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

Henrik Bohnenkamp, *Member*, *IEEE*, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen, *Member*, *IEEE Computer Society*

Abstract—This paper presents MODEST (MOdeling and DEscription language for Stochastic Timed systems), a formalism that is intended to support 1) the modular description of reactive systems' behavior while covering both 2) functional and 3) nonfunctional 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 nondeterministic 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.

Index Terms—Modeling formalism, compositionality, formal semantics, timed automata, stochastic processes.

1 INTRODUCTION

EMBEDDED software development must be supported from the design phase by formal methods to achieve strong results on correctness, performance, cost, and efficiency from the start. Modeling formalisms are needed that are easily accessible on the one hand and highly expressive on the other. In this paper, we present the modeling language MODEST (MOdeling and DEscription language for Stochastic Timed systems), which is a descendant of well-known process algebras, such as CSP and LOTOS, and which is capable of expressing not only functional, but real-time-related, stochastic, and probabilistic aspects of embedded software in a parsimonious way.

Background and motivation. Embedded software [30], [46] 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. Traditional software has a rather transformational nature, mapping input data onto output data. Embedded software is different in many respects. Most importantly, it is subject to complex and permanent interactions with its—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 may go on, nonfunctional 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. Classical abstractions of software that leave out "nonfunctional" aspects such as cost, efficiency, and robustness need to be adapted to current needs. This has resulted in various extensions of lightweight 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 [4], probabilistic automata [55] and hybrid automata [3]. Lightweight notations are typically closer to engineering techniques but lack a formal semantics; rigorous formalisms do have such formal semantics, but their learning curve is typically too steep from a practitioner's perspective and they mostly have a restricted expressiveness.

The description language MODEST that we propose in this paper is intended to have a rigorous formal basis (i.e., semantics) that incorporates several ingredients from lightweight notations, such as exception handling,[1]

- *H. Bohnenkamp and J.-P. Katoen are with the Software Modeling and Verification Group, Informatik 2, University (RWTH) Aachen, 52056 Aachen, Germany. E-mail: {henrik, katoen}@cs.rwth-aachen.de.*
- *P.R. D'Argenio is with the Computer Science Group, FaMAF, Universidad Nacional de Córdoba, Ciudad Universitaria, 5000—Cordoba, Argentina. E-mail: dargenio@famaf.unc.edu.ar.*
- *H. Hermanns is with the Dependable Systems and Software Group, Department of Computer Science, Saarland University, 66123 Saarbrucken, Germany. E-mail: hermanns@cs.uni-sb.de.*

Manuscript received 11 Feb. 2005; revised 16 Aug. 2006; accepted 21 Aug. 2006; published online 19 Oct. 2006.
Recommended for acceptance by A. Wellings.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0036-0205.

1. Exception handling in formal specification languages has received scant attention. Notable exceptions are e.g., Enhanced-LOTOS [33] and Esterel [8].

modularization, atomic assignments, iteration, and simple data types. MODEST is a descendant of classical process algebras like CSP and LOTOS and shares their compositional structure. Its semantics enables formal analysis and provides a solid basis for the development of tool support, whereas the lightweight ingredients are intended to pave the migration path toward engineers. The first industrial case studies [9], [11] with our tool environment [10] confirm that the rigid approach toward 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 MODEST 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 [43]), 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 to 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 lower and upper bounds on their occurrence time.
    5. Random variables are used to quantify the likelihood of an event happening after or within a certain time interval.

While items 1 and 2 affect the dynamics of a model via the (discrete) set of next events, items 4 and 5 are means to affect the model dynamics by the (continuous) elapse of time. Thus, items 1 and 4 describe two distinct types of nondeterminism, while items 2 and 5 represent distinct types of probabilistic behavior. 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, tongue-in-cheek 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. The semantics of MODEST is presented in two steps: Section 4 presents the syntax of MODEST and its operational semantics that associates with each MODEST process a stochastic timed automaton, and Section 5 presents the formal interpretation of stochastic timed automata in terms of probabilistic transition systems. Section 6 shows how some well-known constructs (like location invariants of timed automata and while-loops) can be expressed in MODEST. Section 7 discusses the motivations for the design decisions that have been made while developing MODEST. Section 8 concludes the paper. This paper is an extended and revised version of [26].

2 A Gentle Language Primer

This section introduces the core language features of MODEST by modeling an abstract view of a soccer match. The purpose of this example is to illustrate the main language *concepts* of MODEST and give the reader a feeling for the language. This is the reason to model a well-known, although nontrivial situation, rather than introduce and describe an involved technical subject.

Soccer 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 with the lowest score at the end of the match or that has suddenly no players left on the field *loses* the match. In the following, the potential evolution of a soccer match is described using MODEST. The description heavily uses its compositional features. We distinguish the teams by the numbers 0 and 1.

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 in Fig. 1a. 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 initial number of players.

One of the main activities of players during a match is fouling other players. To describe this behavior, the process $FoulPlay$ is introduced in Fig. 1b. This process describes that a team tries to foul players of the other team, but the

```
int score[2];              process FoulPlay(){
int players[2];              clock c;
players[0] = 11;             float delay;
players[1] = 11;             {= delay = UNIFORM(2,5), c = 0 =};
                             when(c == delay)
                               urgent(c == delay)
                                 throw foul
                           }

        (a)                        (b)
```

Fig. 1. (a) Variable declarations and (b) process *FoulPlay*.

```
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
} } }
```

Fig. 2. Process *Pass*.

time between fouls 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 $\{= \ldots =\}$ are executed atomically. The conditional constructs when$(\cdot)$ and urgent$(\cdot)$ are used to control the sojourn time of a process in a state. 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, at the latest, the process following the construct is *required* to be executed. Since Boolean expressions may refer to clocks, the evaluation of the expressions might change over time. In the process *FoulPlay*, the expressions in the when$(\cdot)$ and urgent$(\cdot)$ constructs 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 in which it was thrown. Exception handling is discussed below.

The ball, once possessed by the team with number *team*, is kicked away, either toward another player or into the goal. This is described by the process *Pass* in Fig. 2. *Pass* takes one parameter, the integer *team*, indicating which team is currently playing the ball. The ball is kicked away, as indicated by the action *kick*, and is either passed to another player (of the own or the opposing team) or goes into a goal (of the own or the opposing team). Note that, due to the absence of when and urgent constructs, the action *kick* is not restricted in any way; however, it is also not *required* to happen. The interpretation is that it is unspecified when the action happens, if it happens at all. The four possible outcomes of the *kick* action are described by means of the palt construct, which describes a *probabilistic choice* between alternatives. The branching probabilities are implicitly determined by so-called *weights*, which are arithmetic expressions (enclosed by colons) that evaluate to nonnegative values. The probability of a branch being chosen 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 $0.9 * players[team]/p_{all}$, the ball is passed to a player of the own team (indicated by action *self*), with probability $0.9 * players[1 - team]/p_{all}$ to a player of the opposing team (action *other*). With probability $0.1 * players[team]/p_{all}$, the ball goes into the goal of the opposing team, and with probability $0.1 * players[1 - team]/p_{all}$, in the own goal. The probabilities of where the ball eventually ends up vary according to the number of players on the field. In particular, the larger the difference between the number of players of the two teams gets, the smaller the probability for the smaller team to keep the ball in possession and to score goals.

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 the case of process *Pass*, the do indicates that the probabilistic choice should be repeated indefinitely. This infinite behavior is aborted when a break construct is executed. Execution continues then 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.

A team can only pass if it possesses the ball. This is described by the process *Play* in Fig. 3a: Whenever action *gotBall* is executed, process *Pass* is invoked. Subsequently, process *Play* is invoked recursively. In addition to the do construct, MODEST, therefore, also allow (tail-)recursion to specify infinite behavior of a process. Whether to choose recursion or the do construct is up to the user.

The behavior of a complete team can be described now as the *parallel composition* of two processes, as done in process *Team* (Fig. 3b): the process *Play* describing the handling of the ball, and the process *FoulPlay* describing the fouling of the other team. Using parallel composition to describe the behavior of a team is justified, since usually only one player can be in possession of the ball, whereas the others can still foul each other.

The two teams on the soccer field behave basically the same. However, slight differences in behavior, caused in essence by the fact that both teams play *against* each other, make it necessary to specialize the process *Team* in two

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

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

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

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

Fig. 3. Process $Play$ and process $Team$.

different ways, resulting in processes $Team0$ and $Team1$. These two processes are defined in Figs. 3c and 3d.

The differences between $Team0$, $Team1$, and $Team$ are the following:

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 processes $Team0$ and $Team1$. The action τ is invisible to other (parallel) components and, thus, cannot be used for synchronization.
2. In both processes, actions are relabeled: In case of $Team0$, 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 $Match$ in Fig. 4a. 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 the cases 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 opposite direction.

The parallel composition forms the *try-block* of the enclosing try/catch construct, which handles 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 both teams waiting for their opponent to pass the ball, although neither of them possesses it.

The process $Match$ describes a soccer match already to a certain degree; however, two things have to be taken care of: First, process $Match$ describes a never-ending match. Second, it is possible for a team to have a negative number of players. Both aspects are unrealistic. To address these situations, a process *Referee* is introduced (Fig. 4b), monitoring the time that has passed so far. It also ensures that there is always a nonnegative number of players of both teams on the field. This is again done by means of exceptions: Exception *gameover* is raised when 90 minutes have passed, whereas exception *noplayers* is thrown if a team has lost all its players. In our example, the use of the urgent() construct guarantees that the match is ended as soon as one of these conditions hold. The first condition depends on time, i.e., the valuation of the clock variable x which changes over time. The second condition depends on data, i.e., the valuation of the array *players*. Note that, in the first case, even though the urgency condition allows $x \geq 90$, the exception is thrown *exactly* at time 90. This is because the earliest time where an urgency constraint becomes true is the latest time where the subsequent process (in this case, throw *gameover*) can, and therefore must, be executed. Thus, in this case, the constraint $x \geq 90$ is equivalent to $x == 90$.

Finally, the complete specification of the soccer match, as given in Fig. 4c, 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*. If no players are left, the game simply stops. In the case that the

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

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

```
Main loop:

try{ par{
       :: Match()
       :: Referee()
     }
}
catch(gameover)
  { swap_shirts }
catch(noplayers)
  { τ }
```
(c)

Fig. 4. Processes $Match$, $Referee$, and the main loop.

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 [4], [13], a well-studied and tool-supported symbolic model for real-time systems, and stochastic automata [21], [27]. Timed automata extend labeled transition systems with 1) clocks to measure time elapsed, 2) guards (that possibly refer to clocks) to specify when an action is enabled, and 3) 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 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 and assignments. We distinguish the following syntactic categories:

1. Var is the set of (typed) *variables* ranged over by x, y, and z. It is sometimes convenient to distinguish the subset $\mathrm{Ck} \subseteq Var$ of *clock variables*, i.e., the variables of type clock that are used to measure the elapsing of time.
2. Exp is the set of *expressions* containing variables (in Var). It is ranged over by e. We distinguish the following subcategories of expressions:
   - $\mathrm{Sxp} \subseteq \mathrm{Exp}$ is the set of *sampling expressions* of the form $\mathrm{sample}(F)$, with the intended meaning that it samples a value for a distinguished (random) variable $\xi \notin \mathrm{Var}$ 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. For example, if x is a variable and
   $$F_x(\xi) = \begin{cases} 0 & \text{if } \xi < x, \\ (\xi - x)/3 & \text{if } \xi \in [x, x+3], \\ 1 & \text{if } \xi > x+3, \end{cases}$$
   then $\mathrm{sample}(F_x)$ samples a random value uniformly distributed in the interval $[x, x+3]$.
   - $\mathrm{Bxp} \subseteq \mathrm{Exp}$ is the set of *Boolean expressions*, ranged over by b, d, and g. These expressions do not contain sampling expressions. For example, expression $players[0]{==}0 \vee players[1]{==}0$ is a Boolean expression.
   - $\mathrm{Axp} \subseteq \mathrm{Exp}$ is the set of *arithmetic expressions*, ranged over by w. These expressions do not contain sampling expressions. An example of an arithmetic expression is $(team + 1)\%2$.
3. Asgn is the set of *assignments* ranged over by A. An assignment is a (type respecting) partial function that maps variables onto expressions (in Exp). Throughout the paper, we use the simultaneous assignment notation $\{= x_1 = e_1 \ldots, x_n = e_n =\}$, where all x_i are different. Formally, it denotes the unique assignment $A \in \mathrm{Asgn}$ defined by $A(x_i) = e_i$ (for $0 < i \leq n$), and undefined otherwise. Other (more involved) ways to represent assignments are discussed in Section 7. For instance, the assignment that appears in the soccer example, $\{= score[team]{+}= 1,\ players[team]{-}= 1 =\}$ (in particular, $score[team]{+}= 1$ is a shorthand for $score[team] = score[team] + 1$).
4. 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 evaluations of expressions may be assigned to them. Sample expressions are used to draw samples from distributions and are used to model random delays. For modeling convenience, some standard probability distribution functions such as $\mathrm{EXP}(rate)$ and $\mathrm{NORMAL}(avr, dev\_stndr)$ are supported. For instance, $\mathrm{UNIFORM}(x, x+3)$ is a shorthand for $\mathrm{sample}(F_x)$ with F_x as above. Boolean expressions are used in guards and urgency constraints. Actions play the same role as in labeled transition systems.

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 are labeled by four attributes:

1. an action a to be performed,
2. a guard g specifying when the edge is enabled,
3. an urgency constraint d specifying when the edge ultimately must be executed (if at all), and
4. 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 modeling locks and semaphores (see, e.g., [7, p. 43]). Notice also that no special condition is imposed on deadlines (as opposed to timed automata with deadlines [13], where the deadline d is required to imply guard g) in case a time-lock occurs if d holds but no guard (in particular g) leaving control state s is true.[2]

In order to deal with probabilistic branching, the situation is somewhat more complicated. The target of an edge is not just a location anymore, but rather a probability distribution over locations or, more precisely, a probability

2. A *time-lock* or *time-deadlock* is a situation in which time cannot progress and no action can be executed either. This means that the lifetime of the modeled system has reached its end. Time-lock is normally an error in the (model of the) system, and is usually catastrophic in safety critical systems.

distribution over pairs $\langle A, s\rangle$ of assignments and locations. 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 distribution over pairs of assignments and (target) control states. On taking the edge $s \xrightarrow{a,g,d} \mathcal{W}$, where $\mathcal{W}$ is a weight expression, 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 (see Section 5 for the formal definition). In the following, Wxp denotes the set of all weight expressions (on pairs of assignments and control states) and let $\mathcal{W}$, $\mathcal{W}'$, $\mathcal{W}_1, \ldots$ range over Wxp. 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 $v(\mathcal{W}(A, s)) \geq 0$ for all A and s, and $v(\mathcal{W}(A, s)) > 0$ for some A and s. Notice the difference between "weight expressions" and "weights." The former is a function $\mathcal{W}$ that, when applied to a pair, (A, s) returns a weight w (i.e., an arithmetic expression) such that $w = \mathcal{W}(A, s)$.

Since a weight expression is a symbolic form of a measure, we extend weight expressions to range over sets and products in the same way measures would do. We lift a weight expression $\mathcal{W}$ to any set $B \in \mathrm{Asgn} \times \mathrm{Loc}$ of pairs of assignments and locations by $\mathcal{W}(B) = \sum_{(A,s)\in B} \mathcal{W}(A, s)$. The product of two weight expressions $\mathcal{W}_1$ and $\mathcal{W}_2$ is defined by

$$(\mathcal{W}_1 \times \mathcal{W}_2)(\langle A_1, s_1\rangle, \langle A_2, s_2\rangle) = \mathcal{W}_1(A_1, s_1) \cdot \mathcal{W}_2(A_2, s_2),$$

for all assignments A_1 and A_2 and locations s_1 and s_2. We lift $(\mathcal{W}_1 \times \mathcal{W}_2)$ to sets of pairs, just like we did for weight expressions.

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

Example 1. Fig. 5 depicts a stochastic timed automaton representing the behavior of process $Play(0)$ of the soccer example. It contains seven locations. The automaton has a distinguished initial location indicated by an incoming arrow without source. Empty assignments, true guards, and false 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 arc connecting the weighted alternatives of assignments and locations. For the time being, ignore assignments to variable t_{Play} and assume it has value 0 all along the figure. Later we explain the precise STA for $Play(0)$ that is obtained by means of the formal semantics given in Section 4.2 (see Example 2); then, the use of t_{Play} will be apparent.

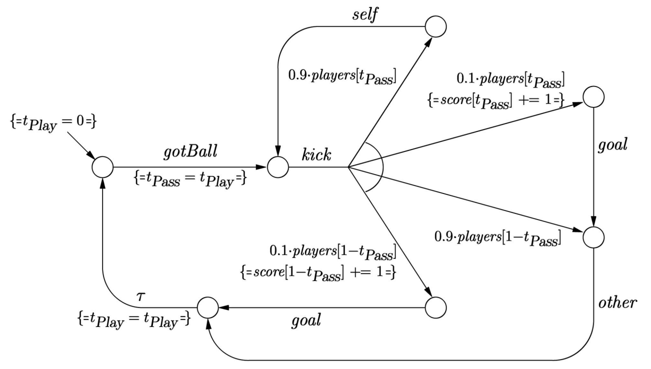


Fig. 5. Stochastic timed automaton representing the behavior of $Play(0)$.

It is worthwhile to emphasize that *STA* provides a *symbolic* framework to represent stochastic timed (and real-time) behavior in much the same way as timed automata represent real-time behavior 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 would 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

In the following, we define the language MODEST. We first introduce the formal syntax (Section 4.1) and briefly discuss the behavior of several language constructs, in particular, those not so commonly seen. The semantics of the language is given in two steps. We first give a structural operational semantics that maps a MODEST term onto an STA (Section 4.2). This STA gives a *symbolic* semantics, in the sense that actual states and actual probabilistic transitions are abstracted with variables, assignments, and expressions perhaps containing sample expressions. The concrete semantics of that MODEST term is then the interpretation of the derived STA in terms of *probabilistic transition systems*, which will be considered in Section 5.

When designing MODEST, we made many decisions in choosing the form of MODEST operations. We mostly omit discussion of these decisions in this section and postpone it until Section 7.

4.1 Syntax

This section 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. A patient action is an action that, when it intends to synchronize, will wait for its synchronizing partner, disregarding its urgency requirements until synchronization is possible. An impatient action, on the contrary, is not willing to wait for its synchronizing partner and, if its urgency condition becomes true, it will not let time progress. This may cause a time-lock situation. 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 break out of a loop, and τ is the unobservable action that is standard in most process calculi to model internal computations.

We distinguish processes and process behaviors. A process is defined by

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

where $x_i \in \mathrm{Var}$ $(0 < i \leq k)$ are different variables, each t_i is the type of x_i, dcl is a sequence of declarations possibly including process definitions, $ProcName$ is a process name, and P is a process behavior. For convenience, we will not dwell upon the syntax of declarations and write process $ProcName(x_1, \ldots, x_k)\{P\}$ instead in the following.

Process behaviors are defined as follows: Let $w_i \in \mathrm{Axp}$, $e_i \in \mathrm{Exp}$ (for $0 < i \leq k$), $b \in \mathrm{Bxp}$, and $asgn_i$ be an assignment of the form $\{= x_1 = e_1, \ldots, x_n = e_n =\}$. Furthermore, let $act \in \mathrm{PAct} \cup \mathrm{IAct} \cup \{\tau\}$ be an action as in standard process calculi (i.e., neither an exception, nor $\flat$, nor the unhandled error $\bot$), $H \subseteq \mathrm{PAct} \cup \mathrm{IAct}$ be a set of observable actions, and $excp$, $excp_i \in \mathrm{Excp}$ be exception names (for $0 < i \leq k$). Finally, let I and G be vectors of equal length which have elements in $\mathrm{Act} \backslash \{\flat, \bot\}$ such that all elements in I are pairwise different and not equal to τ. The intention is that the mapping $I(j) \mapsto G(j)$, for $0 \leq j < \#I$, defines a *relabeling* function.

A process behavior P is constructed according to the grammar given in Fig. 6.

$$\boxed{\begin{array}{rl} P ::= & \mathsf{stop} \mid \mathsf{abort} \mid \mathsf{break} \mid act \mid \mathsf{when}(b)\ P \mid \\ & \mathsf{urgent}(b)\ P \mid P_1;\ P_2 \mid \mathsf{alt}\{::P_1\ \ldots\ ::P_k\} \mid \\ & \mathsf{do}\{::P_1\ \ldots\ ::P_k\} \mid \mathsf{par}\{::P_1\ \ldots\ ::P_k\} \mid \\ & act\ \mathsf{palt}\ \{:w_1:asgn_1;\ P_1\ \ldots\ :w_k:asgn_k;\ P_k\} \mid \\ & ProcName(e_1, \ldots, e_k) \mid \mathsf{throw}(excp) \mid \\ & \mathsf{try}\{P\}\ \mathsf{catch}\ excp_1\ \{P_1\} \ldots \mathsf{catch}\ excp_k\ \{P_k\} \mid \\ & \mathsf{relabel}\ \{I\}\ \mathsf{by}\ \{G\}\ P \mid \mathsf{extend}\ \{H\}\ P \end{array}}$$

Fig. 6. Grammar of MODEST.

4.2 Operational Semantics

The operational semantics of behavior P is defined in terms of the stochastic timed automaton (Loc, Act, $\rightarrow$), 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 $\rightarrow$, and $\rightarrow$ is the smallest relation that is defined by the inference rules defined in the remainder of this section. Inference rules are given following the structured operational semantics style (SOS) in which the semantic of an operation is defined in terms of the semantics of its operands (see, e.g., [53], [54]).

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

Basic processes. Behavior 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 $\bot$. No assignments are executed. Its inference rule reads

$$\mathrm{abort} \xrightarrow{\bot,\mathrm{tt},\mathrm{ff}} \mathcal{D}(\oslash, \mathrm{abort}).$$

Action $\bot$ is always enabled, as the guard is true, and is not forced to occur at any time, as the urgency constraint is false. We recall that assignments are partial functions and, hence, here, $\emptyset$ means the empty assignment (no variable changes its value).

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

$$\mathrm{break} \xrightarrow{\flat,\mathrm{tt},\mathrm{ff}} \mathcal{D}(\oslash, \surd).$$

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

$$act \xrightarrow{act,\mathrm{tt},\mathrm{ff}} \mathcal{D}(\oslash, \surd).$$

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

Conditions. $\mathrm{when}(b)\ P$ 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:

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

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

$\mathrm{urgent}(b)\ P$ enforces the first activity of P to be urgent whenever b holds. It imposes 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

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

Process instantiation. Let

$$\text{process } \mathit{ProcName}(x_1, \ldots, x_k)\{P\}$$

be a process that is part of the current specification. For each instance of process $\mathit{ProcName}$, we assume variable names x_1 through x_k are unique in the whole system. That is, a second (different) instance of process $\mathit{ProcName}$ will be assumed to be defined by

$$\text{process } \mathit{ProcName}(x'_1, \ldots, x'_k)\{P'\},$$

where $x'_1, \ldots, x'_k$ are new variables (and, hence, different from $x_1, \ldots, x_k$) and P' is just like P, where all $x_1, \ldots, x_k$ were respectively replaced by $x'_1, \ldots, x'_k$.[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 all such necessary assignments (see Table 1). This function $\mathbf{A}$ is not used in the inference rule of process instantiation but is necessary for edges that may lead to a process call; see the inference rules for palt, exception handling, and sequential composition further on in this section. We point out that $\mathbf{A}(P)$ is a well-defined assignment; that is, it is indeed a partial function. Notice that unions in Table 1 are guaranteed to be *disjoint* unions because of the uniqueness of variable names. To make this point clear, consider process

$$Q = \mathrm{alt}\{:: \boldsymbol{PN}(e_1) :: \boldsymbol{PN}(e_2)\},$$

where process $\boldsymbol{PN}(x)\{P\}$. Then,

$$\mathbf{A}(Q) = \{y = e_1\} \cup \mathbf{A}(P[x/y]) \cup \{z = e_2\} \cup \mathbf{A}(P[x/z]),$$

provided y and z do neither appear in P nor in the context of process Q, and $P[x/y]$ and $P[x/z]$ are the same P, where

TABLE 1
The Assignment Collecting Function

$\mathbf{A}(P) = \varnothing$	if P has one of the following forms: act palt $\{:w_1:a_1;\ P_1\ \ldots\ :w_k:a_k;\ P_k\}$, act, stop, abort, throw$(excp)$, break
$\mathbf{A}(P) = \mathbf{A}(Q)$	if P has one of the following forms: $Q;\ Q'$, when$(b)\ Q$, urgent$(b)\ Q$, try$\{Q\}$ catch $e_1\ \{P_1\}\ \ldots$ catch $e_k\ \{P_k\}$, relabel $\{I\}$ by $\{G\}\ Q$, extend $\{H\}\ Q$
$\mathbf{A}(P) = \bigcup_{i=1}^{k} \mathbf{A}(P_i)$	if P has one of the following forms: alt$\{::P_1\ \ldots\ ::P_k\}$, do$\{::P_1\ \ldots\ ::P_k\}$, par$\{::P_1\ \ldots\ ::P_k\}$
$\mathbf{A}(\mathit{ProcName}(e_1, \ldots, e_k)) = \{x_1 = e_1, \ldots, x_k = e_k\} \cup \mathbf{A}(Q)$	if process $\mathit{ProcName}(x_1, \ldots, x_k)\{Q\}$

x is changed by y and z, respectively. Then, the well-definedness of $\mathbf{A}$ is guaranteed inductively.

The inference rule for process instantiation is

$$\frac{P \xrightarrow{a,g,d} \mathcal{W}}{\mathit{ProcName}(e_1, \ldots, e_k) \xrightarrow{a,g,d} \mathcal{W}},$$

if process $\mathit{ProcName}(x_1, \ldots, x_k)\{P\}$.

Choice. alt is the usual alternative composition. In case several alternatives in $\mathrm{alt}\{:: P_1 \ldots :: P_k\}$ are enabled, one of these alternatives is chosen nondeterministically. In fact, choice is resolved as in CCS:

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

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

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

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. Notice that $A \cup \mathbf{A}(Q)$ is a well-defined assignment since names in the domain of $\mathbf{A}(Q)$ are ensured to be new fresh names. The inverse of $\mathbf{M}_;$ is used in $\mathcal{W} \circ \mathbf{M}_;^{-1}$ to retrieve the weight expression for the sequential composition from the weight expressions assigned by $\mathcal{W}$ to the first component of a sequential composition.

Loop. Behavior $\mathrm{do}\{:: P_1 \ldots :: P_k\}$ repeatedly chooses an alternative P_i in the same nondeterministic manner as alt. It terminates whenever one of the processes P_i executes a

3. This can always be established by means of renaming, which can be achieved statically if all recursions are *tail* recursions, or dynamically otherwise. We do not provide a treatment of renaming as techniques for consistent renaming can be found elsewhere (see, e.g., [2])

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.

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

Fig. 7. Nonprobabilistic inference rules for auxdo.

break action ($\flat$). The semantics of do is defined using the auxiliary operator auxdo which has two arguments: the actual behavior and the behavior that needs to be resumed on successful termination of the loop body behavior. We have

$$\mathrm{do}\{::P_1 \ldots ::P_k\} \overset{\mathrm{def}}{=} \mathrm{auxdo}\{\mathrm{alt}\{::P_1 \ldots\ ::P_k\}\}\{\mathrm{alt}\{::P_1 \ldots ::P_k\}\}.$$

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

In a nonprobabilistic setting, where transitions have behaviors—rather than (symbolic) probability distributions—as targets, the intuitive behavior above would be encoded by the three inference rules given in Fig. 7. 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 last rule states that, once the loop body terminates its execution successfully, it should be resumed from the beginning. Notice, then, that the second argument in $\mathrm{auxdo}\{\_\}\{\_\}$ is only used to save a copy of the original process to be reexecuted when the body of the loop ends.

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 last two rules from above are combined into 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

$$\frac{P \xrightarrow{\flat,g,d} \mathcal{W}}{\mathrm{auxdo}\{P\}\{Q\} \xrightarrow{\tau,g,d} \mathcal{D}(\oslash, \surd)},$$

$$\frac{P \xrightarrow{a,g,d} \mathcal{W} \qquad (a \neq \flat)}{\mathrm{auxdo}\{P\}\{Q\} \xrightarrow{a,g,d} \mathcal{W} \circ M_{\mathrm{do}}^{-1}},$$

where

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

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

As Q in $\mathrm{auxdo}\{P\}\{Q\}$ is a well-defined MODEST process, it cannot be a successfully terminating process (i.e., it must do an action before terminating); hence, the semantics of $\mathrm{auxdo}\{P\}\{Q\}$ (and so that of do) is well defined.

Relabeling. The semantics for relabeling is as in traditional process algebra. Let

$$Q \ \equiv \mathrm{relabel}\{a_1, \ldots, a_k\} \text{ by } \{a'_1, \ldots, a'_k\}\ P.$$

Q behaves like P except that every observable action or exception a_i is renamed to the corresponding a'_i:

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

where

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

Alphabet extension. Let $Q \equiv \mathrm{extend}\{a_1, \ldots, a_k\}\ P$. extend just extends the alphabet of process P (see Table 2) and might affect behavior only if it appears within the context of a par operator:

$$\frac{P \xrightarrow{a,g,d} \mathcal{W}}{\mathrm{extend}\ \{a_1, \ldots, a_{k0}\}\ P \xrightarrow{a,g,d} \mathcal{W} \circ \mathrm{M}_{\mathrm{ext}}^{-1}},$$

where

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

TABLE 2
Alphabet of a MODEST Term

$$\alpha(\mathsf{stop}) = \alpha(\mathsf{abort}) = \alpha(\mathsf{break}) = \alpha(\mathsf{throw}(excp)) = \varnothing$$
$$\alpha(act) = \{act\} - \{\tau\}$$
$$\alpha(act\ \mathsf{palt}\ \{:w_1{:}asgn_1;\ P_1\ \ldots\ :w_k{:}asgn_k;\ P_k\}) = \alpha(act) \cup \textstyle\bigcup_{i=1}^{k} \alpha(P_i)$$
$$\alpha(\mathsf{when}(b)\ P) = \alpha(\mathsf{urgent}(b)\ P) = \alpha(P)$$
$$\alpha(\mathsf{alt}\{::P_1\ \ldots\ ::P_k\}) = \alpha(\mathsf{do}\{::P_1\ \ldots\ ::P_k\}) = \alpha(\mathsf{par}\{::P_1\ \ldots\ ::P_k\}) = \bigcup_{i=1}^{k} \alpha(P_i)$$
$$\alpha(P_1;\ P_2) = \alpha(P_1) \cup \alpha(P_2)$$
$$\alpha(\mathsf{try}\{P\}\ \mathsf{catch}\ e_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ e_k\ \{P_k\}) = \alpha(P) \cup \textstyle\bigcup_{i=1}^{k} \alpha(P_i)$$
$$\alpha(\mathsf{hide}\{act_1, \ldots, act_k\}\ P) = \alpha(P) - \{act_1, \ldots, act_k\}$$
$$\alpha(\mathsf{relabel}\ \{a_1, \ldots, a_k\}\ \mathsf{by}\ \{a'_1, \ldots, a'_k\}\ P) = \alpha(P)[a_1/a'_1, \ldots, a_k/a'_k] - \{\tau\}$$
$$\alpha(\mathsf{extend}\ \{act_1, \ldots, act_k\}\ P) = \alpha(P) \cup \{act_1, \ldots, act_k\}$$
$$\alpha(ProcName(e_1, \ldots, e_k)) = \alpha(P) \quad \text{provided process } ProcName(x_1, \ldots, x_k)\ \{P\}$$

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

$$\mathrm{throw}(excp) \xrightarrow{excp,\mathbf{tt},\mathbf{ff}} \mathcal{D}(\oslash, \mathrm{abort}).$$

Let $Q \equiv \mathrm{try}\{P\}\ \mathrm{catch}\ excp_1\{P_1\} \ldots \mathrm{catch}\ excp_k\{P_k\}$. Q behaves like P as long as P does not raise an exception $excp_i$ $(0 < i \leq k)$. If P raises exception $excp_i$, it continues behaving 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):

$$\frac{P \xrightarrow{a,g,d} \mathcal{W} \qquad (a \notin \{excp_1, \ldots, excp_k\})}{Q \xrightarrow{a,g,d} \mathcal{W} \circ \mathbf{M}_{\mathrm{try}}^{-1}},$$

where

$$\mathbf{M}_{\mathrm{try}}(A, P') \stackrel{\mathrm{def}}{=} \begin{cases} \langle A, Q \rangle & \text{if } P' \neq \surd \\ \langle A, \surd \rangle & \text{if } P' = \surd. \end{cases}$$

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

$$\frac{P \xrightarrow{excp_i,g,d} \mathcal{W} \qquad (0 < i \leq k)}{Q \xrightarrow{\tau,g,d} \mathcal{D}(\mathbf{A}(P_i), P_i)}.$$

Note that, although raising the exception $excp_i$ results in an unhandled error (see the inference rule for throw), the resulting behavior of the entire expression is P_i, the handler of $excp_i$.

Probabilistic prefix. Let

$$Q \equiv act\ \mathrm{palt}\ \{:w_1 : asgn_1;\ P_1\ \ldots\ :w_k : asgn_k;\ P_k\}.$$

Q performs action act without restriction, randomly selects an alternative i according to the weights $w_1, \ldots, w_k$, 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 behavior. Therefore, w_i must be nonnegative and $w_1 + \cdots + w_k$ must be nonzero. Since weights may contain variables, these conditions are checked at "runtime," i.e., in the concrete semantics (see Section 5).

We define the predicates $neg \equiv \bigvee_{i=1}^{k} w_i < 0$ and

$$zero \equiv \sum_{i=1}^{k} w_i = 0.$$

The inference rule covering the normal situation is

$$Q \xrightarrow{act,\neg(neg \vee zero),\mathbf{ff}} \mathcal{W},$$

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

$$\mathcal{W}(asgn_i \cup \mathbf{A}(P_i), P_i) \stackrel{\mathrm{def}}{=} \textstyle\sum_{j=1}^{k} \mathbf{I}(i,j) \cdot w_j,$$

where $\mathbf{I}(i,j) \stackrel{\mathrm{def}}{=} 1$, if

$$asgn_i \cup \mathbf{A}(P_i) = asgn_j \cup \mathbf{A}(P_j) \wedge P_i = P_j,$$

and 0 otherwise. The guard $\neg(neg \vee zero)$ ensures that the weights are legal.

Note that, besides the assignments $asgn_i$, the (possible) assignments introduced by process instantiation in P_i are also performed. Each $asgn_i \cup \mathbf{A}(P_i)$ is a well-defined assignment, since the domain of $\mathbf{A}(P_i)$ contains only fresh names.

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:

$$Q \xrightarrow{neg\_weight,neg,\mathbf{ff}} \mathcal{D}(\oslash, \mathrm{abort}),\ \text{and}$$
$$Q \xrightarrow{neg\_weight,zero,\mathbf{ff}} \mathcal{D}(\oslash, \mathrm{abort}).$$

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

Example 2. To illustrate the STA semantics of the palt construct, we resume Example 1. Fig. 5 presents the STA derived for $Play(0)$. We have taken t_{Play} to be the *unique* variable that replaces variable $team$ in the definition of $Play(team)$, and similarly, t_{Pass} in the definition of $Pass(team)$. Next to the initial transition, we wrote the value of $\mathbf{A}(Play(0))$. Even when this is not formally part of the STA, it gives the idea that, whatever might be the incoming arrow, it should include the assignment $t_{Play} = 0$. The τ transition is obtained as a consequence of the breaking of the loop in process $Pass$. The assignment $t_{Play} = t_{Play}$ on this transition is a residual of the recursion of $Play$. Since this is a tail recursion, the same variable becomes fresh again and, hence, it can be reused.

Parallel Composition. Let $Q \equiv \mathrm{par}\{:: P_1 \ldots :: P_k\}$. In Q, $P_1, \ldots, P_k$ run concurrently, while synchronizing on their shared actions, thus allowing for multiway synchronization. The alphabet of a process P is the set $\alpha(P) \subseteq \mathrm{PAct} \cup \mathrm{IAct}$ of all actions P recognizes (see Table 2). To define the semantics of MODEST parallel composition, we resort to the auxiliary operator $\|_B$, with $B \subseteq \mathrm{PAct} \cup \mathrm{IAct}$, that behaves like CSP or LOTOS parallel composition [12], [40]. The operator par is defined by

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

with $B_j = (\bigcup_{i=1}^{j} \alpha(P_i)) \cap \alpha(P_{j+1})$. Note that

$$B_j \subseteq \mathrm{IAct} \cup \mathrm{PAct},$$

i.e., B_j contains only observable actions. The special actions $\bot$, $\flat$, τ, and exception names do not belong to it. The behavior 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:

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

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

with $\mathbf{M}_{\mathrm{par}P}(A, P') \overset{\mathrm{def}}{=} \langle A, P' \|_B P \rangle$, 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 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):

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

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:

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

The difference between synchronization of patient and impatient 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, $(\mathcal{W}_1 \times \mathcal{W}_2)(\alpha_1, \alpha_2) \overset{\mathrm{def}}{=} \mathcal{W}_1(\alpha_1) \cdot \mathcal{W}_2(\alpha_2)$, for all α_1 and α_2—corresponding to the product of two probability spaces—and

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

where, as before, $\surd \|_B \surd = \surd$. Function $\mathbf{M}_{\mathrm{par}}$ 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 generalization of timed transition systems, as the target of a transition is not simply a state but a probability distribution over states. In this section, we first introduce probabilistic transition systems (PTS) and its timed variant. Next, we define several notations, mostly related to the instantiation of the different structures that an STA with a particular valuation deals with (namely, expressions, sample expressions, and assignments). They form the basis for the construction of the probability distribution associated to the transitions in the semantics. Subsequently, we define the semantics of an STA in terms of timed PTSs. We end the section discussing bisimulation, which is a semantic relation intended to equate behavior.

Timed probabilistic systems. We start by recapitulating some standard measure theory [56]. A *probability space* is a tuple $(\Omega, \mathcal{F}, \mathbf{P})$, where Ω is the *sample space*, $\mathcal{F} \subseteq 2^{\Omega}$ is a *σ-algebra* on Ω (i.e., a set containing Ω and closed under complement and denumerable union), and $\mathbf{P}$ is a *probability measure* on $\mathcal{F}$ (i.e., a function $\mathbf{P} : \mathcal{F} \to [0,1]$ such that $\mathbf{P}(\Omega) = 1$ and $\mathbf{P}(\biguplus_{i \geq 0} B_i) = \sum_{i \geq 0} \mathbf{P}(B_i)$, where $\{B_i\}_{i \geq 0}$ is a disjoint denumerable family of sets in $\mathcal{F}$). The pair $(\Omega, \mathcal{F})$ is called *measurable space*.

We only consider *Borel measurable spaces*. A Borel measurable space is the smallest measurable space containing all open sets of a topology, which, in our case, are basically multidimensional spaces on the set $\mathbb{R}$ of real numbers. We denote by $\mathcal{B}(\Omega)$ the Borel σ-algebra on sample space Ω. Let $\mathit{Prob}(\Omega)$ denote the set of all probability measures on $\mathcal{B}(\Omega)$.

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

We write $\sigma \xrightarrow{\ell} \mathbf{P}$ whenever $\langle \sigma, \ell, \mathbf{P} \rangle \in \to$. 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 labeled either with 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 $(\Sigma, \mathcal{L}, \to)$ such that*

1. *$\mathcal{L}$ is the disjoint union of a set* Act *of actions and the set $\mathbb{R}_{>0}$ of delays, and*
2. *every transition labeled with $t \in \mathbb{R}_{>0}$ is trivial and satisfies [60]*
   - *time additivity: $\sigma \xrightarrow{t+t'} \sigma' \Longleftrightarrow \sigma \xrightarrow{t} \sigma'' \xrightarrow{t'} \sigma'$ 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, a state 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 $F[v] \overset{\text{def}}{=} \lambda\xi.v(F(\xi))$ denote the instantiation of the sampling expression F with valuation v. $F[v]$ is a distribution function on variable ξ. For example, consider function F_x in Section 3, which describes a uniform distribution in the interval $[x, x+3]$, for whichever value x may take. Then, if v is such that $v(x) = 5$, $F_x[v]$ is the uniform distribution in the interval $[5, 8]$. That is,

$$F_x[v](\xi) = \begin{cases} 0 & \text{if } \xi < 5, \\ (\xi - 5)/3 & \text{if } \xi \in [5, 8], \\ 1 & \text{if } \xi > 8. \end{cases}$$

Valuations are extended to expressions as follows: $v(e)$, for expression $e \in \text{Exp}$, is obtained by replacing each variable x in e by $v(x)$ and by replacing each sample expression $\text{sample}(F)$ by a unique random variable (name)[5] X with distribution $F[v]$. Uniqueness means that each occurrence of $\text{sample}(F)$ in expression e is replaced by a distinct random variable and, hence, sampled with possibly different values. Notice that $v(e)$ is an expression whose variables are random variable names. Moreover, notice that, due to uniqueness, every random variable occurs at most once in the expression $v(e)$.

Example 3. Let

$$e \equiv (x * \text{sample}(\text{EXP}_z)) + (\text{sample}(\text{EXP}_z) * \text{sample}(\text{EXP}_y)),$$

where EXP_λ is the function

$$\text{EXP}_\lambda(\xi) = \begin{cases} 0 & \text{if } t < 0 \\ 1 - e^{-(\xi\lambda)} & \text{if } t \geq 0; \end{cases}$$

that is, EXP_λ is the negative exponential distribution with rate λ. Suppose that v is a valuation such that $v(x) = 12$, $v(y) = 5$, and $v(z) = 18$. Then,

$$v(e) = (v(x) * X) + (Y * Z) = (12 * X) + (Y * Z),$$

where X, Y, and Z are different random variable (names) with distributions EXP_{18}, EXP_{18}, and EXP_5, respectively, (since $\text{EXP}_z[v] = \text{EXP}_{18}$ and $\text{EXP}_y[v] = \text{EXP}_5$).

We defined assignments to be partial functions, but, within this section, we will assume that they are total. A (partial) assignment A would therefore be interpreted to be the total function A' defined by $A'(x) = A(x)$ if A is defined in x, and $A'(x) = x$ otherwise.

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, $(v \circ A)$ is a function from Var to expressions on random variables such that, for all $x \in \text{Var}$, $(v \circ A)(x) = v(A(x))$ and, if random variable X occurs in $(v \circ A)(x)$ and $x \neq y$, then X must not occur in $(v \circ A)(y)$. Let $\text{RVar}(v \circ A)$ be the set of random variables appearing in $v \circ A$. Formally,

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

Note that $\text{RVar}(v \circ A)$ is finite. Let

$$\text{RVar}(v \circ A) = \{X_1, \ldots, X_n\}$$

and let F_i be the probability distribution of random variable X_i (for $0 < i \leq n$). Let $\mathcal{B}(\mathbb{R}^n)$ 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 $\text{RVar}(v \circ A) \to \mathbb{R}$ 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 behavior of an STA is completely identified by the location in which the system is located and the value of all its variables. Let $\Sigma_{\text{Loc}} \overset{\text{def}}{=} \text{Loc} \times Val$ be the set of states and $\mathcal{B}(\Sigma_{\text{Loc}})$ be the Borel algebra with sample space Σ_{Loc}.

Weight expression $\mathcal{W}$ is a *proper* weight expression in valuation v if

$$\neg\left(((\exists A, s : v(\mathcal{W}(\langle A, s\rangle)) < 0) \vee \left(\left(\sum_{\langle A,s\rangle} v(\mathcal{W}(\langle A, s\rangle))\right) = 0\right)\right)$$

holds, i.e., $\mathcal{W}(\langle A, s\rangle)$ does not take a negative value in v for any pair $\langle A, s\rangle$ in the domain of $\mathcal{W}$, and $\sum_{\langle A,s\rangle} \mathcal{W}(\langle A, s\rangle)$ 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.,

$$\pi^v_{\mathcal{W}}(\langle A, s\rangle) \overset{\text{def}}{=} \frac{v(\mathcal{W}(\langle A, s\rangle))}{\sum_{\langle A,s\rangle} v(\mathcal{W}(\langle A, s\rangle))},$$

for every pair $\langle A, s\rangle$. If it is not proper, $\pi^v_{\mathcal{W}}$ is not a (discrete) distribution and, hence, $\mathbf{P}^v_{\mathcal{W}}$ (in Definition 4) would not be a probability measure.

As 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.

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

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

where

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

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 of this paper, we will not dwell upon this distinction.

and

$$F^v_{\langle A,s\rangle} : (\mathrm{RVar}(v \circ A) \to \mathbb{R}) \to \Sigma_{\mathrm{Loc}}$$

is defined by $F^v_{\langle A,s\rangle}(u) \overset{\mathrm{def}}{=} \surd\langle s, (u \circ v \circ A)\rangle$.[6]

For the timed transitions, we have

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

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

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 (so that $\mathbf{P}^v_{\mathcal{W}}$ is well defined). If this is the case, action a can be performed and the next state is selected randomly according to probability measure $\mathbf{P}^v_{\mathcal{W}}$. That is, $\mathbf{P}^v_{\mathcal{W}}$ defines the probability with which new locations and new valuations are selected. This can be seen as a three-step process: 1) sample the target location s' together with the assignments A according to distribution $\pi^v_{\mathcal{W}}$, 2) sample function u (recall that u is also a vector in $\mathbb{R}^{\#\mathrm{RVar}(v \circ A)}$) from random variables in $\mathrm{RVar}(v \circ A)$—this is done by $\mathbf{P}^v_A$—and 3) determine the new state $\langle s', (u \circ v \circ A)\rangle$—which is done by function $F^v_{\langle A,s'\rangle}$. s' is the new location and $u \circ v \circ A$ is the new variable valuation.

We show that $\mathbf{P}^v_{\mathcal{W}}$ is a (well-defined) probability measure on $\mathcal{B}(\Sigma_{\mathrm{Loc}})$.[7] By definition, $\mathbf{P}^v_A$ is a probability measure on $\mathcal{B}(\mathbb{R}^{\#\mathrm{RVar}(v \circ A)})$. We assumed that $F^v_{\langle A,s\rangle} : \mathbb{R}^{\#\mathrm{RVar}(v \circ A)} \to \Sigma_{\mathrm{Loc}}$ is measurable (see footnote 6). This guarantees that $\mathbf{P}^v_A \circ (F^v_{\langle A,s\rangle})^{-1}$ is a probability measure on $\mathcal{B}(\Sigma_{\mathrm{Loc}})$. Moreover, since $\mathcal{W}$ is proper, $\pi^v_{\mathcal{W}}$ is a discrete probability distribution on $\mathrm{Loc} \times \mathrm{Asgn}$. Therefore, the linear combination $\mathbf{P}^v_{\mathcal{W}} = \sum_{s \in \mathrm{Loc}, A \in \mathrm{Asgn}} \pi^v_{\mathrm{W}}(\langle A, s\rangle) \cdot (\mathbf{P}^v_A \circ (F^v_{\langle A,s\rangle})^{-1})$ is a probability measure on $\mathcal{B}(\Sigma_{\mathrm{Loc}})$.

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 $x \in \mathrm{Ck}$ 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 behavior 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 [51]. Another reason is that whenever two systems show equivalent behavior, one can 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.

Bisimulation has been defined for PTSs with continuous probability [14], [16], [21], [29]. Nevertheless, it is not a congruence for MODEST operations. This is due to the use of deadlines. (This has already been observed for the case of timed automata with deadlines [13] and stochastic automata [21], [27] which are submodels of STA.) Recently, however, a variant of bisimulation (called ∇-bisimulation) has been shown to be a congruence for parallel composition and urgent constraints in the setting of timed automata with deadlines [23], [24]. Extending this relation to include continuous probability should be straightforward.

6 Useful Shorthands

This section elaborates on specifying location invariants and some other common notations by direct encoding in MODEST.

Location invariants. While we use urgency constraints for imposing urgency, location invariants, as 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

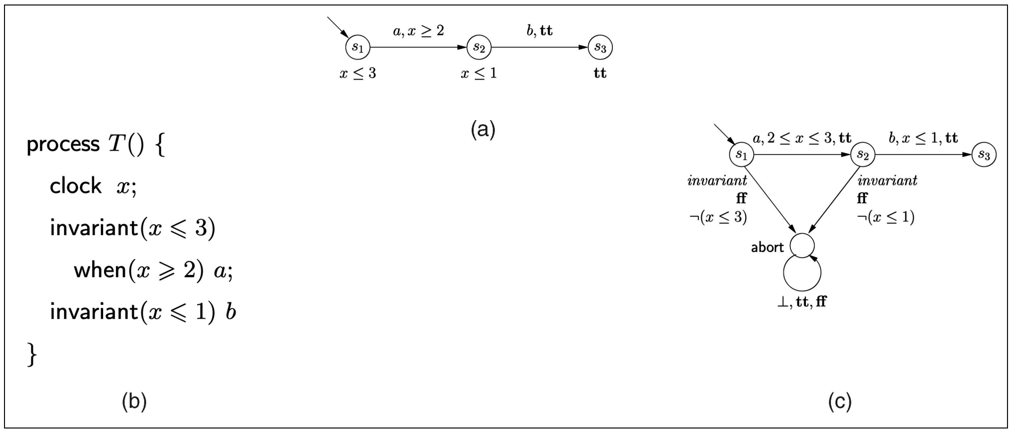
$$\begin{aligned}\mathrm{invariant}(b)\ P \overset{\mathrm{def}}{=} \mathrm{alt}\{\ & :: \mathrm{when}(b)\ P \\ & :: \mathrm{urgent}(\neg b)\ \mathrm{when}(\mathbf{ff}) \\ & \quad\ \mathrm{throw}(\mathit{invariant}) \\ \},\end{aligned}$$

where *invariant* is an exception that is not used in the rest of the MODEST specification. There is no behavioral difference if *invariant* is replaced by any other exception, even if it *is* used elsewhere in the specification. The preference for a fresh name is that it allows us to easily identify the invariants in the derived STA for further manipulation as, for example, to translate it to a timed automata that can be input in a model checker.

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

6. Note that $F^v_{\langle A,s\rangle}$ must be a measurable function. This strictly depends on A. Recall that $A : \mathrm{Var} \to \mathrm{Exp}$; 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 $x \in \mathrm{Var}$, $v(A(x))$ is defined. For example, if $A(x) = \mathrm{sample}(\mathrm{EXP}(1/y))$, then $F^v_{\langle A,s\rangle}$ will not be defined if $v(y) = 0$.

7. We will use here two well-known results in measure theory. The first one states that, if $(\Omega, \mathcal{F}, \mathbf{P})$ is a probability space, $(\Omega', \mathcal{F}')$ is a measurable space, and if $f : \Omega \to \Omega'$ is a measurable function, then $\mathbf{P} \circ f^{-1}$ is a probability measure on $\mathcal{F}'$. The second one states that if $\{\mathbf{P}_i\}_{i \in I}$ is a countable family of probability measures on $\mathcal{F}$ and π is a discrete probability distribution on I, then the linear combination $\sum_{i \in I} \pi(i) \cdot \mathbf{P}_i$ is also a probability measure on $\mathcal{F}$. See, e.g., [56] for further details.



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

Fig. 8. A timed automaton, its MODEST description, and corresponding STA.

use an alt construct in order to define invariants. In fact, the naive solution $\mathrm{invariant}'(b)P \overset{\mathrm{def}}{=} \mathrm{urgent}()\neg bP$ does not work in parallel compositions, as can be seen in the following example: Consider processes $P = \mathrm{invariant}'(x \leq 2)a; b$ and $Q = \mathrm{invariant}'(x \leq 5)c; a$, where a, b, and c are actions. The expected invariant of process $R = \mathrm{par}\{:: P :: Q\}$ is $x \leq 2 \wedge x \leq 5$ (therefore, R can only idle while $x \leq 2$). However, this is not the case. According to the MODEST operational semantics, the only transition from R is

$$R \xrightarrow{c, \mathbf{tt}, \neg(x \leq 5)},$$

since a is a common action and, hence, both P and Q must synchronize on it. As a consequence, R would be allowed to idle while $x \leq 5$, i.e., beyond the intended invariant $x \leq 2$.

A second issue is the guarding of P with the invariant condition b by the alternative $:: \mathrm{when}(b)P$ in the above invariant encoding. The reason for this is that urgency constraints only have effect on edges and not on locations, as is the case for invariants in safety 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. In safety timed automata, however, false invariants indicate impossible situations and, hence, no further execution is allowed. To illustrate the necessity of the guarding, consider the timed automaton in Fig. 8a, where the Boolean formulas below locations indicate invariants. The MODEST process $T()$ defined in Fig. 8b represents this timed automaton, and its semantics is given by the STA in Fig. 8c. 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 $(x \leq 1)$, the b-transition could be executed as soon as location s_2 is reached (see rule 1 in Definition 4).

The definition of invariant() provides the expected compositional behavior. Let predicate

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

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

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

Proof. We only show the case for P of the form $\mathrm{par}\{:: P_1 \ldots :: P_k\}$. 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 $0 < i \leq k$. Since exceptions—in particular *invariant* —are not subject to synchronization, it directly follows from the inference rules of parallel composition that

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

By simple logic calculation, we obtain

$$\mathbf{inv}(P) = \bigwedge_{i=1}^{k} \mathbf{inv}(P_i).$$

□

TABLE 3
The Invariant Function

$\mathbf{inv}(P) = \mathbf{tt}$	if P has one of the following forms: stop, abort, break, act, act palt $\{\ldots\}$, throw$(excp)$,
$\mathbf{inv}(P) = \neg b \wedge \mathbf{inv}(Q)$	if P is of the form: urgent(b) Q
$\mathbf{inv}(P) = \mathbf{inv}(Q)$	if P has one of the following forms: when(b) Q, $Q; Q'$, relabel $\{I\}$ by $\{G\}$ Q, extend $\{H\}$ Q, try$\{Q\}$ catch e_1 $\{P_1\}$ $\ldots$ catch e_k $\{P_k\}$, *ProcName*$(\ldots)$, provided process *ProcName*$(\ldots)\{Q\}$
$\mathbf{inv}(P) = \bigwedge_{i=1}^{k} \mathbf{inv}(P_i)$	if P has one of the following forms: alt$\{::P_1 \ldots ::P_k\}$, do$\{::P_1 \ldots ::P_k\}$, par$\{::P_1 \ldots ::P_k\}$.

Time advances in P as long as no urgency constraint becomes true, i.e., as long as predicate $\mathrm{tp}_P = \neg \bigvee\{d \mid P \xrightarrow{a,g,d}\}$ holds (see Definition 4). Clearly, $\mathrm{tp}_P = \bigwedge\{\neg d \mid P \xrightarrow{a,g,d}\}$, 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 $\mathrm{tp}_P = \mathbf{inv}(P)$. In this case, stochastic timed automata correspond to safety timed automata.

Further shorthand notations. The following shorthands are included in MODEST. Both the alt- and do-construct allow an else alternative as in Promela [39], derivable as follows:

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

In a probabilistic alternative, either assignments or processes (but not both) can be omitted. Concretely, this means that $act\ \mathrm{palt}\{:1:\{=y=3=\}:2:PN(4)\}$ should be interpreted as $act\ \mathrm{palt}\{:1:\{=y=3=\}\surd:2:\{==\}PN(4)\}$. Strictly speaking, however, the latter process is not a legal MODEST expression since $\surd$ is not a language construct (but only a semantic one). In a similar line, conventional assignements like $y=3;$ are to be read as $\{=y=3=\};$. Other useful standard programming constructs, such as while-loops can be defined as usual:

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

Hiding as in $\mathrm{hide}\{act_1, \ldots, act_k\}P$ is a particular form of relabeling in which act_1 to act_k are all mapped to silent τ.

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 construct must be guarded by an action. This choice avoids the typical problems of parallel composition of probabilistic processes (see [25], [58] for a discussion), and allows for defining a sound and elegant composition of STA. It is, therefore, is one of the pillars of our compositional semantics. The restriction originates from the work of Segala and Lynch [55] but is extended here by allowing for weighted expressions instead of probabilities.

For self-contained reasons, we briefly mention why this choice is superior to two commonly found alternative design choices (see [25], [28] for details):

1. to invert the order of our choice by first probabilistically selecting an action and then executing it (both steps performed in one atomic transition), or
2. to split (i.e., break the atomicity) between the performing of an action and the probabilistic choice.

The first of these choices involves unclear or too restrictive decisions when interacting in parallel composition or synchronization. For instance, a synchronization would require a (not always) desirable normalization (to "redistribute" the probability lost on missynchronization), and an interleaving may need fictitious probabilistic parameters to resolve a second-level nondeterminism that occurs "between" the probabilistic choice and the selection of the action (i.e., *within* a transition!). The second choice can already be represented in MODEST by explicitly breaking the atomicity and using a τ-labeled palt construction to represent the probabilistic selection. Besides, it is more restrictive. (Think of rolling two dice as two synchronizing actions: This choice of operations would not allow one to represent atomically the usual uniform distribution on the values of the pair of dice.)

Clocks and distribution sampling. As in timed automata [4], [13], 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 [51]. This allows (in its simplest form) for the reduction of independent parallelism to alternative and sequential composition and is of crucial importance for process algebraic verification purposes. This concept originates from [21], [27] 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 modeling means for hard as well as soft real-time systems. Impatient actions can only synchronize as long as none of their urgency constraints turn true. That is to say, once an urgency constraint of one of the participants becomes true, the synchronization should happen. A real-life application scenario would be that a meeting of some managers must finish (via a synchronization) by the time the first participant needs to leave. Patient actions instead may synchronize as long as at least one of the urgency constraints is still false. A typical example of such synchronization in the manager context is that the meeting can only start (via a synchronization), once all participants are present. It is important to realize that patience and impatience cannot be encoded into each other. An alternative way 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 [6].

Invariants and urgency constraints. STA are based on timed automata with deadlines [13]. This is reflected syntactically by the urgent construct. However, if one restricts oneself to using only the invariant construct (see Section 6), the more standard model of timed automata is retained with all its compositional properties [22], [48], [59]. The latter model is tailored toward hard-real time systems. (In this model, patience and impatience coincide). Timed automata with deadlines,[8] on the other hand, were originally 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 synchronization on patient actions.

Data model and assignments. The MODEST language and semantics have 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 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

$$\{= x = \mathbf{tt},\ y = 0,\ z = \text{EXPONENTIAL}(1/\mathit{delay}) =\}.$$

We foresee that plain C-code fragments may also be used in this context, which would enable one 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 multidimensional assignment.

There is, however, the following generic condition to 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 $A(x) \in \mathrm{dom}\, x$, 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 a 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 [5], 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.)

Synchronization discipline and value passing. Synchronization between MODEST processes is realized by shared actions, i.e., actions contained in the alphabet of multiple processes. This kind of multiway synchronization originates from CSP and enjoys a revival in the FSP [43] (Finite State Processes) language. Alternative synchronization mechanisms, like binary synchronization (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 [32] could be an interesting generalization of the current multiway synchronization 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 [6]. 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 Ada [47]. 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. An unhandled exception terminates the raising process in an error state. Concurrent processes proceed unaffected. A "local" unhandled exception thus does not yield a global system halt. Synchronization 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 synchronization, local action scopes would enable an abstract modeling of information hiding and security issues, as in the π- and $S\pi$-calculus [1], [52]. The restriction to global action scopes is a design choice that has been made for simplicity and might be relaxed.

Priorities. For simplicity, MODEST does not include means to express priorities. The approach 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—as for timed automata [4].

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 (see Section 1) each of them shares with STA. In the table, LTS stands for labeled transition systems, PTS for probabilistic transition systems [55], TA for timed automata [4], PTA for probabilistic timed

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

TABLE 4
Submodels of Stochastic Timed Automata

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

automata [45], DTMC for discrete-time and CTMC for continuous-time Markov chains [44], CTMDP for continuous-time Markov decision processes [31], GSMP for generalized semi-Markov processes [34], and SA for stochastic automata [21], [27]. 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).

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 [50]. Semantic conditions can be incorporated while constructing the automaton underlying a MODEST specification by resorting to algorithms proposed in [18], [20], [37]. Alternatively, one can resolve action nondeterminism using ad hoc schedulers as in [11], [21], [28].

This paper has focused on the theoretical underpinnings of MODEST. The language is supported by the MODEST tool environment prototype MOTOR[9] [10], which has been linked to the stochastic analysis framework MÖBIUS[10] [19]. This tool chain has recently been applied successfully to some industrial case studies originating from varying different domains:

- Stability analysis of plug-and-play networks [9]. This study led to a network protocol redesign that has lately been patented by a large Dutch electronics company.
- Schedulablity analysis of a lacquer production plant [11], [49]. Here, the MODEST language and tool has been used together with UPPAAL to assess the quality and robustness of production schedules in a faulty environment.
- Reliability estimation of wireless train signaling [42]. This case study focused on the upcoming European standard for train interoperability ETCS and estimated how wireless link failures of various types influence the spatial proximity of high speed trains.

Though these practical applications are out of the scope of this paper, it is worth mentioning that they have shown the effectiveness and adequacy of MODEST. More importantly, these case 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.

9. MOTOR is available for download from the following URL: 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/.

ACKNOWLEDGMENTS

This work was partially funded by the NWO Vernieuwingsimpuls award "Verification of Performance and Dependability," the ANPCyT project PICT 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, Advanced Methods for Timed Systems (AMETIST).

REFERENCES

[1] M. Abadi and A.D. Gordon, "A Calculus for Cryptographic Protocols: The SPI Calculus," *Information and Computing,* vol. 148, no. 1, pp. 1-70, 1999.

[2] L. Aceto, "A Static View of Localities," *Formal Aspects of Computing,* vol. 6, pp. 201-222, 1994.

[3] R. Alur et al., "The Algorithmic Analysis of Hybrid Systems," *Theoretical Computer Science,* vol. 138, no. 1, pp. 3-34, 1995.

[4] R. Alur and D.L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science,* vol. 126, no. 2, pp. 183-235, 1994.

[5] C. Baier, F. Ciesinski, and M. Groesser, "PROBMELA: A Modeling Language for Communicating Probabilistic Processes," *Proc. Int'l Conf. Formal Methods and Models for Codesign (MEMOCODE '04),* 2004.

[6] G. Behrmann, A. David, and K.G. Larsen, "A Tutorial on UPPAAL," *Proc. Int'l Conf. Formal Modelling and Analysis of Timed Systems (FORMATS '04),* 2004.

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

[8] G. Berry, "Preemption and Concurrency," *Foundations of Software Technology and Theoretical Computer Science,* pp. 72-93, 1993.

[9] H. Bohnenkamp, J. Gorter, J. Guidi, and J.-P. Katoen, "Are You Still There?—A Lightweight Algorithm to Monitor Node Presence in Self-Configuring Networks," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '05),* pp. 704-709, June 2005.

[10] H. Bohnenkamp, H. Hermanns, J.-P. Katoen, and J. Klaren, "The MODEST Modelling Tool and Its Implementation," *Proc. Conf. Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS '03),* pp. 116-133, 2003.

[11] H. Bohnenkamp, H. Hermanns, J. Klaren, A. Mader, and Y.S. Usenko, "Synthesis and Stochastic Assessment of Schedules for Lacquer Production," *Proc. Int'l Conf. Quantitative Evaluation of Systems (QEST '04),* 2004.

[12] T. Bolognesi and E. Brinksma, "Introduction to the Formal Description Technique LOTOS," *Computer Networks,* vol. 14, pp. 25-59, 1987.

[13] S. Bornot and J. Sifakis, "An Algebraic Framework for Urgency," *Information and Computation,* vol. 163, pp. 172-202, 2001.

[14] M. Bravetti and P.R. D'Argenio, "Tutte le Algebre Insieme: Concepts, Discussions and Relations of Stochastic Process Algebras with General Distributions," *Validation of Stochastic Systems,* LNCS 2925, pp. 44-88, Springer-Verlag, 2004.

[15] M. Bravetti and R. Gorrieri, "The Theory of Interactive Generalised Semi-Markov Processes," *Theoretical Computer Science,* vol. 286, no. 1, pp. 5-32, 2002.

[16] S. Cattani, R. Segala, M.Z. Kwiatkowska, and G. Norman, “Stochastic Transition Systems for Continuous State Spaces and Non-Determinism,” *Proc. Conf. Foundations of Software Science and Computation Structures (FOSSACS ’05),* pp. 125-139, 2005.

[17] D. Cavin, Y. Sasson, and A. Schiper, “On the Accuracy of MANET Simulators,” *Principles of Mobile Computing,* pp. 38-43, ACM Press, 2002.

[18] G. Ciardo and R. Zijal, “Well-Defined Stochastic Petri Nets,” *Modeling, Analysis and Simulation of Computer and Telecommunication Systems,* SCS Simulation Series, pp. 278-284, 1996.

[19] D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derasavi, J. Doyle, W.H. Sanders, and P. Webster, “The Mobius Framework and Its Implementation,” *IEEE Trans. Software Eng.* vol. 28, no. 10, pp. 956-970, Oct. 2002.

[20] D.D. Deavours and W.H. Sanders, “An Efficient Well-Specified Check,” *Proc. Int’l Workshop Petri Nets and Performance Models (PNPM ’99),* pp. 124-133, 1999.

[21] P.R. D’Argenio, “Algebras and Automata for Timed and Stochastic Systems,” PhD thesis, Dept. of Computer Science, Univ. of Twente, 1999.

[22] P.R. D’Argenio and E. Brinksma, “A Calculus for Timed Automata,” *Proc. Int’l Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT ’96),* pp. 110-129, 1996.

[23] P.R. D’Argenio and B. Gebremichael, “The Coarsest Congruence for Timed Automata with Deadlines Contained in Bisimulation,” *Proc. Int’l Conf. Concurrency Theory (CONCUR ’05),* pp. 125-140, 2005.

[24] P.R. D’Argenio and B. Gebremichael, Axiomatising Timed Automata with Deadlines, technical report, 2006, to appear.

[25] P.R. D’Argenio, H. Hermanns, and J.-P. Katoen, “On Generative Parallel Composition,” *Electronic Notes on Theoretical Computer Science,* vol. 22, 1999.

[26] P.R. D’Argenio, H. Hermanns, J.-P. Katoen, and J. Klaren, “Modest: A Modelling Language for Stochastic Timed Systems,” *Joint Int’l Workshop Process Algebra and Performance Modelling and Probabilistic Methods in Verification (PAPM-PROBMIV ’01),* pp. 87-104, 2001.

[27] P.R. D’Argenio, J.-P. Katoen, and E. Brinksma, “An Algebraic Approach to the Specification of Stochastic Systems,” *Programming Concepts and Methods,* pp. 126-147, Chapman & Hall, 1998.

[28] P.R. D’Argenio, J.-P. Katoen, and E. Brinksma, “Specification and Analysis of Soft Real-Time Systems: Quantity and Quality,” *Real-Time Systems Symp. (RTSS ’99),* pp. 104-114, 1999.

[29] J. Desharnais, “Labeled Markov Process,” PhD thesis, McGill Univ., Montréal, 1999.

[30] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli, “Design of Embedded Systems: Formal Models, Validation and Synthesis,” *Proc. IEEE,* vol. 85, no. 3, pp. 366-390, 1997.

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

[32] H. Garavel and M. Sighireanu, “A Graphical Parallel Composition Operator for Process Algebras,” *Proc. Conf. Formal Techniques for Networked and Distributed Systems (FORTE ’99),* pp. 185-202, 1999.

[33] H. Garavel and M. Sighireanu, “On the Introduction of Exceptions in E-LOTOS,” *Proc. Conf. Formal Techniques for Networked and Distributed Systems (FORTE ’96),* pp. 469-484, 1996.

[34] P.W. Glynn, “A GSMP Formalism for Discrete Event Simulation,” *Proc. IEEE,* vol. 77, no. 1, pp. 14-23, 1989.

[35] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, “Symbolic Model Checking for Real-Time Systems,” *Information and Computation,* vol. 111, pp. 193-244, 1994.

[36] H. Hermanns, U. Herzog, and J.-P. Katoen, “Process Algebra for Performance Evaluation,” *Theoretical Computer Science,* vol. 274, pp. 43-87, 2002.

[37] H. Hermanns and D. Turetayev, “A Generalisation of the Well-Specified Check,” *Proc. Int’l Workshop Performability Modeling of Computer and Comm. (PMCCS),* pp. 62-66, 2003.

[38] J. Hillston, “A Compositional Approach to Performance Modelling,” PhD thesis, Univ. 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 International Standard 15437, Information Technology—E-LOTOS,* Int’l Organization for Standardization, 2001.

[42] D.N. Jansen, H. Hermanns, and Y.S. Usenko, “From Stocharts to Modest: A Comparative Reliability Analysis of Train Radio Communications,” *Proc. Workshop Software and Performance (WOSP ’05),* pp. 13-23, 2005.

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

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

[45] M.Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston, “Automatic Verification of Real-Time Systems with Discrete Probability Distributions,” *Theoretical Computer Science,* vol. 282, pp. 101-150, 2002.

[46] E.A. Lee, “Embedded Software,” *Advances in Computers,* M. Zelkowitz, ed., vol. 56, Academic, 2002.

[47] D. Luckham and W. Polak, “ADA Exception Handling: An Axiomatic Approach,” *ACM Trans. Programming Languages and Systems,* vol. 2, no. 2, pp. 225-233, 1980.

[48] N. Lynch and F.W. Vaandrager, “Action Transducers and Timed Automata,” *Formal Aspects of Computing,* vol. 8, no. 5, pp. 499-538, 1996.

[49] A. Mader, H. Bohnenkamp, Y.S. Usenko, D.N. Jansen, J. Hurink, and H. Hermanns, “Synthesis and Stochastic Assessment of Cost-Optimal Schedules,” Technical Report 06-14, Univ. Twente, 2006.

[50] V. Mertsiotakis, “Approximate Analysis Methods for Stochastic Process Algebras,” PhD thesis, Univ. of Erlangen-Nürnberg, 1998.

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

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

[53] G.D. Plotkin, “A Structural Approach to Operational Semantics,” Report DAIMI FN-19, Computer Science Dept., Aarhus Univ., 1981.

[54] J.C. Reynolds, *Theories of Programming Languages.* Cambridge Univ. Press, 1998.

[55] R. Segala and N.A. Lynch, “Probabilistic Simulations for Probabilistic Processes,” *Nordic J. Comp.,* vol. 2, no. 2, pp. 250-273, 1995.

[56] A.N. Shiryaev, “Probability,” *Graduate Texts in Math.,* vol. 95, 1996.

[57] M. Sighireanu, “LOTOS NT User’s Manual,” version 2.4, technical report, INRIA Rhône-Alpes/VASY, 2004.

[58] A. Sokolova and E.P. de Vink, “Probabilistic Automata: System Types, Parallel Composition and Comparison,” *Validation of Stochastic Systems,* LNCS 2925, pp. 1-43, Springer-Verlag, 2004.

[59] W. Yi, P. Pettersson, and M. Daniels, “Automatic Verification of Real-Time Communicating Systems by Constraint Solving,” *Proc. Conf. Formal Techniques for Networked and Distributed Systems (FORTE ’94),* pp. 223-238, 1994.

[60] W. Yi, “Real-Time Behaviour of Asynchronous Agents,” *Proc. Int’l Conf. Concurrency Theory (CONCUR ’90),* pp. 502-520, 1990.

Henrik Bohnenkamp received the diploma degree in computer science from the University Erlangen/Nürnberg, Germany, in 1995 and the PhD degree in computer science from the University of Aachen, Germany, in 2002. He has held positions with the Computer Science Department of the University of Twente, the Netherlands, and is now a researcher with the group on software modeling and verification at the University (RWTH) Aachen, Germany. His research interests include modeling of probabilistic and stochastic systems, semantics, and specification-based testing. He is a member of the IEEE and the IEEE Computer Society.

Pedro R. D’Argenio received the BS degree (1993) and the MS degree (1994) in computer science at the Universidad Nacional de La Plata, Argentina, and the PhD degree in computer science (1999) from the Universiteit Twente, the Netherlands. Currently, he is a lecturer at the Universidad Nacional de Córdoba and a senior researcher for CONICET in Argentina. He also holds a visiting researcher position at the Universiteit Twente. His research interests are in formal methods to achieve dependable systems, particularly in model checking, process algebra, process semantics, and quantitative analysis.

Holger Hermanns studied at the University of Bordeaux, France, and the University of Erlangen/Nürnberg, Germany, where he received the diploma degree in computer science in 1993 (with honors) and the PhD degree from the Department of Computer Science in 1998 (with honors). From 1998 to 2006 he has been with the University of Twente, the Netherlands, holding an associate professor position since October 2001. Since 2003, he has headed the Dependable Systems and Software Group at Saarland University, Germany. His research interests include modeling and verification of concurrent systems, resource-aware embedded systems, and compositional performance and dependability evaluation.

Joost-Pieter Katoen received the master's degree (with honors, 1987) and the PhD degree (1996) in computer science, both from the University of Twente, the Netherlands. He held positions at the Universities of Erlangen/Nürnberg (Germany), Eindhoven, and Twente (the Netherlands) and worked at Philips Research. He is a full professor at the University (RWTH) of Aachen, Germany, and chairs the group on software modeling and verification. His research interests include modeling and verification of distributed and embedded systems, semantics, probabilistic model checking, and software verification. He is a member of the IEEE Computer Society.