QAPL 2005 Preliminary Version

PMaude: Rewrite-based Specification Language for Probabilistic Object Systems

Gul Agha[1] José Meseguer[2] Koushik Sen[3]

Department of Computer Science,
University of Illinois at Urbana Champaign, USA.

Abstract

We introduce a rewrite-based specification language for modelling probabilistic concurrent and distributed systems. The language, based on PMaude, has both a rigorous formal basis and the characteristics of a high-level rule-based programming language. Furthermore, we provide tool support for performing discrete-event simulations of models written in PMaude, and for statistically analyzing various quantitative aspects of such models based on the samples that are generated through discrete-event simulation. Because distributed and concurrent communication protocols can be modelled using *actors* (concurrent objects with asynchronous message passing), we provide an actor PMaude module. The module aids writing specifications in a probabilistic actor formalism. This allows us to easily write specifications that are purely probabilistic – and not just non-deterministic. The absence of such (un-quantified) non-determinism in a probabilistic system is necessary for a form of statistical analysis that we also discuss. Specifically, we introduce a query language called *Quantitative Temporal Expressions* (or QuaTEx in short), to query various quantitative aspects of a probabilistic model. We also describe a statistical technique to evaluate QuaTEx expressions for a probabilistic model.

1 Introduction

In modelling large-scale concurrent systems, it is infeasible to account for the complex interplay of the different factors that affect events in the system. For example, in a large scale computer network like the Internet, network delays, congestion, and failures affect each other in ways that make it infeasible to model the system deterministically. However, non-deterministic models do not allow us to reason about the likely behaviors of a system; *probabilistic* modelling and analysis is necessary to understand such behavior.

[1] Email: agha@cs.uiuc.edu
[2] Email: meseguer@cs.uiuc.edu
[3] Email: ksen@cs.uiuc.edu

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

A probabilistic model allows us to quantify a number of sources of indeterminacy in concurrent systems. The exact time duration of a behavior often depends on the schedulers, loads, etc. and may be represented by a stochastic process. Process or network failures may occur with a certain *rate*. Randomness can also come in explicitly: some parts of the system may implement randomized algorithms.

There has been considerable research on models of probabilistic systems. Both light-weight formalisms such as extensions of UML and SDL and rigorous formalisms based on process algebra [17,16], Petri-nets [23], and stochastic automata [13] has been proposed and successfully used to model and analyze probabilistic systems. The light-weight formalisms are closer to programming languages and easy for engineers to learn; however, some may lack a rigorous semantics. On the other hand, rigorous formalisms can be too cumbersome for engineers to adopt.

To bridge the gap between light-weight and rigorous formalisms, we propose a rewrite-based specification language, called PMaude, for specifying probabilistic concurrent systems. PMaude, which is based on probabilistic rewrite theories, has both a rigorous formal basis and the characteristics of a high-level programming language.[4] Furthermore, we provide tool support for performing *discrete-event simulations* of models written in PMaude and to *statistically* analyze various quantitative aspects of such models. In addition, because various distributed and concurrent communication protocols can be modelled using asynchronous message passing concurrent objects or actors [2,4], we provide an actor PMaude module to aid writing specifications in a *probabilistic-actor formalism*.

Our PMaude language extends standard rewrite theories with support for probabilities. Rewrite theories [24] have already been shown to be a natural and useful semantic framework which unifies different kinds of concurrent systems [24], as well as models of real-time [27]. The Maude system [11,12] provides an execution environment for rewrite theories. The *discrete-event simulator* for PMaude has been implemented as an extension of Maude.

Actor PMaude extends the actor model [2,4] of concurrent computation by allowing us to explicitly associate probability distribution with time for message delay and computation. Actors are inherently autonomous computational objects which interact with each other by sending asynchronous messages. The actor model has been formalized and applied to dependable computing [33] and software architecture [5].

A motivation for writing a specification in actor PMaude is that it allows us to easily write specifications that have *no un-quantified non-determinism*. In Section 3.1, we outline simple requirements which ensure that a specification written in actor PMaude is free of un-quantified non-determinism, i.e. all

[4] Note that there are other formalisms which provide both rigorous formal basis and the features of high-level programming languages [22,9]. PMaude differs from them as it extends rewrite theories rather than extending process-algebra or automata based formalisms.

non-determinism has been replaced by quantified non-determinism such as probabilistic choices and stochastic real-time. Absence of (un-quantified) non-determinism is necessary for the kind of statistical analysis that we propose. This analysis technique extends the existing numerical and statistical model-checking techniques [8,22,30,31]. In particular, in our statistical analysis we allow evaluation of quantitative temporal expressions, called QuaTEx, which allows us get more quantitative insight about a probabilistic model than what is possible using traditional model-checking of temporal properties.

This paper makes the following contributions:

1. We introduce PMaude, a tool for writing specifications in probabilistic rewrite theories. We also explain how models specified in PMaude are simulated in the underlying Maude language.
2. We provide an actor extension of probabilistic rewrite theories which we claim is a natural model to write various probabilistic network protocols. The extension also helps us to write specifications which are free from non-determinism. This is essential for the form of statistical analysis that we introduce.
3. We introduce a new query language QuaTEx to write *quantitative temporal expressions* which can be used to query various quantitative aspects of a probabilistic model with no non-determinism. We describe a statistical technique to evaluate such expressions using discrete-event simulation. We have implemented the technique as a part of the tool VeStA. Furthermore, we describe the integration of PMaude with VeStA.

The rest of the paper is organized as follows. Section 2 introduces PMaude along with its underlying formalism and a translator from PMaude modules to standard Maude modules. In Section 3 we describe actor PMaude module with examples. We introduce QuaTEx and a statistical evaluation technique for QuaTEx in Section 4 followed by a conclusion.

2 PMaude and its Underlying Formalism

In this section, we introduce PMaude and its underlying formalism starting with a brief primer on PMaude and an example. This is followed by a formal introduction to probabilistic rewrite theories along with background concepts and notations. We then explain how probabilistic models specified in PMaude are simulated in the underlying Maude language. The formalism of probabilistic rewrite theories is given to keep the paper self-contained. Further details about the formalism can be found in [20,21]. Readers can go to Section 2.5 skipping the formalisms given in Section 2.2, 2.3, and 2.4 without loss of continuity.

2.1 A Primer on PMaude

In a standard *rewrite theory* [11], transitions in a system are described by labelled conditional rewrite rules (keyword `crl`) of the form

$$\texttt{crl [L]: } t(\overrightarrow{x}) \Rightarrow t'(\overrightarrow{x}) \texttt{ if } C(\overrightarrow{x}) \tag{1}$$

where we assume that the condition C is purely equational. Intuitively, a conditional rule (with label `L`) of this form specifies a *pattern* $t(\overrightarrow{x})$ such that if some fragment of the system's state matches that pattern and satisfies the condition C, then a local transition of that state fragment, changing into the pattern $t'(\overrightarrow{x})$ can take place. In a *probabilistic rewrite rule* we add probability information to such rules. Specifically, our proposed probabilistic rules are of the form,

$$\texttt{crl [L]: } t(\overrightarrow{x}) \Rightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \texttt{ if } C(\overrightarrow{x}) \texttt{ with probability } \overrightarrow{y} := \pi(\overrightarrow{x}) \tag{2}$$

where the set of variables in the left hand side term $t(\overrightarrow{x})$ is $\overrightarrow{x}$, while some new variables $\overrightarrow{y}$ are present in the term $t'(\overrightarrow{x}, \overrightarrow{y})$ on the right hand side. Of course it is not necessary that *all* of the variables in $\overrightarrow{x}$ occur in $t'(\overrightarrow{x}, \overrightarrow{y})$. The rule will match a state fragment if there is a substitution θ for the variables $\overrightarrow{x}$ that makes $\theta(t)$ equal to that state fragment and the condition $\theta(C)$ is true. Because the right hand side $t'(\overrightarrow{x}, \overrightarrow{y})$ may have new variables $\overrightarrow{y}$, the next state is *not uniquely* determined: it depends on the choice of an additional substitution ρ for the variables $\overrightarrow{y}$. The choice of ρ is made according to the probability function $\pi(\theta)$, where π is not a fixed probability function, but a *family* of functions: one for each matching substitution θ of the variables $\overrightarrow{x}$.

It is important to note that our notion of probabilistic rewrite theory can express both probabilistic and non-deterministic behavior in the following sense: in a concurrent system, at any given point many different rules can fire. In a probabilistic rewrite theory, the choice of which rules will fire is *non-deterministic*. Once a match θ of a given probabilistic rule of the general form (2) at a given position has been chosen, then the subsequent *choice* of the substitution ρ for the variables $\overrightarrow{y}$ is made *probabilistically* according to the probability distribution function $\pi(\theta)$. In Fig. 1, we illustrate the interplay between non-determinism and probabilities by means of a simple example in PMaude, modelling a battery-operated clock with a reset-button. Comments in PMaude are prefixed with `***`.

Example 2.1 .

The module in Fig. 1 imports modules `POSREAL` and `PMAUDE` defining the positive real numbers and probability distributions, respectively. A clock in normal stable state is represented as a term `clock(T,C)`, where `T` is the time, and `C` is a real number representing the amount of charge left in the clock

```
pmod EXPONENTIAL-CLOCK is
*** the following imports positive real number module
    protecting POSREAL .

*** the following imports PMaude module that defines the distributions EXPONENTIAL,
*** BERNOULLI, GAMMA, etc.
    protecting PMAUDE .

*** declare a sort Clock
    sort Clock .
*** declare a constructor operator for Clock
    op clock : PosReal PosReal → Clock .
*** declares a constructor operator for a broken clock
    op broken : PosReal PosReal → Clock .

*** T is used to represent time of clock,
*** C represents charge in the clock's battery,
*** t represents time increment of the clock
    vars T C t : PosReal .
    var B : Bool .

    rl [advance]: clock(T,C) ⇒
                        if B then
                            clock(T+t,C-C/1000)
                        else
                            broken(T,C-C/1000)
                        fi
                with probability B:=BERNOULLI(C/1000) and t:=EXPONENTIAL(1.0) .

    rl [reset]: clock(T,C) ⇒ clock(0.0,C) .
endpm
```

Fig. 1. Clock illustrating probabilistic non-deterministic systems

battery. The key rule is `advance`, which has a new boolean variable `B` and a positive real number variable `t` in its righthand side. If all goes well (`B = true`), the clock increments its time by `t` and the charge is slightly decreased, but if `B = false`, the clock will go into state `broken(T,C-`$\frac{\texttt{C}}{1000}$`)`. Here the binary variable `B` (boolean in this case) is distributed according to the Bernoulli distribution with mean $\frac{\texttt{C}}{1000}$. Thus the value of `B` probabilistically *depends on the amount of charge* left in the battery: the lesser the charge left in the battery, the greater is the chance to break the clock. In this way, PMaude supports discrete probabilistic choice as in discrete-time Markov chains. The other extra variable `t` on the righthand side of the rule `advance` is distributed according to the exponential distribution with rate 1.0. Thus, PMaude also allows us to model stochastic continuous-time as found in continuous-time Markov chains. The `reset` rule, which resets the clock to time 0.0, does not have any extra variables in its righthand side and is therefore standard rewrite rule. Given a clock expression `clock(T,C)` one of the two rules `advance`, or `reset` is chosen non-deterministically to apply to the term `clock(T,C)`. If the rule `advance` is chosen, then the clock is advanced probabilistically.

A sample execution for the above module in PMaude can be obtained by first loading the module in an interactive session of PMaude interpreter and then giving a rewrite command with an initial ground term, say `clock`$(0.0, 1000)$. The result will be an execution in which the non-determinism about which rule to apply is resolved by a fair scheduler, but each application of the `advance` rule chooses the value of `B` and `t` probabilistically. Execution of a PMAUDE module requires transforming it into a corresponding Maude module that simulates its behavior, as explained in Section 2.5.

2.2 Background and Notation

A *membership equational theory* [26] is a pair (Σ, E), with Σ a *signature* consisting of a set K of *kinds*, for each kind $k \in K$ a set S_k of *sorts*, a set of *operator* declarations of the form $f : k_1 \dots k_n \to k$, with $k, k_1, \dots, k_n \in K$ and with E a set of *conditional* Σ-*equations* and Σ-*memberships* of the form

$$(\forall \overrightarrow{x})\ t = t' \Leftarrow u_1 = v_1 \wedge \dots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m$$
$$(\forall \overrightarrow{x})\ t : s \Leftarrow u_1 = v_1 \wedge \dots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m$$

The $\overrightarrow{x}$ denote *variables* in the terms t, t', u_i, v_i and w_j above. A membership $w : s$ with w a Σ-term of kind k and $s \in S_k$ asserts that w has sort s. Terms that do not have a sort are considered *error* terms. This allows membership equational theories to specify partial functions within a total framework. A Σ-*algebra* B consists of a K-indexed family of sets $X = \{B_k\}_{k \in K}$, together with

(i) for each $f : k_1 \dots k_n \to k$ in Σ a function $f_B : B_{k_1} \times \dots \times B_{k_n} \to B_k$

(ii) for each $k \in K$ and each $s \in S_k$ a subset $B_s \subseteq B_k$.

We denote the algebra of terms of a membership equational signature by T_Σ. The *models* of a membership equational theory (Σ, E) are those Σ-algebras that satisfy the equations E. The inference rules of membership equational logic are *sound* and *complete* [26]. Any membership equational theory (Σ, E) has an *initial algebra* of terms denoted $T_{\Sigma/E}$ which, using the inference rules of membership equational logic and assuming Σ *unambiguous* [26], is defined as a quotient of the term algebra T_Σ by

- $t \equiv_E t' \quad \Leftrightarrow \quad E \vdash t = t'$
- $[t]_{\equiv_E} \in T_{\Sigma/E,s} \quad \Leftrightarrow \quad E \vdash t : s$

In [10] the usual results about *equational simplification*, *confluence*, *termination*, and *sort-decreasingness* are extended in a natural way to membership equational theories . Under those assumptions a membership equational theory can be executed by equational simplification using the equations from left to right, perhaps modulo some *structural* axioms A (e.g. associativity, commutativity, and identity). The initial algebra with equations E and structural axioms A is denoted $T_{\Sigma, E \cup A}$. If E is confluent, terminating, and sort-decreasing modulo A [10], the isomorphic algebra of fully simplified terms (canonical forms) modulo A is denoted by $Can_{\Sigma, E/A}$. The notation $[t]_A$ represents the A-equivalence class of a term t fully simplified by the equations E.

In a standard *rewrite theory* [24], transitions in a system are described by labelled conditional rewrite rules of the form

$$\texttt{crl [L]} : t(\overrightarrow{x}) \Rightarrow t'(\overrightarrow{x}) \texttt{ if } C(\overrightarrow{x})$$

Intuitively, a rule (with label L) of this form specifies a *pattern* $t(\overrightarrow{x})$ such that if some fragment of the system's state matches that pattern and satisfies the condition C, then a local transition of that state fragment, changing into the pattern $t'(\overrightarrow{x})$ can take place. The Maude system [11,12] provides an execution environment for membership equational theories and for rewrite theories of the form (Σ, E, R), with (Σ, E) a membership equational theory, and R a collection of conditional rewrite rules. Several examples of Maude specification can be found in [25,12].

To succinctly define probabilistic rewrite theories, we use a few basic notions from axiomatic probability theory. A σ-algebra on a set X is a collection $\mathcal{F}$ of subsets of X, containing X itself and closed under complementation and finite or countably infinite unions. For example the power set $\mathcal{P}(X)$ of a set X is a σ-algebra on X. The elements of a σ-algebra are called *events*. We denote by $\mathcal{B}_{\mathbb{R}}$ the smallest σ-algebra on $\mathbb{R}$ containing the sets $(-\infty, x]$ for all $x \in \mathbb{R}$. We also remind the reader that a *probability space* is a triple $(X, \mathcal{F}, \pi)$ with $\mathcal{F}$ a σ-algebra on X and π a *probability measure function*, defined on the σ-algebra $\mathcal{F}$ which evaluates to 1 on X and distributes by addition over finite or countably infinite unions of disjoint events. For a given σ-algebra $\mathcal{F}$ on X, we denote by $PFun(X, \mathcal{F})$ the set

$$\{\pi \mid (X, \mathcal{F}, \pi) \text{ is a probability space}\}$$

2.3 Probabilistic Rewrite Theories

We next define probabilistic rewrite theories after the following definition.

Definition 1 (E/A-canonical ground substitution) *An* E/A-canonical ground substitution *for variables* $\overrightarrow{x}$ *is a function* $[\theta]_A \colon \overrightarrow{x} \to Can_{\Sigma,E/A}$. *We use the notation* $[\theta]_A$ *for such functions to emphasize that an* E/A*-canonical substitution is induced by an ordinary substitution* $\theta \colon \overrightarrow{x} \to T_\Sigma$ *where, for each* $x \in \overrightarrow{x}$, $\theta(x)$ *is fully simplified by* E *modulo* A. *Of course,* $[\theta]_A = [\rho]_A$ *iff for each rule* $x \in \overrightarrow{x}$, $[\theta(x)]_A = [\rho(x)]_A$. *We use* $CanGSubst_{E/A}(\overrightarrow{x})$ *to denote the set of all* E/A*-canonical ground substitutions for the set of variables* $\overrightarrow{x}$.

Intuitively an E/A-canonical ground substitution represents a substitution of ground terms from the term algebra T_Σ for variables of the corresponding sorts, so that all of the terms have already been reduced as much as possible by the equations E modulo the structural axioms A. For example the substitution 10.0×2.0 for a variable of sort PosReal is *not* a canonical ground substitution but a substitution of 20.0 for the same variable is a canonical ground substitution. We now proceed to define probabilistic rewrite theories.

Definition 2 (Probabilistic rewrite theory) *A* probabilistic rewrite theory *is a 4-tuple* $\mathcal{R} = (\Sigma, E \cup A, R, \pi)$*, with* $(\Sigma, E \cup A, R)$ *a rewrite theory with the rules* $r \in R$ *of the form*

$$L : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \textit{ if } C(\overrightarrow{x})$$

where

- $\overrightarrow{x}$ *is the set of variables in* t*,*
- $\overrightarrow{y}$ *is the set of variables in* t' *that are not in* t*; thus,* t' *might have variables coming from the set* $\overrightarrow{x} \cup \overrightarrow{y}$*; however, it is not necessary that all variables in* $\overrightarrow{x}$ *occur in* t'*,*
- C *is a condition of the form* $(\bigwedge_j u_j = v_j) \wedge (\bigwedge_k w_k : s_k)$*, i.e.,* C *is a conjunction of equations and memberships, and all the variables in* u_j*,* v_j *and* w_k *are in* $\overrightarrow{x}$*,*

and π *is a function assigning to each rewrite rule* $r \in R$ *a function*

$$\pi_r : [\![C]\!] \to \mathit{PFun}(\mathit{CanGSubst}_{E/A}(\overrightarrow{y}), \mathcal{F}_r)$$

where $[\![C]\!] = \{[\mu]_A \in \mathit{CanGSubst}_{E/A}(\overrightarrow{x}) \mid E \cup A \vdash \mu(C)\}$ *is the set of* E/A*-canonical substitutions for* $\overrightarrow{x}$ *satisfying the condition* C*, and* $\mathcal{F}_r$ *is a* σ*-algebra on* $\mathit{CanGSubst}_{E/A}(\overrightarrow{y})$*. We denote a rule* r *together with its associated function* π_r*, by the notation*

$$\texttt{crl [L]:}t(\overrightarrow{x}) \Rightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \texttt{ if } C(\overrightarrow{x}) \texttt{ with probability } \overrightarrow{y} := \pi_r(\overrightarrow{x})$$

If the set $\mathit{CanGSubst}_{E/A}(\overrightarrow{y})$ *is empty because* $\overrightarrow{y}$ *is empty then* $\pi_r(\overrightarrow{x})$ *is said to define a* trivial distribution*; this corresponds to an ordinary rewrite rule with no probability. If* $\overrightarrow{y}$ *is nonempty but* $\mathit{CanGSubst}_{E/A}(\overrightarrow{y})$ *is empty because there is no canonical substitution for some* $y \in \overrightarrow{y}$ *because the corresponding sort or kind is empty, then the rule is considered* erroneous *and will be disregarded in the semantics.*

We denote the class of probabilistic rewrite theories as **PRwTh**. For the specification in Example 2.1, the rule `advance` has two variables `B` and `t` on the righthand side. The possible substitutions for `B` are `true` and `false` with `true` chosen with probability $\frac{\texttt{C}}{1000}$. Similarly, the possible substitutions for `t` are positive real numbers sampled from an exponential distribution with rate 1.0.

2.4 Semantics of Probabilistic Rewrite Theories

Let $\mathcal{R} = (\Sigma, E \cup A, R, \pi)$ be a probabilistic rewrite theory such that:

(i) E is confluent, terminating and sort-decreasing modulo A [10].

(ii) the rules R are coherent with E modulo A [11].

Definition 3 (Context) *A* context $\mathbb{C}$ *is a* Σ*-term with a single occurrence of a single variable,* $\odot$*, called the* hole*. Two contexts* $\mathbb{C}$ *and* $\mathbb{C}'$ *are* A*-equivalent if and only if* $A \vdash (\forall \odot)\ \mathbb{C} = \mathbb{C}'$.

Notice that the relation of A-equivalence for contexts defined above is an equivalence relation on the set of contexts. We use $[\mathbb{C}]_A$ for the equivalence class containing context $\mathbb{C}$.

Definition 4 (R/A-matches) *Given* $[u]_A \in Can_{\Sigma,E/A}$*, its* R/A-matches *are triples* $([\mathbb{C}]_A, r, [\theta]_A)$*, where if* $r \in R$ *is a rule*

$$\texttt{rl [L]:}t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y})\ \texttt{if}\ C(\overrightarrow{x})\ \texttt{with probability}\ \overrightarrow{y} := \pi_r(\overrightarrow{x})$$

then $[\theta]_A \in [\![C]\!]$*, that is* $[\theta]_A$ *satisfies condition* C*, and* $[u]_A = [\mathbb{C}(\odot \leftarrow \theta(t))]_A$*, so* $[u]_A$ *is the result of applying* θ *to the term* $t(\overrightarrow{x})$ *and placing it in the context.*

For example, the R/A-matches for the term `clock`$(75.0, 800.0)$ in Example 2.1 are as follows:

- $([\odot]_A, \texttt{advance}, [T \leftarrow 75.0, C \leftarrow 800.0])$
- $([\odot]_A, \texttt{reset}, [T \leftarrow 75.0, C \leftarrow 800.0])$

Definition 5 (E/A-canonical one-step $\mathcal{R}$-rewrite) *An* E/A*-canonical one-step* $\mathcal{R}$*-rewrite is a labelled transition of the form,*

$$[u]_A \xrightarrow{([\mathbb{C}]_A, r, [\theta]_A, [\rho]_A)} [v]_A$$

where

(i) $[u]_A, [v]_A \in Can_{\Sigma,E/A}$

(ii) $([\mathbb{C}]_A, r, [\theta]_A)$ *is an* R/A*-match of* $[u]_A$

(iii) $[\rho]_A \in CanGSubst_{E/A}(\overrightarrow{y})$

(iv) $[v]_A = [\mathbb{C}(\odot \leftarrow t'(\theta(\overrightarrow{x}), \rho(\overrightarrow{y})))]_A$

The above definition describes the steps involved in a one-step computation of a **PRwTh**. First, a R/A-match $([\mathbb{C}]_A, r, [\theta]_A)$ is chosen non-deterministically for the lefthand side of r, and then a substitution $[\rho]_A$ is chosen for the new variables $\overrightarrow{y}$ in the r's righthand side according to the probability function $\pi_r([\theta]_A)$. These two substitutions are then applied to the term $t'(\overrightarrow{x}, \overrightarrow{y})$ to produce the final term v whose equivalence class $[v]_A$ is the result of the step of computation. The non-determinism associated with the choice of the R/A-match must be removed in order to associate a probability space over the space of computations (which are infinite sequences of canonical one-step $\mathcal{R}$-rewrites). The non-determinism is removed by what is called an *adversary* of the system, which defines a probability distribution over the set of R/A-matches. In [20], we describe the association of a probability space over the set of computation paths. We have also shown in [20] that probabilistic rewrite theories have great expressive power. They can express various known models of probabilistic systems like continuous-time Markov chains [32], probabilistic non-deterministic systems [28,29], and generalized semi-Markov processes [14].

2.5 Simulating PMaude *Specifications in Maude*

Due to their non-determinism, probabilistic rewrite rules *are not directly executable*. Consider for example the `advance` rule in Example 2.1 that advances the clock. There are two new variables in its righthand side, namely, a Boolean variable `B`, which will determine whether the clock will continue to function normally or will break, and a positive real variable `t`, which will determine the actual time advance of the clock.

However, probabilistic systems specified in PMaude *can be simulated in Maude*. This is accomplished by transforming a PMaude specification into a corresponding Maude specification in which actual values of the new variables appearing in the righthand side of a probabilistic rewrite rule are obtained by *sampling* the corresponding distribution functions. For example, in the `advance` rule in our clock example, the Boolean variable `B` must be sampled according to the Bernoulli distribution $\mathtt{BERNOULLI}(\frac{\mathtt{C}}{1000})$, whereas the positive real variable must be sampled according to the exponential distribution $\mathtt{EXPONENTIAL}(1.0)$.

This theory transformation uses three key Maude modules as basic infrastructure, namely, `COUNTER`, `RANDOM`, and `SAMPLER`. The module `COUNTER` provides a built-in strategy for the application of the non-deterministic rewrite rule:

```
rl counter ⇒ N:Nat .
```

that rewrites the constant `counter` to a natural number. The built-in strategy applies this rule so that the natural number obtained after applying the rule is exactly the successor of the value obtained in the preceding rule application. The `RANDOM` module is a built-in Maude module providing a random number generator function called `random`. The `SAMPLER` module provides sampling functions for different probability distributions. In the above `advance` rule, the needed sampling functions are

```
op EXPONENTIAL : PosReal → PosReal .
op BERNOULLI : PosReal → Bool .
```

The key rule in the `SAMPLER` module is the rule

```
rl [rnd] : rand ⇒ float(random(counter + 1) / 4294967296) .
```

which rewrites the constant `rand` to a floating point number between 0 and 1 pseudo-randomly chosen according to the uniform distribution. This floating point number is obtained by converting the rational number `random(counter + 1) / 4294967296` into a positive real number, where `4294967296` is the maximum value that the `random` function can attain. The rewrite rules defining the semantics of the `EXPONENTIAL` and `BERNOULLI` sampling functions are then

```
rl EXPONENTIAL(R) ⇒ (- log(rand)) / R .
rl BERNOULLI(R) ⇒ if rand < R then true else false fi .
```

The result of transforming the PMaude module in Example 2.1, is then the module

```
mod EXPONENTIAL-CLOCK-TRANSFORMED is
    protecting POSREAL .
    protecting PMAUDE .

    sort Clock .
    op clock : PosReal PosReal → Clock .
    op broken : PosReal PosReal → Clock .

    vars T C t : PosReal .
    var B : Bool .

    rl [advance]: clock(T,C) ⇒
                    if BERNOULLI(C/1000) then
                        clock(T+EXPONENTIAL(1.0),C-C/1000)
                    else
                        broken(T,C-C/1000)
                    fi

    rl [reset]: clock(T,C) ⇒ clock(0.0,C) .
endm
```

We can then use this transformed module to simulate the original `EXPONENTIAL-CLOCK` PMaude module. In particular, as explained in Section 4, we can use the results of performing Monte-Carlo simulations in this way to formally analyze probabilistic properties of a system, provided all non-determinism has been eliminated from the original PMaude module. In example 2.1 this elimination of non-determinism has not happened because the `reset` rule and the `advance` rule could both be applied to a clock. However, it would be easy to transform this example into one where such non-determinism has been replaced by probabilities. In section 3 we give a general method to specify probabilistic object-oriented distributed systems in a way that eliminates all non-determinism and makes them amenable to the form of statistical analysis discussed in Section 4.

3 Actor PMaude

An actor [2,4] is a concurrent object encapsulating a state and having a unique name. Actors communicate asynchronously by sending messages to each other. On receiving a message, an actor changes its state and sends messages to other actors. Actors provide a natural formalism to model and reason about distributed and concurrent systems. We provide the module, actor PMaude, to aid high level modelling of various concurrent and distributed object systems.

Another motivation for writing a specification in actor PMaude is that it allows us to easily write specifications that have *no non-determinism*. To ensure absence of non-determinism in an actor PMaude specification, we outline simple requirements in Section 3.1. Absence of non-determinism is necessary for statistical analysis as described briefly in Section. 4.

In actor PMaude, we introduce soft real-time (i.e. stochastic) to capture the dynamics of various elements of a system. Specifically, we assume that both message passing and computation by an actor on receiving a message may take some positive real-valued time. This time can be distributed according to some continuous probability distribution function. In a actor PMaude specification, in addition to the functional description of the actors and their computations, we explicitly describe the probability distributions associated with message passing time and computation time. We also allow time associated with message passing or computation to be zero, to indicate synchronous communication and instant computation, respectively. We next describe the actor PMaude module along with the semantics for *one-step computation* which is required for discrete event simulation.

The definition of the various sorts and operators for the actor PMaude module is given in Fig. 2. A term of sort `Actor` represents an actor. An actor has a unique name (a term of sort `ActorName`) and a list of named attributes (a term of the sort `AttributeList`). The attribute list of an actor, which is a list of terms of the sort `Attribute`, represents the state of an actor. An actor is constructed by the mixfix operator[5] ⟨`name:_|_`⟩ that maps an actor name and a list of attributes to an actor.

A message is represented by a term of sort `Msg`. A message contains an address or the name of the actor to which it is targeted and a content (a term of sort `Content`). A message is constructed by the operator $\_ \leftarrow \_$ that maps an actor name and a content to a message. An actor on receiving a message can change it state, i.e. its attributes, and can send out messages to other actors.

An actor or a message can be generically represented by a term of sort `Object`, whose subsorts are `Actor` and `Msg`. To model soft real-time associated with message passing delay or actor computation, we make a message or an actor, respectively, inactive up to a given global time by enclosing them between square brackets `[ ]`. A term of sort `ScheduledObject` represents an object which is not yet active or available to the system. We call such objects *scheduled objects*. A scheduled object is constructed by the operator `[_,_]` that maps a time (a term of the sort `PosReal`) and an object (i.e. an actor or a message) to a scheduled object. The time indicates the global time at which the object will become available to the system.

A term of sort `Config` represents a multiset of objects, scheduled objects, and a global time combined with an empty syntax (juxtaposition) multiset union operator that is declared associative and commutative. The *global state of a system* is represented by a term of the sort `Config` containing

(i) a multiset of objects,

(ii) a multiset of scheduled objects, and

[5] The underscores (_) in a mixfix operator represent the placeholders for its arguments.

(iii) a global time (a term of the sort `PosReal`)[6].

The ground terms `empty`, `nil`, and `null` represents constants of the sorts `Content`, `AttributeList`, and `Config`, respectively.

The module also defines a special `tick` rule which is omitted from Fig. 2 for brevity. The description of the `tick` rule is given below, where we define an one-step computation of a model written in actor PMaude.

One-Step Computation:

An *one-step computation* of a model written in actor PMaude is a transition of the form

$$[u]_A \xrightarrow{\neg\mathtt{tick}}{}^{*} [v]_A \xrightarrow{\mathtt{tick}} [w]_A$$

where

(i) $[u]_A$ is a canonical term of sort `Config`, representing the global state of a system,

(ii) $[v]_A$ is term obtained after a sequence (zero or more) of one-step rewrites such that
- in none of those rewrites is the `tick` rule applied, and
- $[v]_A$ *cannot be further rewritten* by applying any rule except the `tick` rule.

(iii) $[w]_A$ is obtained after a one-step rewrite of $[v]_A$ by applying the `tick` rule, which does the following
- finds and removes the scheduled object, if one exists, with the smallest global time, say `[T',Obj]`, from the term $[v]_A$ to a term, say $[v']_A$,
- adds the term `Obj` to $[v']_A$ through multiset union to get the term $[v'']_A$, and
- replaces the global time of the term $[v'']_A$ with `T'` to get the final term $[w]_A$.

Such a one-step computation represents a single step in a discrete-event simulation of a model written in actor PMaude.

Example 3.1 As an example, let us consider the model in Fig. 3. In the example, a client `c` continuously sends messages to a server `s`. The time interval between the messages is distributed exponentially with rate 2.0. The message sending of the client is triggered when it receives a self-sent message of the form `(C`$\leftarrow$ `empty)`. The delay associated with the message from the client to the server is distributed exponentially with rate 10.0 (see rule labelled `send`). The message contains a natural number which is incremented by 1 by the client, each time it sends a message. The server, when not busy, can receive a message and increment its attribute `total` by the number received in the message (see rule labelled `compute`). If the server is busy processing

[6] Note that `PosReal` is a subset of `Configuration`.

```
mod ACTORS is
    protecting PosReal .

    sorts ActorName Attribute AttributeList Content .
    sorts Actor Msg Object Config ScheduledObject .
    subsort Attribute < AttributeList .
    subsort Actor < Object .
    subsort Msg < Object .
    subsort Object < Config .
    subsort PosReal < Config .
    subsort ScheduledObject < Config .

    op empty : → Content .
    op _← _ : ActorName Content → Msg .
    op ⟨name:_|_⟩ : ActorName AttributeList → Actor .
    op nil : → AttributeList .
    op null : → Config .

    op __ : Config Config → Config [assoc comm id: null] .
    op _,_ : AttributeList AttributeList → AttributeList [assoc id: nil] .
    op [_,_] : PosReal Object → ScheduledObject .
    *** tick rule is omitted for brevity
endm
```

Fig. 2. Actor PMaude module

a message (computation time is exponentially distributed with rate 1.0), it drops any message it receives (see rule labelled `busy-drop`). Note that we can modify the rule `busy-drop` to allow the server actor to enqueue any message it receives when it is busy.

The rule for sending a message by a client `C` to a server `S` is labelled by `send`. The lefthand side of the rule matches a fragment of the global state consisting of a client actor of the form ⟨`name: C | counter: N, server: S`⟩, a message of the form `(C← empty)`, and a global time of the form `T`. The rule states that the client `C`, on receiving an empty message, produces two messages: an empty message to itself and a message to a server, whose name is contained in its attribute `server`. Both the messages were produced as scheduled objects to represent that they are inactive till the delay time associated with the messages has elapsed. The delay times t_1 and t_2 are substituted probabilistically.

Note that the model has no non-determinism. All non-determinism has been replaced by probabilistic choices. A model with no non-determinism is a key requirement for our statistical analysis technique briefly described in Section. 4. We next give sufficient conditions to ensure that a specification written in actor PMaude has no non-determinism.

3.1 *Sufficient conditions for absence of un-quantified non-determinism in an actor* PMaude *specification:*

(i) The initial global state of the system or the initial configuration can have at most one non-scheduled message.

(ii) The computation performed by any actor after receiving a message must

```
apmod SIMPLE-CLIENT-SERVER is
 protecting PMAUDE .
 including ACTORS .
 protecting NAT .

 vars t t1 t2 T : PosReal .
 vars C S : ActorName .
 vars N M : Nat .
 op counter:_ : Nat → Attribute .
 op server:_ : ActorName → Attribute .
 op total:_ : Nat → Attribute .
 op ctnt : Nat → Content .

 rl [send]: ⟨name: C | counter: N, server: S⟩ (C← empty) T ⇒
   ⟨name: C | counter: N+1, server: S⟩ [T+t1,(C← empty)] [T+t2,(S← ctnt(N))] T
        with probability t1:=EXPONENTIAL(2.0) and t2:=EXPONENTIAL(10.0) .

 rl [compute]: ⟨name: S | total: M⟩ (S← ctnt(N)) T ⇒ [T+t,⟨name: S | total: M+N⟩] T
        with probability t:=EXPONENTIAL(1.0) .

 rl [busy-drop]: [t,⟨name: S | total: M⟩] (S← ctnt(N)) ⇒ [t,⟨name: S | total: M⟩] .

 op init : → Config .
 op c : → ActorName .
 op s : → ActorName .
 eq init = ⟨name: c | counter: 0, server: s⟩ ⟨name: s | total: 0⟩ (c← empty) 0.0 .

endapm
```

Fig. 3. A simple Client-Server model with exponential distribution on message sending delay and computation time by the server

have no un-quantified non-determinism; however, there may be probabilistic choices.

(iii) The messages produced by an actor in a particular computation (i.e. on receiving a message) can have at most one non scheduled message.

(iv) No two scheduled objects become active at the same global time. This is ensured by associating continuous probability distributions with message delays and computation time.

We next provide the specification of a practical system to show the expressiveness of actor PMaude.

Example 3.2 The model of a symmetric polling server [19] with 5-stations is given in Fig. 4. Each station has a single-message buffer and they are cyclically attended to by a single server. The server polls a station i. If there is a message in the buffer of station i, then the server serves the station. Once the station is served, or once the station is polled in case the station has an empty buffer, the server moves on to poll the station $(i+1)$ modulo N, where N is the number of stations. The polling time, the service time, and the time for arrival of a message at each station is exponentially distributed. Note that this model can be represented by a continuous-time Markov chain.

In Fig. 4, we modelled each station and the server as actors. Messages that arrive at each station-actor are modelled as self-sending scheduled messages having exponentially distributed delays (see rule labelled `produce`). The start of polling of a station by the server is modelled as an instantaneous `poll` message (i.e. with no delay) sent by the server to the station (see rule labelled `next`). On receiving a `poll` message, a station sends itself a scheduled `serve` message (see rule labelled `poll`), i.e. a message having delay equal to the polling time. On receiving a `serve` message, if the station finds that its buffer is empty, it sends an instantaneous `next` message (i.e. with no message delay) to the server indicating that the server needs to poll the next station (see rule labelled `serve`). Otherwise, if the buffer has a message (indicated by non-zero value of the attribute `buf`), it sends itself a scheduled `done` message (i.e. a message having delay equal to the serving time). On receiving a `done` message, the station sends an instantaneous `next` message (i.e. with no message delay) to the server indicating that the server needs to poll the next station (see rule labelled `served`).

Note that the model has no un-quantified non-determinism, since it meets the conditions given in Section 3.1.

A more complex example of modelling and analysis of a denial of service resistant TCP/IP protocol can be found in [3].

4 QuaTEx

Once a probabilistic system has been specified in PMaude using criteria such as those in Section 3.1 that ensure that there is no non-determinism, we want to formally analyze the system by evaluating various quantitative properties of the system. In this section, we introduce a language to express various quantitative properties of a probabilistic system. We also give a statistical technique to evaluate such properties.

To query various quantitative aspects of a probabilistic model, we introduce a query language called *Quantitative Temporal Expressions* (or QuaTEx in short). The language is mainly motivated by probabilistic computation tree logic (PCTL) [15] and Eagle [7]. In QuaTEx, some example queries that can be encoded are as follows:

(i) Out of 100 clients, what is the expected number of clients that successfully connect to a server under a *denial of service* attack?

(ii) What is the probability that a client connected to a server within 10 seconds after it initiated the connection request?

We next introduce the notation that we will use to describe the syntax and the semantics of QuaTEx, followed by a few motivating examples.

We assume that an execution path is an infinite sequence

$$\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$$

where s_0 is the unique initial state of the system, typically a term of sort `Config` representing the initial global state, and s_i is the state of the system after the i^{th} computation step. If the k^{th} state of this sequence cannot be

```
apmod SYMMETRIC-POLLING is
 protecting PMAUDE .  including ACTORS .  protecting NAT .  protecting POSREAL .

*** Variable declarations.
  vars t T : PosReal .  vars C S : ActorName .  vars N M : Nat .

*** Operator declarations.
  op buf:_ : Nat → Attribute .
  op server:_ : ActorName → Attribute .
  op client:_ : Nat → Attribute .
  op station:_ : Nat → ActorName .
  ops poll serve done next : → Content .
  op increment : Nat → Nat .

*** Each station produces messages at the rate of 0.2. For this each station sends a message
***  to itself with message delay exponentially distributed with rate 0.2.
  rl [produce]: ⟨name: C | buf: M, server: S⟩ (C← empty) T
        ⇒⟨name: C | buf: 1, server: S⟩ [T+t,(C← empty)] T
          with probability t:=EXPONENTIAL(0.2) .

*** On receiving a poll message from the server, the station sends a scheduled serve message
***  to itself to imitate the time associated with polling.
  rl [poll]: ⟨name: C | buf: M, server: S⟩ (C← poll) T
        ⇒ ⟨name: C | buf: M, server: S⟩ [T+t,(C← serve)] T
          with probability t:=EXPONENTIAL(200.0) .

*** On receiving a serve message, if the buffer is empty then the station sends a next message
***  to the server; otherwise, it send a scheduled done message to itself.
  rl [serve]: ⟨name: C | buf: M, server: S⟩ (C← serve) T ⇒
        if M > 0 then
            ⟨name: C | buf: M, server: S⟩ [T+t,(C← done)] T
        else
            ⟨name: C | buf: M, server: S⟩ (S← next) T
        fi with probability t:=EXPONENTIAL(1.0) .

*** On receiving a done message, the station sends a next message to the server.
  rl [served]: ⟨name: C | buf: M, server: S⟩ (C← done)
        ⇒⟨name: C | buf: 0, server: S⟩ (S← next) .

*** On receiving a next message, the server sends a poll message to the next station.
  rl [next]: ⟨name: S | client: N⟩ (S← next) T
        ⇒ ⟨name: S | client: increment(N)⟩ (station(N)← poll) T .

*** Define increment as increment(N) = (N+1) modulo 5, which is the number of stations
  eq increment(N) = if N >= 5 then 1 else N+1 fi .
*** Create the initial configuration with 5 stations and 1 server and a next message.
  op init : → Config .
  op s : → ActorName .
  eq init = ⟨name: s | client: 1⟩ (s← next) 0.0 ⟨name: station(1) | buf: 1, server: s⟩
            ⟨name: station(2) | buf: 1, server: s⟩ ⟨name: station(3) | buf: 1, server: s⟩
            ⟨name: station(4) | buf: 1, server: s⟩ ⟨name: station(5) | buf: 1, server: s⟩ .

endapm
```

Fig. 4. Symmetric Polling System with 5-stations

rewritten any further (i.e. is absorbing), then $s_i = s_k$ for all $i \geq k$.

We denote the i^{th} state in an execution path π by $\pi[i] = s_i$. Also, we denote the suffix of a path π starting at the i^{th} state by $\pi^{(i)} = s_i \rightarrow s_{i+1} \rightarrow s_{i+2} \rightarrow \cdots$. We let $Path(s)$ be the set of execution paths starting at state s. Note that, because the samples are generated through discrete-events simulation of a PMaude model with no non-determinism, $Path(s)$ is a measurable set and has an associated probability measure. This is essential to compute the expected value of a path expression from a given state.

4.1 QuaTEx *through Examples*

The language QuaTEx, which is designed to query various quantitative aspects of a probabilistic model, allows us to write temporal query expressions like temporal formulas in a temporal logic. It supports a framework for parameterized recursive temporal operator definitions using a few primitive non-temporal operators and a temporal operator ($\bigcirc$). The temporal operator $\bigcirc$, called the *next* operator, takes an expression at the next state and makes it an expression for the current state. For example, suppose we want to know *"the probability that along a random path from a given state, the client C gets connected with S within* 100 *time units."* This can be written as the following query:

$$\begin{aligned}
&\texttt{IfConnectedInTime}(t) = \underline{\texttt{if}}\ t > \mathit{time}()\ \underline{\texttt{then}}\ 0\ \underline{\texttt{else}}\ \underline{\texttt{if}}\ \mathit{connected}()\ \underline{\texttt{then}}\ 1\\
&\qquad\qquad\qquad\qquad\qquad \underline{\texttt{else}}\ \bigcirc(\texttt{IfConnectedInTime}(t))\ \underline{\texttt{fi}}\ \underline{\texttt{fi}};\\
&\underline{\texttt{eval}}\ \mathbf{E}[\texttt{IfConnectedInTime}(\mathit{time}() + 100)];
\end{aligned}$$

The first four lines of the query define the recursive temporal operator $\texttt{IfConnectedInTime}(t)$, which returns 1, if along an execution path C gets connected to S within time t and returns 0 otherwise. The state function *time()* returns the global time associated with the state; the state function *connected*() returns true, if in the state, C gets connected with S and returns false otherwise. Then the state query at the fifth line returns the expected number of times C gets connected to S within 100 time units along a random path from a given state. This number lies in $[0, 1]$ since along a random path either C gets connected to S within 100 time units or C does not get connected to S within 100 time units. In fact, this expected value is equal to the probability that along a random path from the given state, the client C gets connected with S within 100 time units.

A further rich query is as follows:

$$\begin{aligned}
&\texttt{NumConnectedInTime}(t, \mathit{count}) = \underline{\texttt{if}}\ t > \mathit{time}()\ \underline{\texttt{then}}\ \mathit{count}\\
&\qquad\qquad \underline{\texttt{else}}\ \underline{\texttt{if}}\ \mathit{anyConnected}()\ \underline{\texttt{then}}\ \bigcirc(\texttt{NumConnectedInTime}(t, 1 + \mathit{count}))\\
&\qquad\qquad\qquad\qquad \underline{\texttt{else}}\ \bigcirc(\texttt{NumConnectedInTime}(t, \mathit{count}))\ \underline{\texttt{fi}}\ \underline{\texttt{fi}};\\
&\underline{\texttt{eval}}\ \mathbf{E}[\texttt{NumConnectedInTime}(\mathit{time}() + 100, 0)]
\end{aligned}$$

In this query, the state function *anyConnected*() returns true if any client C_i gets connected to S in the state. We assume that in a given execution path, at any state, at most one client gets connected to S.

$$
\begin{aligned}
Q &::= D \;\; \underline{\texttt{eval}} \; \mathbf{E}[\mathit{PExp}]; \\
D &::= \; \text{set of } \mathit{Defn} \\
\mathit{Defn} &::= N(x_1, \ldots, x_m) = \mathit{PExp}; \\
\mathit{SExp} &::= c \mid f \mid F(\mathit{SExp}_1, \ldots, \mathit{SExp}_k) \mid x_i \\
\mathit{PExp} &::= \mathit{SExp} \mid \bigcirc N(\mathit{SExp}_1, \ldots, \mathit{SExp}_n) \\
&\quad\; \mid \underline{\texttt{if}} \;\; \mathit{SExp} \;\; \underline{\texttt{then}} \;\; \mathit{PExp}_1 \;\; \underline{\texttt{else}} \;\; \mathit{PExp}_2 \;\; \underline{\texttt{fi}}
\end{aligned}
$$

Fig. 5. Syntax of QuaTEx

4.2 Syntax of QuaTEx

The syntax of QuaTEx is given in Fig. 5. A query in QuaTEx consists of a set of definitions D followed by a query of the expected value of a path expression *PExp*. In QuaTEx, we distinguish between two kinds of expressions, namely, *state expressions* (denoted by *SExp*) and *path expressions* (denoted by *PExp*); a path expression is interpreted over an execution path and a state expression is interpreted over a state. A definition $\mathit{Defn} \in D$ consists of a definition of a *temporal operator*. A temporal operator definition consists of a name N and a set of formal parameters on the left-hand side, and a path expression on the right-hand side. The formal parameters denote the *freeze formal parameters*. When using a temporal operator in a path expression, the formal parameters are replaced by state expressions. A state expression can be a constant c, a function f that maps a state to a concrete value, a k-ary function mapping k state expressions to a state expression, or a formal parameter. A path expression can be a state expression, a next operator followed by an application of a temporal operator defined in D, or a conditional expression $\underline{\texttt{if}}$ *SExp* $\underline{\texttt{then}}$ PExp_1 $\underline{\texttt{else}}$ PExp_2 $\underline{\texttt{fi}}$. We assume that expressions are properly typed. Typically, these types would be `integer`, `real`, `boolean` etc. The condition *SExp* in the expression $\underline{\texttt{if}}$ *SExp* $\underline{\texttt{then}}$ PExp_1 $\underline{\texttt{else}}$ PExp_2 $\underline{\texttt{fi}}$ must have the type `boolean`. The temporal expression *PExp* in the expression $\mathbf{E}[\mathit{PExp}]$ must be of type `real`. We also assume that expressions of type `integer` can be coerced to the `real` type.

4.3 Semantics of QuaTEx

Next, we give the semantics of a subset of query expressions that can be written in QuaTEx. In this subclass, we impose the restriction that the value of a path expression *PExp* that appears in any expression $\mathbf{E}[\mathit{PExp}]$ can be determined from a finite prefix of an execution path. We call such temporal expressions *bounded* path expressions. The semantics is given in Fig. 6. $(\pi)[\![\mathit{PExp}]\!]_D$ is the value of the path expression *PExp* over the path π. Similarly, $(s)[\![\mathit{SExp}]\!]_D$ is the value of the state expression *SExp* in the state s. Note that if the value of a bounded path expression can be computed from a finite

$$(s)[\![c]\!]_D = c$$

$$(s)[\![f]\!]_D = f(s)$$

$$(s)[\![F(SExp_1,\ldots,SExp_k)]\!]_D = F((s)[\![SExp_1]\!]_D,\ldots,(s)[\![SExp_k]\!]_D)$$

$$(s)[\![\mathbf{E}[PExp]]\!]_D = \mathbf{E}[(\pi)[\![PExp]\!]_D] \text{ for } \pi \in Paths(s)$$

$$(\pi)[\![SExp]\!]_D = (\pi[0])[\![SExp]\!]_D$$

$$(\pi)[\![\underline{\texttt{if}}\ SExp\ \underline{\texttt{then}}\ PExp_1\ \underline{\texttt{else}}\ PExp_2\ \underline{\texttt{fi}}]\!]_D =$$

$$\text{if } (\pi[0])[\![SExp]\!]_D = \text{true then } (\pi)[\![PExp_1]\!]_D \text{ else } (\pi)[\![PExp_2]\!]_D$$

$$(\pi)[\![\bigcirc N(SExp_1,\ldots,SExp_m)]\!]_D =$$

$$(\pi^{(1)})[\![B[x_1 \mapsto (\pi[0])[\![SExp_1]\!]_D,\ldots,x_m \mapsto (\pi[0])[\![SExp_m]\!]_D]]\!]_D$$

$$\text{where } N(x_1,\ldots,x_m) = B; \in D$$

Fig. 6. Semantics of QuaTEx

prefix π_{fin} of an execution path π, then the evaluations of the path expression over all execution paths having the common prefix π_{fin} are the same. Since a finite prefix of a path defines a basic cylinder set (i.e. a set containing all paths having the common prefix) having an associated probability measure, we can compute the expected value of a bounded path expression over a random path from a given state. In our analysis tool, we statistically estimate the expected value through Monte-Carlo simulation instead of calculating it exactly based on the underlying probability distributions of the model. The exact procedure is described in Section 4.5.

4.4 Example Encoding of Standard Temporal Operators

In probabilistic computation tree logic (PCTL) [15] and continuous stochastic logic (CSL) [1,6] a compound temporal logic operator is the *until* operator $\mathcal{U}$. A path satisfies $\phi_1 \mathcal{U} \phi_2$, iff some state s along the path satisfies ϕ_2 and all states before s along the path satisfies ϕ_1. We can easily encode the until operator as follows:

$$\texttt{Until}(\phi_1,\phi_2) = \underline{\texttt{if}}\ \phi_2\ \underline{\texttt{then}}\ 1\ \underline{\texttt{else}}\ \underline{\texttt{if}}\ \phi_1\ \underline{\texttt{then}}\ \bigcirc(\texttt{Until}(\phi_1,\phi_2))\ \underline{\texttt{else}}\ 0\ \underline{\texttt{fi}}\ \underline{\texttt{fi}};$$

The operator takes as arguments two state expressions ϕ_1 and ϕ_2 of type Boolean. It returns 1 if $\phi_1 \mathcal{U} \Phi_2$ holds along the path and 0 otherwise. We return 1 and 0 instead of *true* or *false* because we may want to calculate the probability that a path from a given state satisfies $\phi_1 \mathcal{U} \phi_2$, i.e., $Pr[\phi_1 \mathcal{U} \phi_2]$. For example the following QuaTEx expression

$$\texttt{Until}(\phi_1, \phi_2) = \underline{\texttt{if}}\ \phi_2\ \underline{\texttt{then}}\ 1\ \underline{\texttt{else}}\ \underline{\texttt{if}}\ \phi_1\ \underline{\texttt{then}}\ \bigcirc\,(\texttt{Until}(\phi_1, \phi_2))\ \underline{\texttt{else}}\ 0\ \underline{\texttt{fi}}\ \underline{\texttt{fi}};$$

$$\underline{\texttt{eval}}\ \mathbf{E}[\texttt{Until}(\neg \mathit{resend}(), \mathit{receive}())]$$

queries the probability that a message is received without re-sending. The state predicates (or state expressions of type Boolean) *resend()* and *receive()* returns *true* iff a message is re-sent and received in the current state, respectively. Note that $\neg$ is a unary function with the usual meaning mapping a state expression to another state expression.

Similarly, we can encode the bounded until operator $\phi_1\, \mathcal{U}^{\leq t} \phi_2$ of CSL as follows:

$$\begin{aligned} \texttt{UntilBounded}(\phi_1, \phi_2, t) = {} & \underline{\texttt{if}}\ t > \mathit{time}()\ \underline{\texttt{then}}\ 0\ \underline{\texttt{else}}\ \underline{\texttt{if}}\ \phi_2\ \underline{\texttt{then}}\ 1\ \underline{\texttt{else}} \\ & \underline{\texttt{if}}\ \phi_1\ \underline{\texttt{then}}\ \bigcirc\,(\texttt{UntilBounded}(\phi_1, \phi_2, t))\ \underline{\texttt{else}}\ 0\ \underline{\texttt{fi}}\ \underline{\texttt{fi}}\ \underline{\texttt{fi}}; \end{aligned}$$

where the state function *time*() returns the global time associated with the state.

However, QuaTEx is more expressive than the temporal logic operators of PCTL and CSL. It can be used for counting as described through an example in Section 4.1. It can be used to relate data temporally. For example, suppose we want to know *"the probability that along a random path from a given state, if a message is sent then the same message is received within* 100 *time units."* This can be written as the following query:

$$\begin{aligned} \texttt{Received}(m, t) = {} & \underline{\texttt{if}}\ t > \mathit{time}()\ \underline{\texttt{then}}\ 0\ \underline{\texttt{else}}\ \underline{\texttt{if}}\ \mathit{receive}(m)\ \underline{\texttt{then}}\ 1 \\ & \underline{\texttt{else}}\ \bigcirc\,(\texttt{Received}(m, t))\ \underline{\texttt{fi}}\ \underline{\texttt{fi}}; \\ \underline{\texttt{eval}}\ \mathbf{E}[& \underline{\texttt{if}}\ \mathit{send}()\ \underline{\texttt{then}}\ \texttt{Received}(\mathit{messageId}(), 100 + \mathit{time}())\ \underline{\texttt{else}}\ 1\ \underline{\texttt{fi}}] \end{aligned}$$

where, the state function *time*() returns the global time associated with the state; the state function *send*() returns true, iff in the state a message is sent; *receive*(m) returns true, iff in the state a message with id m is received; *messageId*(m) returns the id of the message that is sent in the current state. Note that along a path, the path expression $\underline{\texttt{if}}\ \mathit{send}()\ \underline{\texttt{then}}\ \texttt{Received}(\mathit{messageId}(), 100 + \mathit{time}())\ \underline{\texttt{else}}\ 1\ \underline{\texttt{fi}}$ returns 1 if a message is sent in the current state and the same message (i.e. the message having the same id) is received within 100 time units at some later state. Here the data, message id, is related temporally which is otherwise not possible using the traditional probabilistic temporal logics.

4.5 Statistical Evaluation of a QUATEX *Expression*

Given a probabilistic model and a QUATEX expression, we evaluate the expression at the initial state of the model. The evaluation of all path and state expressions, except the expectation expression, is straightforward and follows directly from the semantics. However, the evaluation of an expression of the form $\mathbf{E}[PExp]$ can be difficult to compute numerically for a complex probabilistic model. Rather, the expected value of a QUATEX expression is statistically evaluated with respect to two parameters α and δ provided as input. Specifically, we approximate the expected value by the mean of n samples such that the size of $(1-\alpha)100\%$ confidence interval [18] for the expected value computed from the samples is bounded by δ. We next describe the details of this computation.

Let X be random variable giving the value of the expression $PExp$ along a random path π from a state s. Then $(s)[\![\mathbf{E}[PExp]]\!]_D = \mathbf{E}[X]$. Let $X_1, \ldots, X_n$ be n random variables having the same distribution as X. By Central Limit Theorem [18], we know that if

$$Z = \frac{\bar{X} - \mu}{S/\sqrt{n}}$$

where $\bar{X} = \frac{\sum_{i\in[1,n]} X_i}{n}$, $S^2 = \frac{\sum_{i\in[1,n]} X_i^2 - \bar{X}^2}{n-1}$, and $\mu = \mathbf{E}[X]$, then Z has student's t-distribution with $n-1$ degrees of freedom for large enough n. If T is random variable having t-distribution with k degrees of freedom, then we define $t_{\alpha,k}$ as follows:

$$Pr[T < t_{\alpha,k}] = 1 - \alpha$$

The values of $t_{\alpha,k}$ for various values of α and k can be obtained from a distribution table or by numerical computation.

Let $x_1, \ldots, x_n$ be n samples from X. Then for large enough n (i.e. $n > 30$) a $(1-\alpha)100\%$ confidence interval is given by

$$(\bar{x} - t_{\alpha/2,n-1}\frac{s}{\sqrt{n}}, \bar{x} + t_{\alpha/2,n-1}\frac{s}{\sqrt{n}})$$

where $\bar{x} = \frac{\sum_{i\in[1,n]} x_i}{n}$, $s^2 = \frac{\sum_{i\in[1,n]} x_i^2 - \bar{x}^2}{n-1}$. We want the size of this $(1-\alpha)100\%$ confidence interval to be less than or equal to δ. That is we want

$$2t_{\alpha/2,n-1}\frac{s}{\sqrt{n}} \leq \delta$$

We compute $\mathbf{E}[X]$ iteratively using the function *computeExpectedValueOfX()* described below. The function iteratively tries to find a sample size n, such that a $(1-\alpha)100\%$ confidence interval computed from n samples has a size less than or equal to δ. The mean of these n sample, i.e. $\bar{x}$, is then returned as the estimated value for $\mathbf{E}[X]$.

computeExpectedValueOfX()

Input: α, δ, X the random variable
Output: Approximate $\mathbf{E}[X]$

```
begin
   n = 0;
   while(d > δ)
   begin
      n = n + 100;
      Let x_1, ..., x_n be n samples of X
      d = 2 t_{α/2,n-1} s/√n
   end
   return x̄
end
```

4.6 Implementation

We have implemented the evaluator for QuaTEx in Java. The tool, called VeStA, takes as input an actor PMaude model, an initial actor PMaude term representing the initial configuration of the system, and a QuaTEx expression along with the two parameters α and δ.

VeStA performs discrete-event simulation by invoking the Maude interpreter[12]. VeStA maintains the current configuration of the system as an actor PMaude term represented as a Java string. This term is initialized to the initial actor PMaude term provided as input. At every simulation step, VeStA passes the current configuration term to the Maude interpreter for a one-step computation and obtains the result of rewriting as a term representing the next configuration. The value of the application of a function on the current state, as required by certain QuaTEx expressions, is computed by VeStA by parsing the current configuration term.

5 Conclusion

We have introduced PMaude, a rewrite-based formal modelling language for probabilistic concurrent systems with support for discrete-event simulation and statistical analysis. One important advantage of PMaude is that the well-known expressiveness of rewrite rules to specify concurrent systems [25] is in this way naturally extended to specify concurrent probabilistic systems. In fact, a PMaude specification may have both probabilistic rewrite rules and ordinary rewrite rules, which can be viewed as a no-probability special case of probabilistic rules. The language allows high-level specification of a wide-range of probabilistic systems. In particular, it supports concurrent object-oriented programming through actors. PMaude specifications can be simulated in the underlying Maude language. We have also introduced QuaTEx, a language to specify quantitative temporal expressions that can be used to query various quantitative aspects of a probabilistic model. We have already used PMaude and VeStA to model and analyze a DoS resistant TCP/IP protocol [3]. We plan to use the tool to model and analyze various other network protocols.

Acknowledgement

The authors would specially like to acknowledge Nirman Kumar for his contribution to the development of an earlier finitary version of PMaude. The work is supported in part by the ONR Grant N00014-02-1-0715.

References

[1] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time Markov chains. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102, pages 269–276. Springer, 1996.

[2] G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.

[3] G. Agha, C. Gunter, M. Greenwald, S. Khanna, J. Meseguer, K. Sen, and P. Thati. Formal modeling and analysis of dos using probabilistic rewrite theories. In *Workshop on Foundations of Computer Security (FCS'05) (Affiliated with LICS'05)*, 2005.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[5] M. Astley and G. A. Agha. Customization and composition of distributed objects: middleware abstractions for policy management. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–9, 1998.

[6] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous-time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[7] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer-Verlag, January 2004.

[8] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proceedings of 15th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)*, volume 1026 of *LNCS*.

[9] H. C. Bohnenkamp, H. Hermanns, J.-P. Katoen, and R. Klaren. The modest modeling tool and its implementation. In *13th International Conference on Computer Performance Evaluations, Modelling Techniques and Tools*, volume 2794 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2003.

[10] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1–2):35–132, 2000.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual, Version 1.0*, june 2003. http://maude.cs.uiuc.edu/maude2-manual/.

[13] P. D'Argenio. *Algebras and automata for timed and stochastic systems*. PhD thesis, University of Twente, Enschede, The Netherlands, 1999.

[14] P. W. Glynn. On the role of generalized semi-markov processes in simulation output analysis. In *WSC '83: Proceedings of the 15th IEEE conference on Winter simulation*, pages 39–44, 1983.

[15] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[16] H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. *Theoretical Compututer Science*, 274(1-2):43–87, 2002.

[17] J. Hillston. *A Compositional Approach to Performance Modelling*. Distinguished Dissertations Series. Cambridge University Press, 1996.

[18] R. V. Hogg and A. T. Craig. *Introduction to Mathematical Statistics*. Macmillan, New York, NY, USA, fourth edition, 1978.

[19] O. C. Ibe and K. S. Trivedi. Stochastic petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, Dec. 1990.

[20] N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical Report UIUCDCS-R-2003-2347, University of Illinois at Urbana-Champaign, May 2003.

[21] N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model for probabilistic distributed object systems. In *Proceedings of 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'03)*, volume 2884 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2003.

[22] M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker, 2002.

[23] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley and Sons, 1995.

[24] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[25] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[26] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.

[27] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[28] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[29] R. Segala. *Modelling and Verification of Randomized Distributed Real Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.

[30] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *16th conference on Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 202–215. Springer, July 2004.

[31] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of probabilistic systems. In *17th conference on Computer Aided Verification (CAV'05)*, LNCS (To Appear). Springer, July 2005.

[32] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton, 1994.

[33] D. C. Sturman and G. Agha. A protocol description language for customizing semantics. In *Symposium on Reliable Distributed Systems*, pages 148–157, 1994.