set of (typed) *variables* ranged over by x, y and z. It is sometimes convenient to distinguish the subset $\mathsf{Ck} \subseteq \mathsf{Var}$ of *clock variables*, i.e., the variables of type clock that are used to measure the elapse of time.
- Exp is the set of *expressions* containing variables (in Var). It is ranged over by e. We distinguish the following subcategories of expressions:
  - · $\mathsf{Sxp} \subseteq \mathsf{Exp}$ is the set of *sampling expressions* of the form $\mathsf{sample}(F)$, with the intended meaning that it samples a value for distinguished (random) variable $\xi \notin \mathsf{Var}$ according to distribution F. Formally, F is a function on ξ (and possibly variables in Var) such that for every instance of variables in Var, F is a distribution function on ξ.
  - · $\mathsf{Bxp} \subseteq \mathsf{Exp}$ is the set of *Boolean expressions*. These expressions do not contain sampling expressions. It is ranged over by b, d, and g.
  - · $\mathsf{Axp} \subseteq \mathsf{Exp}$ is the set of *arithmetic expressions*. These expressions do not contain sampling expressions. It is ranged over by w.
- Asgn is the set of *assignments* ranged over by A. An assignment is a function that maps variables onto expressions (in Exp). $\{\!=\ x_1 = e_1, \ldots, x_n = e_n\ =\!\}$ denotes the unique assignment $A \in \mathsf{Asgn}$ defined by $A(x_i) = e_i$ (for $0 < i \leqslant n$), and $A(y) = y$ if $y \neq x_i$ for all i.
- Act is a set of *action names* ranged over by a.

Variables, assignments and expressions serve the usual purpose. Variables may occur in expressions and expressions may be assigned to them. Sample expressions are used to draw samples from distributions and are used to model random delays. For modelling convenience, some standard probability distribution functions such as EXP(rate) and NORMAL$(\mathit{avr}, \mathit{dev_stndr})$, are supported.

Boolean expressions are used in guards and urgency constraints. Actions play the same role as in labelled transition systems.

Example 1.
Some expressions and assignments in our soccer example are, for instance, $\{\!=\ score[team] \mathrel{+}= 1, players[team] \mathrel{-}= 1\ =\!\}$, which is an assignment, and $players[0] == 0 \vee players[1] == 0$ which is a Boolean expression. An example of an arithmetic expression is $(team + 1)\%2$, and $\textsc{Uniform}(2, 5)$ is a sample expression, which is to be understood as an abbreviation of $\mathsf{sample}(F)$ where

$$F(\xi) = \begin{cases} 0 & \text{if } \xi < 2, \\ (\xi - 2)/3 & \text{if } \xi \in [2, 5], \\ 1 & \text{if } \xi > 5. \end{cases}$$

□

The model. A stochastic timed automaton consists of control states, called *locations*, that are connected by edges. For ease of understanding, let us first assume that there is no probabilistic branching. In this simple case, edges would be labelled by four attributes:

(i) an action a to be performed,
(ii) a guard g specifying when the edge is enabled,
(iii) an urgency constraint d specifying when the edge ultimately should be executed (if at all), and
(iv) a set A of assignments to be carried out atomically.

The edge $\xrightarrow{a,g,d,A}$ in location s is *enabled* whenever the system is in control state s and guard g holds given the current values of the variables – including the clocks. If, in addition, urgency constraint d holds, then the system is obliged to take the edge $\xrightarrow{a,g,d,A}$ before time progresses. Thus, time may progress in location s as long as no urgency constraint of one of its outgoing edges holds. On "executing" $s \xrightarrow{a,g,d,A} s'$, action a is performed, the assignments in A are carried out atomically, and the system moves to control state s'. Note that by means of this mechanism, variables may be tested (in a guard) and updated (in an assignment) in a single atomic step. This test-and-set mechanism is, for instance, useful for modelling locks and semaphores, see e.g., [6, pp. 43].

In order to deal with probabilistic branching, the situation is somewhat more complicated. The target of an edge is not just a control state, but rather a probability distribution over control states, or more precise, a probability distribution over pairs $\langle A, s\rangle$ of assignments and control states. This is because different probabilistic branches may trigger different assignments and successor control states in one edge. The actual probability for each such pair is determined by weights. Suppose s can either move to control state s' (with weight w') or to s'' (with weight w'') where $s' \neq s''$, while performing assignment A' and A'', respectively. If weights w' and w'' are just constants, the probability to "move" to $\langle A', s' \rangle$ equals $\frac{w'}{w'+w''}$, and the probability to move to $\langle A'', s'' \rangle$ is $\frac{w''}{w'+w''}$. In this case, the likelihoods can be determined easily. As we support weights that are expressions containing variables – possibly even clocks – the situation is a bit more complicated. Rather than working with constant weights, *weight expressions* are used. Intuitively speaking, these are a kind of symbolic probability distributions, over pairs of assignments and (target) control states. On taking the edge $s \xrightarrow{a,g,d} \mathcal{W}$, the system moves to control state s' assigning values according to assignment A' with a probability that is determined by $\mathcal{W}(\langle A', s' \rangle)$. For the above example with two possible successor control states, this probability is $\frac{v(w')}{v(w')+v(w'')}$ for control state s' (and similar for s''), where $v(w)$ denotes the value of w after instantiating the variables occurring in w given the current variable valuation v, i.e., the valuation in control state s. In the sequel, Wxp denotes the set of weight expressions (on pairs of assignments and control states). Formally, a weight expression $\mathcal{W}$ is a mapping from an assignment and a control state onto an arithmetic expression (in Axp) and it only makes sense in valuation v if $v(\mathcal{W}(A, s)) \geqslant 0$ for all A and s, and $v(\mathcal{W}(A, s)) > 0$ for some A and s.

We are now in a position to formally define the semantical model for Modest. Control states are, from now on, referred to as locations.

Definition 1.
A *stochastic timed automaton* (*STA*, for short) is a triple $(\mathsf{Loc}, \mathsf{Act}, \rightarrow)$, where Loc is a set of *locations* and $\rightarrow \subseteq \mathsf{Loc} \times (\mathsf{Act} \times \mathsf{Bxp} \times \mathsf{Bxp}) \times \mathsf{Wxp}$ is the edge relation. □

Example 2.
Figure 1 depicts a stochastic timed automaton with 7 locations (i.e., control states). The automaton has a distinguished initial location indicated by an incoming arrow without source. Empty assignments, **tt**rue guards and **ff**alse urgency constraints are omitted from edges. Most edges lead to trivial weight expressions, where only one pair of assignment and location gets probability 1 assigned. On the occurrence of action *kick* a probabilistic choice appears with four branches, indicated by the circle fragment connecting the weighted alternatives of assignments and locations. The automaton in fact corresponds to the process instantiation *Play*(0) of our running soccer example where, for convenience, *players*[*i*] is abbreviated by *p*[*i*]. In Section 4, it will be explained how an STA is obtained from a Modest specification. □

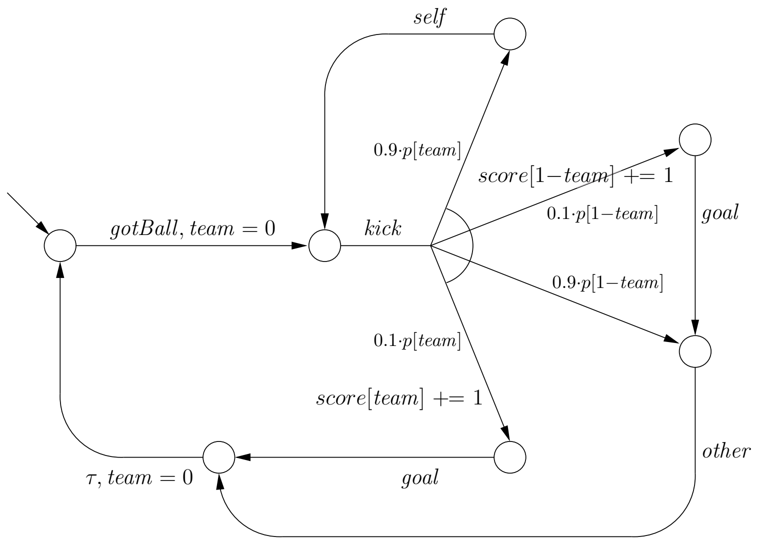


Fig. 1. Stochastic timed automaton for process invocation *Play*(0).

It is worthwhile to remark that *STA* provide a *symbolic* framework to represent stochastic timed (and real-time) behaviour in much the same way as timed automata represent real-time behaviour in a symbolic manner. Whereas the semantics of timed automata is typically described by (infinite) timed transition systems, the interpretation of a stochastic timed automaton is defined in terms of (infinite) timed *probabilistic* transition systems. This is further explained in Section 5. In particular, this second level of semantics defines exactly what the (probabilistic) interpretation of sampling is, and how weight expressions are interpreted probabilistically. As a second remark, we like to emphasize that *STA* have been developed to provide semantics to MODEST. These automata are closed under all operators of the language, most notably parallel composition (with synchronization).

4 Formal Definition of Modest

4.1 Syntax

This subsection formally defines the syntax of MODEST. We assume that the set of actions Act is partitioned into: a set PAct of *patient* actions, a set IAct of *impatient* actions, a set Excp of *exception names*, the *unhandled error* action $\bot$, the break action $\flat$, and the unobservable (or silent) action τ. The difference between patient and impatient actions becomes clear when defining the semantics of parallel composition. Exception names are distinguished actions that are used for raising exceptions. Action $\flat$ is used to abort a loop, and τ is the unobservable action that is standard in most process calculi to model internal computations.

We distinguish processes and process behaviours. A process is defined by:

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

where $x_i \in \mathsf{Var}$, t_i $(0 < i \leqslant k)$ are valid types, dcl is a sequence of declarations possibly including process definitions, *ProcName* is a process name and P is a process behaviour. For convenience, we will not dwell upon the syntax of declarations and write $\textsf{process}\ \mathit{ProcName}(x_1, \ldots, x_k)\ \{P\}$ instead in the sequel.

Process behaviours are defined as follows. Let $w_i \in \mathsf{Axp}$, $e_i \in \mathsf{Exp}$ (for $0 < i \leqslant k$), $b \in \mathsf{Bxp}$, and asgn_i an assignment of the form $\{\text{=}\ x_1 = e_1,\ \ldots\ ,\ x_n = e_n\ \text{=}\}$. Furthermore, let $\mathit{act} \in \mathsf{PAct} \cup \mathsf{IAct} \cup \{\tau\}$ be an action as in standard process calculi (i.e., neither an exception, $\flat$, nor the unhandled error $\bot$), $H \subseteq \mathsf{PAct} \cup \mathsf{IAct}$ be a set of observable actions, and $\mathit{excp}, \mathit{excp}_i \in \mathsf{Excp}$ be exception names (for $0 < i \leqslant k$). Finally, let I and G be vectors of equal length in $\mathsf{Act} \setminus \{\flat, \bot\}$ such that all elements in I are pairwise different and different from τ. The intention is that the mapping $I(j) \mapsto G(j)$, for $0 \leqslant j < \#I$, defines a *relabelling* function.

A process behaviour P is constructed according to the following grammar:

$$\begin{aligned}
P ::= {} & \textsf{stop} \;\Big|\; \textsf{abort} \;\Big|\; \textsf{break} \;\Big|\; \mathit{act} \;\Big|\; \textsf{when}(b)\ P \;\Big|\; \textsf{urgent}(b)\ P \;\Big|\; P_1;\ P_2 \;\Big| \\
& \textsf{alt}\{::P_1\ \ldots\ ::P_k\} \;\Big|\; \textsf{do}\{::P_1\ \ldots\ ::P_k\} \;\Big|\; \textsf{par}\{::P_1\ \ldots\ ::P_k\} \;\Big| \\
& \mathit{act}\ \textsf{palt}\ \{:w_1{:}\mathit{asgn}_1;\ P_1\ \ldots\ :w_k{:}\mathit{asgn}_k;\ P_k\} \;\Big|\; \mathit{ProcName}(e_1, \ldots, e_k) \;\Big| \\
& \textsf{throw}(\mathit{excp}) \;\Big|\; \textsf{try}\{P\}\ \textsf{catch}\ \mathit{excp}_1\ \{P_1\}\ \ldots\ \textsf{catch}\ \mathit{excp}_k\ \{P_k\} \;\Big| \\
& \textsf{relabel}\ \{I\}\ \textsf{by}\ \{G\}\ P \;\Big|\; \textsf{extend}\ \{H\}\ P
\end{aligned}$$

Let us briefly describe the syntactic constructs. stop is the behaviour that cannot perform any action and can be viewed as a deadlocked process. abort is the behaviour that has aborted. Whereas stop cannot perform any action, the behaviour abort is only able to perform $\bot$ (*ad infinitum*). break is the behaviour that can only perform a break action $\flat$ and then successfully terminates. *act* can perform a visible or invisible action, and then successfully terminates. when(b) P behaves like P in case b holds. urgent(b) P imposes an urgency constraint b on the behaviour P, i.e., P is forced to be executed as soon as b holds. alt and do are the usual alternative and iterative statements. In case several alternatives in an alt-statement are enabled one of these alternatives is non-deterministically chosen. If no alternative is enabled, the statement blocks and waits until one of its alternatives becomes enabled. Iterations successfully terminate on the occurrence of a break action. Behaviours are put in parallel using the par-construct. Parallel behaviours have to perform common actions jointly, and perform other actions autonomously. palt-behaviours execute action *act*, perform the assignments in $asgn_i$, and evolve into behaviour P_i in an atomic way. The probability to perform the i-th assignment and the move to the i-th behaviour is determined by the weight expressions w_1 through w_k. Behaviour throw($excp$) raises an exception with name *excp*. Behaviour try$\{P\}$ catch $excp_1$ $\{P_1\}$ ... catch $excp_k$ $\{P_k\}$ behaves like P, except that on raising an exception $excp_i$ in P this is handled by the behaviour P_i. Processes can be invoked in the usual way. Finally, relabel is the usual relabeling as in traditional process algebra and allows for the renaming of visible actions and exception names. Behaviour extend $\{H\}P$ behaves like P. The only effect is that the actions on which P is forced to synchronize (with a parallel behaviour) are extended with the actions in H.

4.2 Operational Semantics

The operational semantics of behaviour P is defined in terms of the stochastic timed automaton $(\mathsf{Loc}, \mathsf{Act}, \twoheadrightarrow)$ where Act is the set of actions occurring in P, Loc is the set of behaviors that are derivable from P—locations are thus Modest terms—using the edge relation $\twoheadrightarrow$, and $\twoheadrightarrow$ is the smallest relation that is defined by the inference rules defined in the remainder of this section.

Let $\mathcal{D}(r)$ denote the *deterministic* weight expression defined by $\mathcal{D}(A, s) = 1$ and $\mathcal{D}(A', s') = 0$ for all $A', s' \neq A, s$. Intuitively speaking, the assignments A and target location s are chosen with probability 1.

Basic actions. Behaviour stop does not perform any activity and thus does not produce any transition.

abort is a process that indicates an unhandled error by persistent executions of action $\perp$. No assignments are executed. Its inference rule reads:

$$\mathsf{abort} \xrightarrow{\perp, \mathbf{tt}, \mathbf{ff}} \mathcal{D}(\varnothing, \mathsf{abort})$$

Action $\perp$ is always enabled as the guard is true, and is not forced to occur at any time as the urgency constraint is false.

break can perform the break action $\flat$ without restriction and then successfully terminates. We use the symbol $\surd$ to denote the successfully terminated process. This process (that cannot be specified syntactically) does not have any transition (like stop), but is used in other inference rules to distinguish successfully terminated processes from non-terminated ones. The inference rule for break reads:

$$\mathsf{break} \xrightarrow{\flat, \mathbf{tt}, \mathbf{ff}} \mathcal{D}(\varnothing, \surd)$$

act performs action *act* with no restriction and then successfully terminates. No assignments are executed.

$$act \xrightarrow{act,\mathbf{tt},\mathbf{ff}} \mathcal{D}(\varnothing, \surd)$$

Actions indicate a particular activity a process intends to perform. If the action *act* is visible it may be used for synchronization purposes.

Conditions. $\mathsf{when}(b)\ P$ restricts the first activity of P to be performed only whenever b holds. As a consequence, guards from every edge leaving location P are strengthened with b:

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

Recall that $\mathcal{W}$ denotes a weight expression, i.e., a "symbolic" probability distribution.

$\mathsf{urgent}(b)\ P$ enforces the first activity of P to be urgent whenever b holds. It imposes, so to speak, an extra urgency constraint b on the initial step of P. So, if d is the urgency constraint of an edge leaving P, the new urgency constraint is $d \vee b$, i.e., either the transition becomes urgent because it was required to become urgent in P, or because of the new requirement b. The inference rule reads:

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

Process instantiation. Let $\mathsf{process}\ \mathit{ProcName}(x_1, \ldots, x_k)\ \{P\}$ be a process that is part of the current specification. Without loss of generality, we assume variable names x_1 through x_k to be unique[3]. The process invocation $\mathit{ProcName}(e_1, \ldots, e_k)$ behaves like P where variables $x_1, \ldots, x_k$ are instantiated with the values of expressions $e_1, \ldots, e_k$ under the current valuation of the variables.

To accomplish this *call-by-value* approach, just before executing $\mathit{ProcName}(e_1, \ldots, e_k)$, the assignments $x_1 = e_1, \ldots, x_k = e_k$ are performed atomically[4]. Operationally speaking, all incoming edges of a process invocation are equipped with the assignments to the parameters of the (possible) next process invocation. Since $\mathit{ProcName}(e_1, \ldots, e_k)$ may occur within another statement, e.g., as an alternative in an alt- or a do-statement, a function $\mathbf{A}$ is used to collect

[3] This can always be established by means of renaming.

[4] It is important to realize that a call-by-name strategy is inadequate for MODEST—unlike the more traditional process algebra like CCS, CSP and LOTOS—due to the presence of shared variables. Using a call-by-name paradigm would lead to unintended read-write interferences.

Table 1
The assignment collecting function

$\mathbf{A}(P) = \varnothing$	if P has one of the following forms:
	act, act palt $\{:w_1:asgn_1;\ P_1 \ \ldots\ :w_k:asgn_k;\ P_k\}$,
	stop, abort, $\mathsf{throw}(excp)$, break
$\mathbf{A}(P) = \mathbf{A}(Q)$	if P has one of the following forms:
	$Q;\ Q'$, $\mathsf{when}(b)\ Q$, $\mathsf{urgent}(b)\ Q$,
	$\mathsf{try}\{Q\}\ \mathsf{catch}\ excp_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ excp_k\ \{P_k\}$,
	$\mathsf{relabel}\ \{I\}\ \mathsf{by}\ \{G\}\ Q$, $\mathsf{extend}\,\{H\}\ Q$
$\mathbf{A}(P) = \bigcup_{i=1}^{k} \mathbf{A}(P_i)$	if P has one of the following forms:
	$\mathsf{alt}\{::P_1\ \ldots\ ::P_k\}$, $\mathsf{do}\{::P_1\ \ldots\ ::P_k\}$, $\mathsf{par}\{::P_1\ \ldots\ ::P_k\}$
$\mathbf{A}(ProcName(e_1,\ldots,e_k)) = \{x_1 = e_1,\ldots,x_k = e_k\} \cup \mathbf{A}(Q)$	
	if $\mathsf{process}\ ProcName(x_1,\ldots,x_k)\{Q\}$

all such necessary assignments (cf. Table 1). This function **A** is not used in the inference rule of process instantiation but is necessary for edges that may lead to a process call, cf. the inference rules for palt, exception handling, and sequential composition furtheron in this section.

The inference rule for process instantiation is as follows:

$$\frac{P \xrightarrow{a,g,d} \mathcal{W}}{ProcName(e_1,\ldots,e_k) \xrightarrow{a,g,d} \mathcal{W}} \quad \text{if } \mathsf{process}\ ProcName(x_1,\ldots,x_k)\{P\}.$$

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

$$\frac{P_i \xrightarrow{a,g,d} \mathcal{W}_i \qquad (0 < i \leqslant k)}{\mathsf{alt}\{::P_1\ \ldots\ ::P_k\} \xrightarrow{a,g,d} \mathcal{W}_i}$$

Sequential composition. $P;\ Q$ executes P until it successfully terminates. When P terminates, it continues with the execution of Q:

$$\frac{P \xrightarrow{a,g,d} \mathcal{W}}{P;\ Q \xrightarrow{a,g,d} \mathcal{W} \circ \mathbf{M}_{;}^{-1}}$$

where

$$\mathbf{M}_;(A,P') \overset{\text{def}}{=} \begin{cases} \langle A, P';\, Q\rangle & \text{if } P' \neq \surd \\ \langle A \cup \mathbf{A}(Q), Q\rangle & \text{if } P' = \surd \end{cases}$$

The assignments that are carried out if P successfully terminates are those that P performed on terminating together with $\mathbf{A}(Q)$. The latter assignments are necessary whenever one of the possible initial behaviors of Q is a process invocation. This is used to realize a call-by-value approach as discussed before. Note that the inverse of $\mathbf{M}_;$ is used in $\mathcal{W} \circ \mathbf{M}_;^{-1}$, to retrieve the wheight expresion for the sequential composition from the the wheigts assigned by $\mathcal{W}$ to the first component of a sequential composition.

Loop. Behaviour $\mathsf{do}\{::P_1 \ldots ::P_k\}$ repeatedly chooses an alternative P_i in a nondeterministic manner. It terminates whenever one of the processes P_i executes a break action ($\flat$). The semantics of do is defined using the auxiliary operator auxdo which has two arguments: the actual behaviour and the behaviour that needs to be resumed on successful termination of the loop behaviour. We have:

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

Behaviour $\mathsf{auxdo}\{P\}\{Q\}$ behaves like P as long as no break actions are performed and terminates successfully if P performs a break (i.e., $\flat$). If P, however, successfully terminates, behaviour Q is resumed.

In the usual non-probabilistic setting, where transitions have behaviours as targets —rather than (symbolic) probability distributions—, the intuitive behaviour above would be encoded by the following three inference rules:

$$\frac{P \xrightarrow{\flat,g,d} P'}{\mathsf{auxdo}\{P\}\{Q\} \xrightarrow{\tau,g,d} \surd}$$

$$\frac{P \xrightarrow{a,g,d} P' \qquad (a \neq \flat \wedge P' \neq \surd)}{\mathsf{auxdo}\{P\}\{Q\} \xrightarrow{a,g,d} \mathsf{auxdo}\{P'\}\{Q\}} \qquad \frac{P \xrightarrow{a,g,d} \surd \qquad (a \neq \flat)}{\mathsf{auxdo}\{P\}\{Q\} \xrightarrow{a,g,d} \mathsf{auxdo}\{Q\}\{Q\}}$$

The first rule represents the break of the loop: as soon as the body loop executes a break action, the loop terminates successfully. The other two inference rules represent the execution within the loop. In particular, the second rule states than once the loop body terminates its execution successfully, it should be resumed from the beginning.

In a probabilistic setting it may happen that the loop body successfully terminates with probability p or it continues doing something else with probability $1-p$. In this sense, the first two rules are combined in only one that considers these two cases in one probability distribution. In our case, probabilities are represented symbolically by weight expressions.

The inference rules are:

$$\frac{P \xrightarrow{\flat,g,d} \mathcal{W}}{\mathsf{auxdo}\{P\}\{Q\} \xrightarrow{\tau,g,d} \mathcal{D}(\varnothing,\surd)} \qquad \frac{P \xrightarrow{a,g,d} \mathcal{W} \qquad (a \neq \flat)}{\mathsf{auxdo}\{P\}\{Q\} \xrightarrow{a,g,d} \mathcal{W} \circ \mathbf{M}_{\mathsf{do}}^{-1}}$$

where

$$\mathbf{M}_{\mathsf{do}}(A,P') \overset{\mathrm{def}}{=} \begin{cases} \langle A, \mathsf{auxdo}\{P'\}\{Q\}\rangle & \text{if } P' \neq \surd \\ \langle A, \mathsf{auxdo}\{Q\}\{Q\}\rangle & \text{if } P' = \surd \end{cases}$$

The right inference rule corresponds to the loop break and is as expected. The left inference rule applies to the occurrence of an action of P that differs from $\flat$. It is the obvious generalisation of the two nonprobabilistic rule. It states that the loop behaves as $\mathsf{auxdo}\{P'\}\{Q\}$, whenever P evolves into P' unless $P' \neq \surd$. If P instead successfully terminates, the loop resumes from its beginning $\mathsf{auxdo}\{Q\}\{Q\}$.

Relabelling and hiding. The semantics for relabelling is as usual in traditional process algebra: $\mathsf{relabel}\ \{a_1,\ldots,a_k\}\ \mathsf{by}\ \{a'_1,\ldots,a'_k\}\ P$ behaves like P except that every observable action or exception a_i is renamed by the corresponding a'_i:

$$\frac{P \xrightarrow{a,g,d} \mathcal{W} \qquad f = [a_1/a'_1,\ldots,a_k/a'_k]}{\mathsf{relabel}\ \{a_1,\ldots,a_k\}\ \mathsf{by}\ \{a'_1,\ldots,a'_k\}\ P \xrightarrow{f(a),g,d} \mathcal{W} \circ \mathbf{M}_{\mathsf{rel}}^{-1}}$$

where

$$\mathbf{M}_{\mathsf{rel}}(A,P') \overset{\mathrm{def}}{=} \begin{cases} \langle A, \mathsf{relabel}\ \{a_1,\ldots,a_k\}\ \mathsf{by}\ \{a'_1,\ldots,a'_k\}\ P'\rangle & \text{if } P' \neq \surd \\ \langle A, \surd\rangle & \text{if } P' = \surd \end{cases}$$

Alphabet extension. extend just extends the alphabet of a process (cf. Table 2) and does not affect behaviour:

$$\frac{P \xrightarrow{a,g,d} \mathcal{W}}{\mathsf{extend}\ \{act_1,\ldots,act_k\}\ P \xrightarrow{a,g,d} \mathcal{W} \circ \mathbf{M}_{\mathsf{ext}}^{-1}}$$

where

$$\mathbf{M}_{\mathsf{ext}}(A,P') \overset{\mathrm{def}}{=} \begin{cases} \langle A, \mathsf{extend}\ \{act_1,\ldots,act_k\}\ P'\rangle & \text{if } P' \neq \surd \\ \langle A, \surd\rangle & \text{if } P' = \surd \end{cases}$$

Exception handling. An exception $excp \in \mathsf{Excp}$ is raised by the simple behaviour $\mathsf{throw}(excp)$:

$$\mathsf{throw}(excp) \xrightarrow{excp,\mathbf{tt},\mathbf{ff}} \mathcal{D}(\varnothing, \mathsf{abort})$$

Behaviour $Q \equiv \mathsf{try}\{P\}\,\mathsf{catch}\,\mathit{excp}_1\ \{P_1\}\ \dots\ \mathsf{catch}\,\mathit{excp}_k\ \{P_k\}$ behaves like P as long as P does not raise an exception excp_i $(0 < i \leqslant k)$. If P raises exception excp_i, it behaves as P_i, i.e., P_i is the exception handler of excp_i. Unhandled exceptions are not handled by any P_i and thus propagate outside Q (where they might be handled):

$$\frac{P \xrightarrow{a,g,d} \mathcal{W} \qquad (a \notin \{\mathit{excp}_1, \dots, \mathit{excp}_k\})}{\mathsf{try}\{P\}\ \mathsf{catch}\ \mathit{excp}_1\ \{P_1\}\ \dots\ \mathsf{catch}\ \mathit{excp}_k\ \{P_k\} \xrightarrow{a,g,d} \mathcal{W} \circ \mathbf{M}_{\mathsf{try}}^{-1}}$$

where

$$\mathbf{M}_{\mathsf{try}}(A, P') \overset{\text{def}}{=} \begin{cases} \langle A, \mathsf{try}\{P'\}\ \mathsf{catch}\ \mathit{excp}_1\ \{P_1\}\ \dots\ \mathsf{catch}\ \mathit{excp}_k\ \{P_k\}\rangle & \text{if } P' \neq \surd \\ \langle A, \surd\rangle & \text{if } P' = \surd \end{cases}$$

The inference rule for the case in which an exception is handled is:

$$\frac{P \xrightarrow{\mathit{excp}_i,g,d} \mathcal{W} \qquad (0 < i \leqslant k)}{\mathsf{try}\{P\}\ \mathsf{catch}\ \mathit{excp}_1\ \{P_1\}\ \dots\ \mathsf{catch}\ \mathit{excp}_k\ \{P_k\} \xrightarrow{\tau,g,d} \mathcal{D}(\mathbf{A}(P_i), P_i)}$$

Note that although raising the exception excp_i results in an unhandled error (cf. the inference rule for throw), the resulting behaviour of the entire expression is P_i, the handler of excp_i. To realize the call-by-value mechanism, the assignments for a (possible) process instantiation in P_i are considered.

Probabilistic prefix. Behaviour $\mathit{act}\ \mathsf{palt}\ \{{:}w_1{:}\mathit{asgn}_1;\ P_1\ \dots\ {:}w_k{:}\mathit{asgn}_k;\ P_k\}$ performs action act without restriction, randomly selects an alternative i according to the weights $w_1, \dots, w_k$, performs an assignment according to asgn_i, and evolves into P_i.

Weights are arithmetic expressions (not containing sampling expressions) requiring particular treatment. A probability distribution is obtained by dividing a given weight by the sum of all weights in the palt-construct, i.e., $\frac{w_i}{w_1+\cdots+w_k}$ is the probability of performing asgn_i while evolving into P_i—provided there is no index $j \neq i$ with the same assignments and evolving behaviour. Therefore, w_i must be non-negative and $w_1 + \cdots + w_k$ non-zero. Since weights may contain variables, these conditions are checked at "run time", i.e., in the concrete semantics, cf. Section 5.

Let predicates $\mathit{neg} \equiv \bigwedge_{i=1}^{k} w_i < 0$ and $\mathit{zero} \equiv \sum_{i=1}^{k} w_i = 0$. The inference rule covering the normal situation is:

$$\mathit{act}\ \mathsf{palt}\ \{{:}w_1{:}\mathit{asgn}_1;\ P_1\ \dots\ {:}w_k{:}\mathit{asgn}_k;\ P_k\} \xrightarrow{\mathit{act},\neg(\mathit{neg} \vee \mathit{zero}),\mathbf{ff}} \mathcal{W}$$

with $\mathcal{W}$ being the weight expression:

$$\mathcal{W}(\mathit{asgn}_i \cup \mathbf{A}(P_i), P_i) \overset{\text{def}}{=} \textstyle\sum_{j=1}^{k} \mathbf{I}(i,j) \cdot w_j$$

where

$$\mathbf{I}(i,j) \stackrel{\text{def}}{=} \textbf{if } (asgn_i \cup \mathbf{A}(P_i) = asgn_j \cup \mathbf{A}(P_j) \wedge P_i = P_j) \textbf{ then } 1 \textbf{ else } 0.$$

The guard $\neg(neg \vee zero)$ ensures that the weights are legal.

Note that besides the assignments $asgn_i$ also the (possible) assignments introduced by process instantiation in P_i are performed.

The two abnormalities that might happen during execution are that one of the weight expressions evaluates to a negative number, or that the sum of all weights is zero. The following two axioms deal with these situations:

$$act \ \mathsf{palt}\ \{{:}w_1{:}asgn_1;\ P_1\ \ldots\ {:}w_k{:}asgn_k;\ P_k\} \xrightarrow{neg\_weight,neg,\mathbf{ff}} \mathcal{D}(\varnothing, \mathsf{abort})$$

$$act \ \mathsf{palt}\ \{{:}w_1{:}asgn_1;\ P_1\ \ldots\ {:}w_k{:}asgn_k;\ P_k\} \xrightarrow{no\_weight,zero,\mathbf{ff}} \mathcal{D}(\varnothing, \mathsf{abort})$$

The labels *neg_weight* and *no_weight* are predefined *exceptions*. It is therefore possible to catch them and handle the abnormal situations, if necessary.

Parallel composition. Behaviour $\mathsf{par}\{{::}P_1\ \ldots\ {::}P_k\}$ runs $P_1, \ldots, P_k$ concurrently, while synchronizing them on the intersected alphabet, thus allowing for multi-way synchronization. The alphabet of a process P is the set $\alpha(P) \subseteq \mathsf{PAct} \cup \mathsf{IAct}$ of all actions P recognizes (cf. Table 2). 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 [40,11]. 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 $B_j = (\bigcup_{i=1}^{j} \alpha(P_i)) \cap \alpha(P_{j+1})$. Note that B_j only contains observable actions, i.e., $\bot$, $\flat$, τ, and exception names do not belong to B_j. The behaviour of $||_B$ is formally defined in the following. Action $a \notin B$ (which is not intended to synchronize) can be performed autonomously, i.e., without the cooperation of the other parallel component:

$$\frac{P_1 \xrightarrow{a,g,d} \mathcal{W} \qquad (a \notin B)}{P_1 \,||_B\, P_2 \xrightarrow{a,g,d} \mathcal{W} \circ \mathbf{M}^{-1}_{\mathsf{par}P_2}} \qquad\qquad \frac{P_2 \xrightarrow{a,g,d} \mathcal{W} \qquad (a \notin B)}{P_1 \,||_B\, P_2 \xrightarrow{a,g,d} \mathcal{W} \circ \mathbf{M}^{-1}_{\mathsf{par}P_1}}$$

with $\mathbf{M}_{\mathsf{par}P}(A, P') \stackrel{\text{def}}{=} \langle A, P' \,||_B\, P\rangle$, where $\surd \,||_B\, \surd = \surd$. Note that a parallel composition successfully terminates whenever all its components do so.

Modest provides two synchronization modes which depend on the action type. An action can be either patient or impatient. A process that wants to synchronize on a patient action always waits for its partner to be ready. Accordingly, its urgency constraint needs to be *relaxed* to the requirements of the

Table 2
Alphabet of a MODEST term

$\alpha(\mathsf{stop}) = \alpha(\mathsf{abort}) = \alpha(\mathsf{break}) = \alpha(\mathsf{throw}(excp)) = \varnothing$

$\alpha(act) = \{act\} - \{\tau\}$

$\alpha(act\ \mathsf{palt}\ \{:w_1:asgn_1;\ P_1\ \ldots\ :w_k:asgn_k;\ P_k\}) = \alpha(act) \cup \bigcup_{i=1}^{k} \alpha(P_i)$

$\alpha(\mathsf{when}(b)\ P) = \alpha(\mathsf{urgent}(b)\ P) = \alpha(P)$

$\alpha(\mathsf{alt}\{::P_1\ \ldots\ ::P_k\}) = \alpha(\mathsf{do}\{::P_1\ \ldots\ ::P_k\}) = \alpha(\mathsf{par}\{::P_1\ \ldots\ ::P_k\}) = \bigcup_{i=1}^{k} \alpha(P_i)$

$\alpha(P_1;\ P_2) = \alpha(P_1) \cup \alpha(P_2)$

$\alpha(\mathsf{try}\{P\}\ \mathsf{catch}\ excp_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ excp_k\ \{P_k\}) = \alpha(P) \cup \bigcup_{i=1}^{k} \alpha(P_i)$

$\alpha(\mathsf{relabel}\ \{a_1,\ldots,a_k\}\ \mathsf{by}\ \{a'_1,\ldots,a'_k\}\ P) = \alpha(P)[a_1/a'_1,\ldots,a_k/a'_k] - \{\tau\}$

$\alpha(\mathsf{extend}\ \{act_1,\ldots,act_k\}\ P) = \alpha(P) \cup \{act_1,\ldots,act_k\}$

$\alpha(ProcName(e_1,\ldots,e_k)) = \alpha(P)$ provided $\mathsf{process}\ ProcName(x_1,\ldots,x_k)\ \{P\}$

partner. As a consequence, an urgency constraint in a patient synchronization is met whenever *all* the components meet their respective urgency constraints (i.e., the synchronization meets the *conjunction* of the urgency constraints):

$$\frac{P_1 \xrightarrow{a,g_1,d_1} \mathcal{W}_1 \quad P_2 \xrightarrow{a,g_2,d_2} \mathcal{W}_2 \qquad (a \in B \cap \mathsf{PAct})}{P_1 \,||_B\, P_2 \xrightarrow{a,g_1\wedge g_2,d_1\wedge d_2} (\mathcal{W}_1 \times \mathcal{W}_2) \circ \mathbf{M}_{\mathsf{par}}^{-1}}$$

However, a process that intends to synchronize on an impatient action is *not willing* to wait for the partner. Therefore, an urgency constraint in an impatient synchronization should be met as soon as *one* of the synchronizing components meets its urgency constraints, i.e., the synchronization meets the *disjunction* of the urgency constraints:

$$\frac{P_1 \xrightarrow{a,g_1,d_1} \mathcal{W}_1 \quad P_2 \xrightarrow{a,g_2,d_2} \mathcal{W}_2 \qquad (a \in B \cap \mathsf{IAct})}{P_1 \,||_B\, P_2 \xrightarrow{a,g_1\wedge g_2,d_1\vee d_2} (\mathcal{W}_1 \times \mathcal{W}_2) \circ \mathbf{M}_{\mathsf{par}}^{-1}}$$

The difference between synchronization of patient and impatience actions is only given by the way the urgency constraints are related, while the guard of the resulting transition is the conjunction of the guards of its constituents. In both cases, $(\mathcal{W}_1 \times \mathcal{W}_2)(\alpha_1, \alpha_2) \overset{\text{def}}{=} \mathcal{W}_1(\alpha_1) \cdot \mathcal{W}_2(\alpha_2)$, for all α_1 and α_2—

corresponding to the product of two probability spaces—and

$$\mathbf{M}_{\mathsf{par}}(\langle A_1, P_1' \rangle, \langle A_2, P_2' \rangle) \overset{\text{def}}{=} \begin{cases} \textbf{if } A_1 \cup A_2 \text{ is not a function } \textbf{then} \\ \quad \langle \varnothing, \mathsf{throw}(\mathit{inconsistent}) \rangle \\ \textbf{else} \quad \langle A_1 \cup A_2, P_1' \mid\mid_B P_2' \rangle \end{cases}$$

where, as before, $\surd \mid\mid_B \surd = \surd$. Function $\mathbf{M}_{\mathsf{par}}$ determines the continuation after the synchronization. Note that during synchronization 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 the predefined exception *inconsistent* and not performing any assignment.

5 Concrete Semantics

The semantics of a timed automaton can be given as an infinite-state labeled transition system in which transitions are either labeled with actions or with delays (i.e., real numbers). In a similar way, the semantics of a stochastic timed automaton is defined using timed probabilistic transition systems. These transition systems are infinite-state and are a slight generalization of timed transition systems as the target of a transition is not simply a state but a probability distribution over states. This semantics is defined in this section.

Timed probabilistic systems. We start by recapitulating some standard measure theory [55]. 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 $(\Omega, \mathcal{F})$. We only consider Borel measurable spaces and denote by $\mathcal{B}(\Omega)$ the Borel σ-algebra on sample space Ω. Let $\mathit{Prob}(\Omega)$ denote the probability measure on the Borel measurable space $(\Omega, \mathcal{B}(\Omega))$.

Definition 2.
A *probabilistic transition system* (*PTS*, for short) is a triple $(\Sigma, \mathcal{L}, \longrightarrow)$ where Σ is a set of *states*, $\mathcal{L}$ is a set of *labels*, and $\longrightarrow \subseteq \Sigma \times \mathcal{L} \times \mathit{Prob}(\Sigma)$ is the *(probabilistic) transition relation.* □

We write $\sigma \xrightarrow{\ell} \mathbf{P}$ whenever $\langle \sigma, \ell, \mathbf{P} \rangle \in \longrightarrow$. A probabilistic transition $\sigma \xrightarrow{\ell} \mathbf{P}$ is said to be *trivial* if its probability measure $\mathbf{P}$ is deterministic, i.e., a measure such that $\mathbf{P}(\{\sigma'\}) = 1$ for a given $\sigma' \in \Sigma$. In this case we write $\sigma \xrightarrow{\ell} \sigma'$.

In a timed PTS, transitions are either labeled by an action (as before) or with a real number indicating the amount of elapsed time. The latter transitions, also called timed transitions, have a single target state with probability 1.

Definition 3.
A *timed probabilistic transition system* is a PTS $(\Sigma, \mathcal{L}, \longrightarrow)$ such that:

- $\mathcal{L}$ is the disjoint union of a set Act of actions and the set $\mathbb{R}_{>0}$ of delays
- every transition labeled with $t \in \mathbb{R}_{>0}$ is trivial and satisfies [60]:
  - · time additivity: $\sigma \xrightarrow{t+t'} \sigma' \iff \sigma \xrightarrow{t} \sigma'' \xrightarrow{t'} \sigma'$ for some σ'', and
  - · time determinism: $\sigma \xrightarrow{t} \sigma'$ and $\sigma \xrightarrow{t} \sigma''$ imply $\sigma' = \sigma''$.

□

When defining the interpretation of stochastic timed automata, states in a timed PTS consists of a location indicating the state of control, and a valuation indicating the current values of all variables. Valuations are defined as follows.

Valuations. A *valuation* is a function that, to each variable in Var, assigns a value of its type. Let *Val* be the set of all valuations ranged over by v, v', v_1 and so forth. Let $F[v] \stackrel{\text{def}}{=} \lambda\xi.\, v(F(\xi))$ denote the instantiation of the sampling expression F with valuation v. $F[v]$ is a distribution function on variable ξ. Valuations are extended to expressions in the usual way: $v(e)$, for expression $e \in \mathsf{Exp}$, is obtained by replacing each variable x in e by $v(x)$ and by replacing each sample expression $\mathsf{sample}(F)$ by a unique random variable (name)[5] X with distribution $F[v]$. Uniqueness means that each occurrence of $\mathsf{sample}(F)$ in expression e is replaced by a distinct random variable, and, hence, sampled with possibly different values.

Example 3.
Let $e \equiv (x * \mathsf{sample}(\textsc{exp}(z))) + (\mathsf{sample}(\textsc{exp}(z)) * \mathsf{sample}(\textsc{exp}(y)))$, where EXP abbreviates EXPONENTIAL. Then $v(e) = (v(x) * X) + (Y * Z)$, where X, Y and Z are different random variable (names) with distributions $\textsc{exp}(v(z))$, $\textsc{exp}(v(z))$, and $\textsc{exp}(v(y))$, respectively. An example of sampling is to assign 3 to X, 5.5555... to Y, and $\sqrt{2}$ to Z. □

Valuation v is extended to assignment A by $v \circ A$ where it is required that random variables are unique among the assigned expressions. That is, if random variable X occurs in $(v \circ A)(x)$ and $x \neq y$ then it must not occur in $(v \circ A)(y)$. Let $\mathsf{RVar}(v \circ A)$ be the set of random variables appearing in $v \circ A$. Formally:

$$\mathsf{RVar}(v \circ A) = \{\, X \mid x \in \mathsf{Var} \text{ and } X \text{ occurs in } (v \circ A)(x) \,\}.$$

Note that $\mathsf{RVar}(v \circ A)$ is finite. Let $\mathsf{RVar}(v \circ A) = \{\, X_1, \ldots, X_n \,\}$ and let F_i be the probability distribution of random variable X_i (for $0 < i \leqslant n$). Let

[5] A random variable *is* a function. The term “random variable name” is used to distinguish between the symbol and the function. In the remainder we will not dwell upon this distinction anymore.

$\mathcal{B}(\mathbb{R}^n)$ be the Borel algebra on the n-dimensional real space, and $\mathbf{P}^v_A$ be the unique probability measure on $\mathcal{B}(\mathbb{R}^n)$ induced by $F_1, \ldots, F_n$ in the respective positions. As there is a trivial bijection between functions $\mathsf{RVar}(v \circ A) \to \mathbb{R}$ and $\mathbb{R}^n$, we identify u with the element $(u(X_1), \ldots, u(X_n)) \in \mathbb{R}^n$.

Interpretation of a stochastic timed automaton. A *state* in the behaviour of a STA is completely identified by the location in which the system is located and the value of all its variables. Let $\Sigma_{\mathsf{Loc}} \overset{\text{def}}{=} \mathsf{Loc} \times \mathit{Val}$ be the set of states and $\mathcal{B}(\Sigma_{\mathsf{Loc}})$ be the Borel algebra with sample space Σ_{Loc}.

Weight expression $\mathcal{W}$ is a *proper* weight expression in valuation v if $\neg(\ (\exists \alpha : v(\mathcal{W}(\alpha)) < 0) \vee ((\sum_\alpha v(\mathcal{W}(\alpha))) = 0)\ )$ holds, i.e., $\mathcal{W}(\alpha)$ does not take a negative value in v for any α in the domain of $\mathcal{W}$, and $\sum_\alpha \mathcal{W}(\alpha)$ does not evaluate to 0 in v. If $\mathcal{W}$ is proper in v, $\pi^v_{\mathcal{W}}$ denotes the discrete distribution function derived from the weight expression evaluated in v, i.e., $\pi^v_{\mathcal{W}}(\alpha) \overset{\text{def}}{=} \frac{v(\mathcal{W}(\alpha))}{\sum_\alpha v(\mathcal{W}(\alpha))}$ for all α. If it is not proper, $\pi^v_{\mathcal{W}}$ is not a (discrete) distribution and, hence, $\mathbf{P}^v_{\mathcal{W}}$ is not a probability measure.

Definition 4.
The semantics of stochastic timed automaton $(\mathsf{Loc}, \mathsf{Act}, \twoheadrightarrow)$ is the timed PTS $(\Sigma_{\mathsf{Loc}}, \mathsf{Act} \cup \mathbb{R}_{>0}, \longrightarrow)$ where $\longrightarrow$ is the smallest relation satisfying the following inference rules:

$$\textit{(1)}\quad \frac{s \xrightarrow{a,g,d} \mathcal{W} \qquad v(g) \text{ holds} \qquad \mathcal{W} \text{ is proper in } v}{\langle s, v\rangle \xrightarrow{a} \mathbf{P}^v_{\mathcal{W}}}$$

where

$$\mathbf{P}^v_{\mathcal{W}}(B) \overset{\text{def}}{=} \sum_{s \in \mathsf{Loc}, A \in \mathsf{Asgn}} \pi^v_{\mathcal{W}}(\langle A, s\rangle) \cdot (\mathbf{P}^v_A \circ (F^v_{\langle A,s\rangle})^{-1})(B)$$

and $F^v_{\langle A,s\rangle} : (\mathsf{RVar}(v \circ A) \to \mathbb{R}) \to \Sigma_{\mathsf{Loc}}$ is defined by $F^v_{\langle A,s\rangle}(u) \overset{\text{def}}{=} \langle s, (u \circ v \circ A)\rangle$. [6]

For the timed transitions we have:

$$\textit{(2)}\quad \frac{\forall t' < t : (v + t')(\mathsf{tp}_s) \text{ holds}}{\langle s, v\rangle \xrightarrow{t} \langle s, v + t\rangle}$$

[6] Note that $F^v_{\langle A,s\rangle}$ must be a measurable function. This strictly depends on A. Recall that $A : \mathsf{Var} \to \mathsf{Exp}$, then $A(x)$ is an expression that contains parameterized sample expressions and hence $v(A(x))$, when defined, defines a probability measure on the product space obtained from all random variables appearing in $v(A(x))$. In other words, $\lambda x.v(A(x))$ should be a random variable on the domain of x. Precisely, $F^v_{\langle A,s\rangle}$ is a measurable function whenever, for all $x \in \mathsf{Var}$, $v(A(x))$ is defined. For example, if $A(x) = \mathsf{sample}(\textsc{exp}(1/y))$, then $F^v_{\langle A,s\rangle}$ will not be defined if $v(y) = 0$.

where $\mathsf{tp}_s \overset{\text{def}}{=} \neg \bigvee\{d \mid s \xrightarrow{a,g,d} \mathcal{W}\}$ is the *time progress condition*, and $(v{+}t)(x) \overset{\text{def}}{=} v(x){+}t$ if $x \in \mathsf{Ck}$ and $v(x)$ otherwise. $\square$

Like for the semantics of timed automata, there are two inference rules that determine the transition relations: one that corresponds to taking an edge in the stochastic timed automaton, and one that controls the advance of time.

Inference rule (1) defines the execution of a control transition $s \xrightarrow{a,g,d} \mathcal{W}$. It requires that the guard g holds in valuation v (enabledness) and $\mathcal{W}$ is proper. As $\mathcal{W}$ is proper, $\mathbf{P}^v_{\mathcal{W}}$ defines a well-defined random selection of a location and the new valuation. This can be seen as a three-step process: (i) sample the target location s' together with the assignments A according to distribution $\pi^v_{\mathcal{W}}$, (ii) sample function u from random variables in $\mathsf{RVar}(v \circ A)$—this is done by $\mathbf{P}^v_A$—and (iii) determine the new state $\langle s', (u \circ v \circ A)\rangle$—which is done by function $F^v_{\langle A,s'\rangle}$. $u \circ v \circ A$ is the new variable valuation.

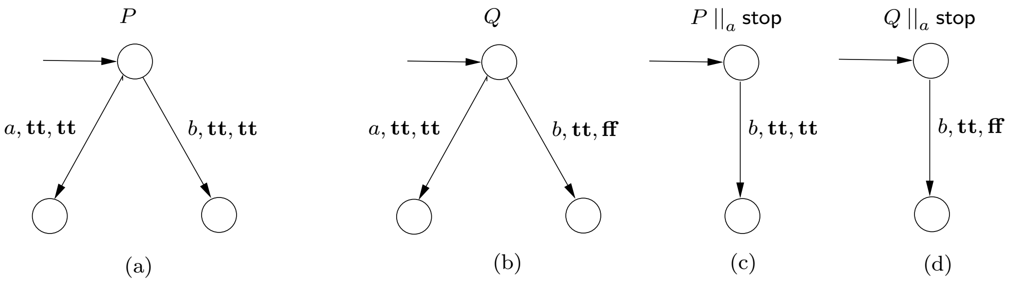
Note that the semantics of the palt-construct guarantees that for every MODEST term P and every valuation v, if $P \xrightarrow{a,g,d} \mathcal{W}$ and $v(g)$ holds, $\mathcal{W}$ is indeed proper in v.

Inference rule (2) controls the passage of time. It states that idling for t time units in state $\langle s, v\rangle$ is allowed as long as no urgency constraint is violated within this period. When t time units have elapsed in valuation v, the value of every clock $x \in \mathsf{Ck}$ is increased by t units, while the value of other variables remains unchanged.

Applying the inference rules of Section 4 to a MODEST specification yields a stochastic timed automaton. Subsequently, Definition 4 yields the timed probabilistic transition system that corresponds to the MODEST specification.

Bisimulation. When studying the behaviour of systems it is important to be able to check whether two systems behave in the same manner. For instance, this is useful to determine whether the model of a system implementation conforms to its specification. This is typically done with equivalence relations such as bisimulation [52]. Another reason is that whenever two systems show equivalent behaviour, one could be replaced by the other as part of a larger system. This requires the equivalence relation to be a congruence for the operators of the modeling language at hand.

For discrete-time probabilistic systems, Larsen-Skou's probabilistic bisimulation [47] is a well-accepted and investigated equivalence. This notion can be lifted to the continuous-time setting, as shown in, for instance, [14,22,26,28,16]. Unfortunately, this form of bisimulation is not a congruence for parallel com-



P
$a, \mathbf{tt}, \mathbf{tt}$
$b, \mathbf{tt}, \mathbf{tt}$
(a)
Q
$a, \mathbf{tt}, \mathbf{tt}$
$b, \mathbf{tt}, \mathbf{ff}$
(b)
$P \,||_a$ stop
$b, \mathbf{tt}, \mathbf{tt}$
(c)
$Q \,||_a$ stop
$b, \mathbf{tt}, \mathbf{ff}$
(d)

Fig. 2. Bisimulation is not a congruence in general

position [7]. Consider, for instance, the processes

$$\begin{aligned} P \equiv\ & \mathsf{alt}\{\ ::\mathsf{when}(\mathbf{tt})\ \mathsf{urgent}(\mathbf{tt})\ a \\ & \quad ::\mathsf{when}(\mathbf{tt})\ \mathsf{urgent}(\underline{\mathbf{tt}})\ b\ \}, \qquad \text{and} \\ Q \equiv\ & \mathsf{alt}\{\ ::\mathsf{when}(\mathbf{tt})\ \mathsf{urgent}(\mathbf{tt})\ a \\ & \quad ::\mathsf{when}(\mathbf{tt})\ \mathsf{urgent}(\underline{\mathbf{ff}})\ b\ \} \end{aligned}$$

whose automata are depicted in Figure 2(a) and (b), respectively. For convenience, the (slight) difference between the processes P and Q is underlined. As both P and Q have an outgoing edge with a valid urgency condition, both processes can only perform either the action a or b immediately. Hence, they are (probabilistic) bisimilar. Their composition with the context $\ldots \,||_{\{a\}}$ stop, however, yields non-bisimilar automata. This is depicted in Figure 2(c) and (d), respectively. Whereas the process $Q \,||_{\{a\}}$ stop is allowed to idle arbitrarily before performing action b, $P \,||_{\{a\}}$ stop is obliged to perform b immediately.

It is worthwhile to stipulate that there are interesting subsets of Modest for which bisimulation is indeed a congruence. A simple syntactic criterion, for instance, is to forbid the use of the urgent-construct and instead only allow the invariant-construct that will be introduced in Section 6. In this case, safety timed automata [35] are obtained. Alternatively, the concrete semantics can be adapted slightly such that it is possible to distinguish processes that are not subject to any interaction with their context anymore (so-called closed systems), and processes that do (i.e., open systems). This approach is worked out for a subset of stochastic timed automata in [22,26] and can be generalized in a rather straightforward way to the setting of this paper.

[7] This has already been observed for the case of timed automata with deadlines [12] and stochastic automata [22,26] which are submodels of STA.

6 Invariants and Some Shorthands

This section introduces some shorthand notations for modeling convenience, and elaborates on specifying location invariants — as opposed to urgency constraints — in MODEST.

Some shorthand notations. The following shorthands are considered. Both the alt- and do-construct allow an else alternative as in Promela [39]. else is a shorthand defined as follows:

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

In a probabilistic alternative, either assignments or processes (but not both) can be omitted, e.g.,

$$act\ \mathsf{palt}\ \{:1\!:\ \{\!=\ y = 3\ =\!\}\ :2\!:\ PN(4)\ \}$$

should be interpreted as

$$act\ \mathsf{palt}\ \{:1\!:\ \{\!=\ y = 3\ =\!\}\ \surd\ :2\!:\ \{\!=\quad =\!\}\ PN(4)\ \}$$

Note however that, strictly speaking, the last process is not a legal MODEST expression since $\surd$ is not a language construct (but only a semantic one). Other useful standard programming constructs, such as while-loops can be defined as usual:

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

As it is well known, hiding is a particular form of relabeling in which actions are renamed by the silent action τ:

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

Location invariants. Although we use urgency constraints for imposing urgency, location invariants like in safety timed automata [35] are more common. Location invariant b on process (i.e., location) P specifies that P can perform an initial activity as long as b holds. Once b becomes false, however, P is stuck and cannot perform any initial activity anymore (and forbids time to advance). This construct can be defined in MODEST as follows:

$$\begin{aligned}\mathsf{invariant}(b)\ P \overset{\text{def}}{=} \mathsf{alt}\{ &:: \mathsf{when}(b)\ P\\ &:: \mathsf{urgent}(\neg b)\ \mathsf{when}(\mathbf{ff})\ \mathsf{throw}(\mathit{invariant})\\ \}&\end{aligned}$$

where *invariant* is an exception that is not used in the rest of the Modest specification.

$\mathsf{invariant}(b)\ P$ behaves like P but due to the alternative with urgency constraint $\neg b$, it disallows the progress of time beyond the validity of b. Note that the alternative in which the exception *invariant* is raised is never executed as the guard does not hold. Note also that it is indeed necessary to use an alt construct in order to define invariants. In fact, the naive solution $\mathsf{invariant}'(b)\ P \overset{\text{def}}{=} \mathsf{urgent}(\neg b)\ P$ does not work in parallel compositions, as can be seen in the following example. Consider processes $P = \mathsf{invariant}'(x \leqslant 2)\ a; b$ and $Q = \mathsf{invariant}'(x \leqslant 5)\ c; a$, where a, b, c are actions. The expected invariant of process $R = \mathsf{par}\{::\ P :: Q\ \}$ is then $x \leqslant 2 \wedge x \leqslant 5$ (therefore R can only idle while $x \leqslant 2$). However, this is not the case. According to Modest operational semantics, the only transition from R is $R \xrightarrow{c,tt,\neg(x\leqslant 5)}$ since a is a common action and hence both P and Q must synchronise on it. As a consequence, R would be allowed to idle while $x \leqslant 5$ i.e. beyond the intended invariant $x \leqslant 2$.

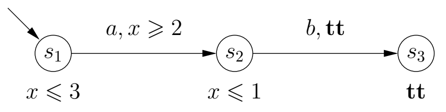
A second issue is the guarding of P with the invariant condition b by the alternative $::\ \mathsf{when}(b)\ P$ in the above invariant encoding. The reason for this is that urgency constraints only have effect on edges and not on locations as it is the case for invariants in timed automata. If on entering a location an urgency constraint is false, it only limits the execution of its respective transition (apart from time progress), but not the execution of any other transition whose guard is valid. On the contrary, false invariants in timed automata indicate impossible situations and hence no execution is further allowed. To illustrate the necessity of the guarding, consider the timed automaton in Figure 3.(a) where the boolean formulas below locations indicate invariants. The Modest process $T()$ defined in Figure 3.(b) represents this timed automaton, and its semantics is given by the STA in Figure 3.(c). Notice that action b in the second edge cannot be executed (in any of the two automata). However, if the edge from s_2 to s_3 in the STA were not guarded with the invariant $(x \leqslant 1)$, the b-transition could be executed as soon as location s_2 is reached (see rule *(1)* in Definition 4).

The definition of $\mathsf{invariant}()$ provides the expected compositional behaviour. Let predicate

$$\mathbf{inv}(P) \overset{\text{def}}{=} \bigwedge\{\neg d \mid P \xrightarrow{\mathit{invariant},g,d}\}.$$

Here, we assume $\bigwedge \varnothing = \mathbf{tt}$. Intuitively speaking, predicate $\mathbf{inv}(P)$ is the *time invariant* of process P. It follows:

(a)



(b)

```
process T() {
  clock x;
  invariant(x ⩽ 3) when(x ⩾ 2) a;
  invariant(x ⩽ 1) b
}
```

(c)

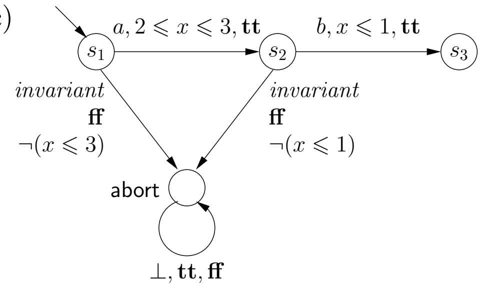


Fig. 3. A timed automaton, its MODEST description and corresponding STA

Proposition 1 $\mathbf{inv}(P)$ *can be recursively defined as in Table 3.*

Proof. We only show the case for P of the form $\mathsf{par}\{::P_1 \ldots ::P_k\}$. The other cases follow in a similar manner. Let $\mathbf{inv}(P_i) = \bigwedge\{\neg d_i \mid P_i \xrightarrow{invariant,g_i,d_i}\}$ for $0 < i \leqslant k$. Since exceptions—in particular *invariant*—are not subject to synchronization it directly follows from the inference rules of parallel composition:

$$\mathbf{inv}(P) = \bigwedge \left(\{\neg d_1 \mid P_1 \xrightarrow{invariant,g_1,d_1}\} \cup \ldots \cup \{\neg d_k \mid P_k \xrightarrow{invariant,g_k,d_k}\}\right).$$

By simple logic calculation, we obtain $\mathbf{inv}(P) = \bigwedge_{i=1}^{k} \mathbf{inv}(P_i)$. □

Time advances in P as long as no urgency constraint becomes true, i.e., as long as predicate $\mathsf{tp}_P = \neg \bigvee\{\, d \mid P \xrightarrow{a,g,d}\}$ holds (cf. Definition 4). Clearly, $\mathsf{tp}_P = \bigwedge\{\, \neg d \mid P \xrightarrow{a,g,d}\}$, and hence, $\mathbf{inv}(P)$ is the part of tp_P that controls the time progress by only *invariant*-labeled transitions. If in a MODEST specification only the invariant-construct is used, it follows that $\mathsf{tp}_P = \mathbf{inv}(P)$. In this case, stochastic timed automata correspond to safety timed automata.

7 Design Rationales

After having introduced the language and its semantics, we are now in a position to provide a deeper discussion of the design decisions that led us to set up MODEST and the model of STA in precisely the way we decided to. This section is intended to allow readers to distinguish optional and mandatory choices in the language setup.

Probabilistic branching. The attentive reader has realized that in MODEST each occurrence of a palt construct must be guarded by an action. In particular, it is not possible to choose probabilistically among actions prior to changing

Table 3
The invariant function

$\mathbf{inv}(P) = \mathbf{tt}$	if P has one of the following forms:
	stop, abort, break, act, act palt $\{\ldots\}$, $\mathsf{throw}(excp)$,
$\mathbf{inv}(P) = \neg b \wedge \mathbf{inv}(Q)$	if P is of the form: $\mathsf{urgent}(b)$ Q
$\mathbf{inv}(P) = \mathbf{inv}(Q)$	if P has one of the following forms:
	$\mathsf{when}(b)$ Q, Q; Q', $\mathsf{relabel}$ $\{I\}$ by $\{G\}$ Q, extend $\{H\}$ Q,
	$\mathsf{try}\{Q\}$ catch $excp_1$ $\{P_1\}$ $\ldots$ catch $excp_k$ $\{P_k\}$,
	$ProcName(\ldots)$ provided $\mathsf{process}$ $ProcName(\ldots)\{Q\}$
$\mathbf{inv}(P) = \bigwedge_{i=1}^{k} \mathbf{inv}(P_i)$	if P has one of the following forms:
	$\mathsf{alt}\{::P_1 \ldots ::P_k\}$, $\mathsf{do}\{::P_1 \ldots ::P_k\}$, $\mathsf{par}\{::P_1 \ldots ::P_k\}$

state. This restriction avoids the typical problems of parallel composition of probabilistic processes (see for a discussion [24,58]), and allows for defining a sound and elegant composition of STA. It, therefore, is one of the pillars of our compositional semantics. The restriction originates from the work of Lynch and Segala [54], but is extended here by allowing for weighted expressions instead of probabilities.

Clocks and distribution sampling. As in timed automata [3,12], clocks play a prominent role in MODEST. For modeling soft real-time systems in particular, the distinction between the setting of clocks (i.e., sampling from a general probability distribution) and the completion of a random delay is essential to obtain so-called expansion laws as in process calculi [52]. This allows (in its simplest form) for the reduction of independent parallelism in terms of alternative and sequential composition, and is of crucial importance for process algebraic verification purposes. This concept originates from [22,26] and is also adopted (in a slightly different form) in stochastic process algebras that support general distributions such as [15].

Patience and impatience. MODEST distinguishes patient and impatient actions. This feature has been introduced in order to provide a language that encompasses compositional modelling means for hard as well as soft real-time systems. Impatient actions can only synchronise as long as none of their urgency constraints turn true. That is to say, once a urgency constraint of one of the participants becomes true, the synchronisation should happen. A real-life application scenario would be that a meeting of some managers must finish (via a synchronisation) by the time the first participant needs to leave. Patient actions instead may synchronise as long as at least one of the urgency constraints is still false. A typical example of such synchronisation in the manager context is that the meeting can only start (via a synchronisation), once all participants are present. It is important to realize that patience and impatience cannot be encoded into each other. An alternative to express impatience in a patient setting is to use the concept of *urgent channels* as they are provided, for instance, by the timed-automata model checker UPPAAL [5].

Invariants and urgency constraints. STA are based on timed automata with deadlines [12]. This is reflected syntactically by the urgent construct. However, if one restricts to only using the invariant construct (as introduced in Section 6), the more standard model of timed automata is retained with all its compositional properties [23,50,59]. The latter model is tailored towards hard-real time systems. (In this model patience and impatience coincide). Timed automata with deadlines [8], on the other hand, have originally been introduced for modeling soft-real time systems. However, in this model, compositionality is shallow, because—as discussed in Section 5—bisimulation is a congruence only for limited usages with synchronisation on patient actions.

Data model and assignments. The MODEST language and semantics has stayed rather abstract with respect to the way assignment functions are specified. This is a deliberate decision, because we do not intend to prescribe unnecessary details. One may opt for functional declarations, as in standard ML [34] or E-LOTOS [41] or for imperative programs, as in LOTOS-NT [57]. The notation used in our examples (and in the current version of the tool) is defining the assignment function A by a sequence of assignments of the form

$$\{\!=\ x = \mathbf{tt},\ y = 0,\ z = \textsc{Exponential}(1/\mathit{delay\_time})\ =\!\}$$

We foresee that plain C-code fragments may also be used in this context, which would enable to include more complex data manipulations in a single atomic block. As an artificial example, it allows us to write, for instance

```
{=
  x = true ;
  for (int i=1, i<3, i++) {
    x = !x;
  }
=}
```

In this context, the assignment expression $A(x)$ is to be understood as the fixpoint of the function (in lambda-notation) corresponding to this fragment (which is $\lambda x.\,\mathbf{tt}$ in this example). Such a code fragment may also give rise to a multi-dimensional assignment.

[8] As the term deadline is somewhat misleading, we use the term urgency constraint instead [56].

There is however the following generic condition that must be met by any assignment function A. For each variable x, the assignment $A(x)$ must be a *random variable* on the domain of x, whenever the expression $A(x)$ is instantiated with concrete parameters. If no sampling is used in $A(x)$, this requirement boils down to the obvious requirement that $A(x) \in \mathsf{dom}\, x$, and in particular the code computing $A(x)$ must be terminating. In the presence of sampling, this requirement asserts termination with probability 1, as in, for example:

```
{=
  x = true ;
  while (x==true) {
    x = BERNOULLI(0.5)
  }
=}
```

(where BERNOULLI(0.5) corresponds to an unbiased probabilistic choice between $\{\mathbf{tt}, \mathbf{ff}\}$). Here, the code may not terminate, but this occurs with probability 0. The assignment function described by this code is $\lambda x.\, X$, where X is a random variable on $\{\mathbf{tt}, \mathbf{ff}\}$ taking value $\mathbf{ff}$ with probability 1 and $\mathbf{tt}$ with probability 0. Ensuring termination of such code fragment is left to the user, and surely it is advised to abstain from specifying code fragments like the ones above. Other approaches, such as PROMELA [39] or PROBMELA [4], are even more relaxed, and allow termination with probability less than 1. (Since PROMELA does not model probabilistic steps, this means that atomic statements may or may not terminate.)

Synchronisation discipline and value passing. Synchronisation between MODEST processes is realized by shared actions, i.e., actions contained in the alphabet of multiple processes. This kind of multi-way synchronisation originates from CSP, and enjoys a revival in the FSP [44] (Finite State Processes) language. Alternative synchronisation mechanisms, like binary synchronisation (as in CCS and the π-calculus) could also have been adopted for MODEST, if desired. For future extensions of MODEST, a graphical composition operator in the style of [31] could be an interesting generalisation of the current multi-way synchronisation paradigm. Value passing in MODEST takes place by means of shared variables. This mechanism is also adopted, for instance, in the timed-automata model checker UPPAAL [5]. For the sake of simplicity, the scheme of LOTOS with notions such as value generation and value matching has not been adopted.

Exception handling and scoping. MODEST exception handling is inspired by the exception handling mechanism of Ada [49]. Exceptions in MODEST are declared globally, and if thrown they may (or may not) be caught by a catching exception handler at the same or at a higher level. If unhandled, the exception is visible to parallel components just like ordinary actions. An unhandled exception terminates (in an error state) the process that raised it. Concurrent processes proceed unaffected. A "local" unhandled exception thus does not yield a global system halt. Synchronisation on exceptions is not possible in MODEST.

MODEST actions are also declared globally, so local actions are not directly supported (while local variables are), but can implicitly be achieved via the hide-construct. Together with action synchronisation, local action scopes would enable an abstract modelling of information hiding and security issues, as in the π- and $S\pi$-calculus [53,1]. The restriction to global action scopes is a design choice that has been made for simplicity, and might be relaxed.

Priorities. For the sake of simplicity, MODEST does currently not include means to express priorities. The approach recently proposed in [14] shows a possible way of incorporating priorities.

8 Concluding Remarks

This paper has introduced the modeling formalism MODEST, a language to model real-time and stochastic concurrent systems. The formal semantics has been provided in two layers: an operational semantics maps MODEST terms onto a finite-state model whose interpretation is given in terms of infinite transition systems—in the same vein as for timed automata [3].

MODEST is quite expressive covering a wide range of timed, probabilistic, nondeterministic, and stochastic models. Table 4 lists a selection of prominent models and makes precise which semantic concepts (cf. Section 1) each of them shares with *STA*. Apart from action nondeterminism, each listed semantic concept can be detected syntactically, while parsing a MODEST specification. Table 4 thus provides sufficient criteria for identifying submodels syntactically on the level of MODEST.

Action nondeterminism is a principal feature of compositional formalisms, yet it implies that DTMCs, CTMCs 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 poor to be of much use. More liberal syntactic conditions for the absence of action nondeterminism can be adopted from [51]. Semantic conditions can be incorporated while constructing the automaton underlying a MODEST specification, by resorting to algorithms proposed in [18,21,37]. Alternatively, one can resolve action nondeterminism using ad-hoc schedulers as in [22,27,10].

This paper has focused on the theoretical underpinnings of MODEST. The lan-

Table 4
Submodels of stochastic timed automata. LTS stands for labeled transition systems [43], PTS for probabilistic transition systems [54], TA for timed automata [3], PTA for probabilistic timed automata [46], DTMC for discrete- and CTMC for continuous-time Markov chains [45], CTMDP for continuous-time Markov decision processes [30], GSMP for generalized semi-Markov processes [33], and SA for stochastic automata [22,26]. CTMCs and CTMDPs are obtained if only negative exponential random variables are used, and clocks only occur in a restricted form (indicated by R; guards are right-continuous and clocks can be uniquely mapped on the random variables they use).

	LTS	PTS	TA	PTA	DTMC	CTMC	CTMDP	GSMP	SA	STA
probabilistic branching	-	+	-	+	+	+	+	+	+	+
clocks	-	-	+	+	-	R	R	+	+	+
random variables	-	-	-	-	-	EXP	EXP	+	+	+
delay nondeterminism	-	-	+	+	-	-	-	-	-	+
action nondeterminism	+	+	+	+	-	-	+	-	+	+

guage is supported by the Modest tool environment prototype Motor [9] [9] which has been linked to the stochastic analysis framework Möbius [20] [10]. This tool chain has recently been applied successfully to some industrial case studies originating from varying different industrial domains (plug-and-play networks [8], lacquer production plants [10], and a European standard for wireless train signaling [42]). These case studies have shown the effectiveness and adequacy of Modest. More importantly, though, these studies have confirmed that the formal underpinning of Modest—as laid down in this paper—is the basis of a trustworthy analysis. As convincingly illustrated in [17], the absence of such a rigorous basis easily leads to contradictory results for even simple models.

Acknowledgements

This work has been partially funded by the NWO Vernieuwingsimpuls award "Verification of performance and dependability", the ANPCyT project PICT

[9] Motor is available for download from `http://fmt.cs.utwente.nl/tools/motor/`.
[10] The Möbius software was developed by W.H. Sanders and the Performability Engineering Research Group (PERFORM) at the University of Illinois at Urbana-Champaign. See `http://www.mobius.uiuc.edu/`.

11-11738 “Teoría y Herramientas para la Construcción de Software Crítico”, the NWO/DFG-project “Validation of Stochastic Systems (VOSS)”, and the European Community Project IST-2001- 35304, AMETIST.

References

[1] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. and Comp.*, **148**(1):1–70, 1999.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, **138**(1): 3–34, 1995.

[3] R. Alur and D.L. Dill. A theory of timed automata. *Th. Comp. Sc.*, **126**(2):183–235, 1994.

[4] C. Baier, F. Ciezinski, and M. Groesser. PROBMELA: a modeling language for communicating probabilistic processes. In: *ACM-IEEE Int. Conf. on Formal Methods and Models for Codesign*, ACM Press, 2004.

[5] G. Behrmann, A. David, and K.G. Larsen. A tutorial on UPPAAL. In: *Formal Methods for the Design of Real-Time Systems*, LNCS 3185: 200–237. Springer-Verlag, 2004. (see also `www.uppaal.com`).

[6] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.

[7] G. Berry. Preemption and concurrency. In: *Found. of Software Techn. and Th. Comp. Sc.*, LNCS 761: 72–93. Springer-Verlag, 1993.

[8] Henrik Bohnenkamp, Johan Gorter, Jarno Guidi, and Joost-Pieter Katoen. Are you still there? — A lightweight algorithm to monitor node presence in self-configuring networks. 2005. Submitted to IPDS 2005.

[9] H. Bohnenkamp, H. Hermanns, J.-P. Katoen and J. Klaren. The MODEST modelling tool and its implementation. In: *Modelling Techniques and Tools for Computer Perfermance Evaluation*, LNCS 2794: 116–133. Springer-Verlag, 2003.

[10] H. Bohnenkamp, H. Hermanns, J. Klaren, A. Mader and Y.S. Usenko. Synthesis and stochastic assessment of schedules for lacquer production. In: *Quantitative Evaluation of Systems*, IEEE CS Press, pp. 28–38, 2004.

[11] T. Bolognesi and E. Brinksma. Introduction to the formal description technique LOTOS. *Computer Networks*, **14**: 25–59, 1987.

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

[13] V. Bos and J. J. T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. PhD thesis, Eindhoven University of Technology, 2002.

[14] M. Bravetti and P.R. D'Argenio. Tutte le algebre insieme: Concepts, discussions and relations of stochastic process algebras with general distributions. In *Validation of Stochastic Systems*, LNCS 2925: 44–88. Springer-Verlag, 2004.

[15] M. Bravetti and R. Gorrieri. The theory of interactive generalised semi-Markov processes. *Th. Comp. Sc.*, **286**(1): 5–32, 2002.

[16] S. Cattani, R. Segala, M.Z. Kwiatkowska and G. Norman. Stochastic transition systems for continuous state spaces and non-determinism. In V. Sassone, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, LNCS, Springer-Verlag, 2005 (to appear).

[17] D. Cavin, Y. Sasson, and A. Schiper. On the accuracy of MANET simulators. In: *Principles of Mobile Computing*, pp. 38–43. ACM Press, 2002.

[18] G. Ciardo and R. Zijal. Well-defined stochastic Petri nets. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 278–284. SCS Simulation Series, 1996.

[19] Special issue on embedded systems. *IEEE Computer*, **33**(9), 2000.

[20] D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derasavi, J. Doyle, W.H. Sanders and P. Webster. The MÖBIUS framework and its implementation. *IEEE Tr. on Softw. Eng.*, **28**(10):956–970, 2002.

[21] D.D. Deavours, and W.H. Sanders. An efficient well-specified check. In *Petri Nets and Performance Models*, IEEE CS Press, 1999.

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

[23] P.R. D'Argenio and E. Brinksma. A calculus for timed automata (extended abstract). In: *Formal Techniques in Real-Time and Fault Tolerant Systems*, LNCS 1135: 110-129. Springer-Verlag, 1996.

[24] P.R. D'Argenio, H. Hermanns and J.-P. Katoen. On generative parallel composition. *Electr. Notes on Th. Comp. Sc.*, **22**, 1999.

[25] P.R. D'Argenio, H. Hermanns, J.-P. Katoen and J. Klaren. MODEST: A modelling language for stochastic timed systems. In: *Process Algebra and Probabilistic Methods*, LNCS 2165: 87–104. Springer-Verlag, 2001.

[26] P.R. D'Argenio, J.-P. Katoen and E. Brinksma. An algebraic approach to the specification of stochastic systems (extended abstract). In: *Programming Concepts and Methods*, pp. 126–147, Chapman & Hall, 1998.

[27] P.R. D'Argenio, J.-P. Katoen and E. Brinksma. Specification and Analysis of Soft Real-Time Systems: Quantity and Quality. In: *Proc. of the 20th IEEE Real-Time Systems Symposium*, pp. 104–114. IEEE Society Press, 1999.

[28] Josée Desharnais. *Labeled Markov Process.* PhD thesis, McGill University, Montréal, 1999.

[29] S. Edwards, L. Lavagno, E.A. Lee and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation and synthesis. *Proceedings of the IEEE*, **85**(3):366–390, 1997.

[30] E.A. Feinberg and A. Shwartz. *Handbook of Markov Decision Processes.* Kluwer, 2002.

[31] H. Garavel and M. Sighireanu. A graphical parallel composition operator for process algebras. In *Formal Methods for Protocol Engineering and Distributed Systems*, pp. 185–202, Kluwer, 1999.

[32] H. Garavel and M. Sighireanu. On the introduction of exceptions in E-LOTOS. In: *Formal Description Techniques*, pp. 469–484. Kluwer, 1996.

[33] P.W. Glynn. A GSMP formalism for discrete event simulation. *Proc. of the IEEE*, **77**(1):14–23, 1989.

[34] M.R. Hansen and H. Rischel. *Introduction to Programming using SML.* Addison-Wesley, 1999.

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

[36] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274:43–87, 2002.

[37] H. Hermanns and D. Turetayev. A generalisation of the well-specified check. In: *Performability Modeling of Computer and Communication Systems*, pp. 62–66, 2003.

[38] Jane Hillston. *A Compositional Approach to Performance Modelling.* PhD thesis, University of Edinburgh, 1994.

[39] G.J. Holzmann. *The* Spin *Model Checker.* Addison-Wesley, 2002.

[40] C. Hoare *Communicating Sequential Processes.* Prentice-Hall, 1985.

[41] ISO/IEC. *Information Technology - E-LOTOS.* ISO/IEC International Standard 15437, 2001.

[42] D.N. Jansen, H. Hermanns and Y.S. Usenko. From StoCharts to MoDeST: a comparative reliability analysis of train radio communications. Submitted to WOSP 2005.

[43] R.M. Keller. Formal verification of parallel programs. *Communication of the ACM*, **19**(7): 371–384, 1976.

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

[45] V.G. Kulkarni. *Modeling and Analysis of Stochastic Systems.* Chapman & Hall, 1995.

[46] M.Z. Kwiatkowska, G. Norman, R. Segala and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, **282**: 101–150, 2002.

[47] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inf. and Comp.*, **94**(1): 1–28, 1991.

[48] E.A. Lee. Embedded software. In: M. Zelkowitz, editor, *Advances in Computers*, vol. **56**, Academic Press, 2002.

[49] D. Luckham and W. Polak. ADA exception handling: an axiomatic approach. *ACM Trans. on Prog. Lang. and Sys.*, **2**(2): 225–233, 1980.

[50] N. Lynch and F.W. Vaandrager. Action transducers and timed automata. *Formal Aspects of Comput.*, **8**(5): 499–538, 1996.

[51] V. Mertsiotakis. *Approximate Analysis Methods for Stochastic Process Algebras.* PhD thesis, University of Erlangen-Nürnberg, 1998.

[52] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[53] R. Milner. *Communicating and Mobile Systems: The π-Calculus*, Cambridge Univ. Press, 1999.

[54] R. Segala and N.A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic J. of Computing*, **2**(2): 250–273, 1995.

[55] A.N. Shiryaev. *Probability.* Graduate texts in Mathematics, vol. 95, 1996.

[56] J. Sifakis. Personal communication, 2004.

[57] M. Sighireanu. LOTOS NT user's manual (version 2.4). Tech. Rep. INRIA Rhône-Alpes/VASY, 2004 (`ftp://ftp.inrialpes.fr/pub/vasy/traian`).

[58] A. Sokolova and E.P. de Vink Probabilistic automata: system types, parallel composition and comparison In *Validation of Stochastic Systems*, LNCS 2925: 1–43. Springer-Verlag, 2004.

[59] W. Yi, P. Pettersson and M. Daniels. Automatic verification of real-time communicating systems by constraint solving. In: *Formal Description Techniques*, pp. 223–238, North-Holland, 1994.

[60] W. Yi. Real-time behaviour of asynchronous agents. In: *Concurrency Theory*, LNCS 458: 502–520, 1990.