A Probabilistic Strategy Language for Probabilistic Rewrite Theories and Its Application to Cloud Computing

Lucian Bentea, Peter Csaba Ölveczky

A Probabilistic Strategy Language for Probabilistic Rewrite Theories and its Application to Cloud Computing

Lucian Bentea[1] and Peter Csaba Ölveczky[1,2]

[1] University of Oslo
[2] University of Illinois at Urbana-Champaign

Abstract. Several formal models combine probabilistic and nondeterministic features. To allow their probabilistic simulation and statistical model checking by means of pseudo-random number sampling, all sources of nondeterminism must first be quantified. However, current tools offer limited flexibility for the user to define how the nondeterminism should be quantified. In this paper, we propose an expressive *probabilistic strategy language* that allows the user to define complex strategies for quantifying the nondeterminism in *probabilistic rewrite theories*. We have implemented PSMaude, a tool that extends Maude with a probabilistic simulator and a statistical model checker for our language. We illustrate the convenience of being able to define different probabilistic strategies on top of a system by a cloud computing example, where different load balancing policies can be specified by different probabilistic strategies. We then use PSMaude to analyze the QoS provided by different policies.

1 Introduction

Many formal analysis tools support the modeling of systems that exhibit both probabilistic and nondeterministic behaviors. To allow their probabilistic simulation and statistical model checking using pseudo-random number sampling, the nondeterminism must be quantified to obtain a fully probabilistic model. However, there is typically limited support for user-definable adversaries to quantify the nondeterminism in reasonably expressive models; such adversaries are either added by the tool or must be encoded directly into the system model.

In this paper we propose an expressive *probabilistic strategy language* for *probabilistic rewrite theories* [18, 1] that allows users to define complex adversaries for a model, and therefore allows us to separate the definition of the system model from that of the adversary needed to quantify the nondeterminism in the system model.

Rewriting logic is a simple and expressive logic for concurrent systems in which the data types are defined by an algebraic equational specification and where the local transition patterns are defined by conditional labeled rewrite rules of the form $l : t \longrightarrow t'$ **if** *cond*, where l is a label and t and t' are terms representing state fragments. Maude [11] is a high-performance simulation, reacha-

bility, and LTL model checking tool for rewriting logic that has been successfully applied to many large applications (see, e.g., [25, 22] for an overview).

Rewriting logic has been extended to *probabilistic rewrite theories* [18, 1], where probabilities are introduced by new variables in the righthand side t' of a rewrite rule. These variables are instantiated according to a probability distribution associated with the rewrite rule. Probabilistic rewrite theories, together with the VeStA statistical model checker [27], have been applied to analyze sensor network algorithms [17] and defense mechanisms against denial of service attacks [3, 13]. However, since the probabilistic rewrite theories were highly nondeterministic, adversaries had to be encoded into the model before any analysis could take place.

Probabilistic model checking suffers from state space explosion which renders it unfeasible for automated analysis of the complex concurrent systems targeted by rewriting logic. *Statistical model checking* [20, 28, 26] trades absolute confidence in the correctness of the model checking for computational efficiency, and essentially consists of simulating a number of different system behaviors until a certain confidence level is reached. This not only makes statistical analysis feasible, but also makes such model checking amenable to parallelization, which is exploited in the parallel version PVeStA [2] of the statistical model checker VeStA. PVeStA has recently been used to analyze an adaptive system specified as a hierarchical probabilistic rewrite theory [10].

To support the analysis of probabilistic rewrite theories where our strategy language has been used to quantify all nondeterminism, we have formalized and integrated into (Full) Maude both probabilistic simulation and statistical model checking. Our strategy language and its implementation, the PSMaude tool [4], enable a Maude-based safety/QoS modeling and analysis methodology in which:

1. A non-probabilistic rewrite theory defines all possible behaviors in a simple "uncluttered" way; this model can then be directly subjected to important safety analyses to guarantee the absence of bad behaviors.
2. Different QoS policies and/or probabilistic environments can then be defined as probabilistic strategies on top of the basic verified model for QoS reasoning by probabilistic simulation and statistical model checking.

We exemplify in Section 4 the usefulness of this methodology and of the possibility to define different complex probabilistic strategies on top of the same model with a cloud computing example, where a (non-probabilistic) rewrite theory defines all the possible ways in which requested resources can be allocated on servers in the cloud, as well as all possible environment behaviors. We can then use standard techniques to prove safety properties of this model. However, one could imagine a number of different *policies* for assigning resources to service providers and users, such as, e.g.,

– Service providers might request virtual machines uniformly across different regions (for fault-tolerance and omnipresence), or with higher probability at certain locations, or with higher probability at more stable servers.

– Service users may be assigned virtual machines either closer to their locations, on physical servers with low workload, or on reliable servers, with high probability.

Each load balancing policy can be naturally specified as a probabilistic strategy on top of the (non-probabilistic) model of the cloud infrastructure that has been proved to be "safe." We then use PSMaude to perform simulation and statistical model checking to analyze the QoS effect of the different load balancing policies.

2 Preliminaries

Rewriting Logic and Maude. A *rewrite theory* [24] is a tuple $\mathcal{R} = (\Sigma, E \cup A, L, R)$, where $(\Sigma, E \cup A)$ is a membership equational logic theory, with E a set of equations $(\forall \vec{x})\ t = t'$ **if** *cond*, and membership axioms $(\forall \vec{x})\ t : s$ **if** *cond*, where t and t' are Σ-terms, s is a sort, and *cond* is a conjunction of equalities and sort memberships, and with A a collection of *structural axioms* specifying properties of operators, like commutativity, associativity, etc., R is a set of rewrite rules $(\forall \vec{x})\ l : t \longrightarrow t'$ **if** *cond*, where $l \in L$ is a label, t and t' are terms of the same kind, *cond* is a conjunction of equalities, memberships and rewrites, and $\vec{x} = vars(t) \cup vars(t') \cup vars(cond)$. Such a rule specifies a transition from an instance of the term t to the corresponding instance of t', provided that *cond* is satisfied. $vars(t)$ denotes the set of variables in a term t; if $vars(t) = \emptyset$, then t is a *ground term*. If E is terminating, confluent and sort-decreasing modulo A, then $Can_{\Sigma,E/A}$ denotes the algebra of fully simplified ground terms, and we denote by $[t]_A$ the A-equivalence class of a term t. An *E/A-canonical ground substitution* for a set of variables $\vec{x}$ is a function $[\theta]_A : \vec{x} \to Can_{\Sigma,E/A}$; we denote by $CanGSubst_{E/A}(\vec{x})$ the set of all such functions. We also denote by $[\theta]_A$ the homomorphic extension of $[\theta]_A$ to Σ-terms. A *context* is a Σ-term with a single *hole* variable $\odot$; two contexts $\mathbb{C}$ and $\mathbb{C}'$ are A-equivalent if $A \vdash (\forall \odot)\ \mathbb{C}(\odot) = \mathbb{C}'(\odot)$. Given $[u]_A \in Can_{\Sigma,E/A}$, its *$R/A$-matches* are triples $([\mathbb{C}]_A, r, [\theta]_A)$ where $\mathbb{C}$ is a context, $r \in R$ is a rewrite rule, $[\theta]_A \in CanGSubst_{E/A}(\vec{x})$ is such that $E \cup A \vdash \theta(cond)$, and $[u]_A = [\mathbb{C}(\odot \leftarrow \theta(t))]_A$. We denote by $\mathcal{M}([u]_A)$ the set of all such triples, and define the set of rules that are *enabled* for a term $[u]_A$, the set of *valid contexts* for $[u]_A$ and a rule r, and the set of *valid substitutions* for $[u]_A$, a rule r, and a context $[\mathbb{C}]_A$, in the expected way:

$$\begin{aligned}
\texttt{enabled}([u]_A) &= \{r \in R \mid \exists [\mathbb{C}]_A, \exists [\theta]_A :\ ([\mathbb{C}]_A, r, [\theta]_A) \in \mathcal{M}([u]_A)\} \\
C([u]_A, r) &= \{[\mathbb{C}]_A \in Can_{\Sigma,E/A}(\odot) \mid \exists [\theta]_A :\ ([\mathbb{C}]_A, r, [\theta]_A) \in \mathcal{M}([u]_A)\} \\
S([u]_A, r, [\mathbb{C}]_A) &= \{[\theta]_A \in CanGSubst_{E/A}(\vec{x}) \mid ([\mathbb{C}]_A, r, [\theta]_A) \in \mathcal{M}([u]_A)\}
\end{aligned}$$

Maude [11] is a high-performance simulation, reachability analysis, and LTL model checking tool for rewrite theories. We use Maude syntax, so that conditional rules are written `crl [l]: t => t' if` *cond*. In object-oriented Maude specifications [11], the system state is a term of sort `Configuration` denoting a multiset of objects and messages, with multiset union denoted by juxtaposition.

A class declaration `class` C `|` att_1 `:` s_1`,` $\ldots$`,` att_n `:` s_n declares a class C with attributes $att_1, \ldots, att_n$ of sorts $s_1, \ldots, s_n$, respectively. A *subclass* inherits the attributes and rules of its superclass(es). Objects are represented as terms `<` o `:` C `|` att_1 `:` val_1`,` $\ldots$`,` att_n `:` val_n `>`, where o is the object's identifier of sort `Oid`, C is the object's class, and where $val_1, \ldots, val_n$ are the values of the object's attributes $att_1, \ldots, att_n$. For example, the rule

```
rl [l]: m(O, w)  < O : C | a1 : x, a2 : O', a3 : z >   =>
                 < O : C | a1 : x + w, a2 : O', a3 : z >  m'(O', x) .
```

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions change the attribute `a1` of `O` and send a new message `m'(O',x)`. "Irrelevant" attributes (such as `a3` and the righthand side occurrence of `a2`) need not be mentioned.

Markov Chains. Given $\Omega \neq \emptyset$, a *σ-algebra* over Ω is a collection $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ such that $\Omega \setminus F \in \mathcal{F}$ for all $F \in \mathcal{F}$, and $\bigcup_{i \in I} F_i \in \mathcal{F}$ for all collections $\{F_i\}_{i \in I} \subseteq \mathcal{F}$ indexed by a countable set I. Given a σ-algebra $\mathcal{F}$ over Ω, a function $\mathbb{P} : \mathcal{F} \to [0,1]$ is a *probability measure* if $\mathbb{P}(\Omega) = 1$ and $\mathbb{P}\left(\cup_{i \in I} F_i\right) = \sum_{i \in I} \mathbb{P}(F_i)$, for all collections $\{F_i\}_{i \in I} \subseteq \mathcal{F}$ of pairwise disjoint sets. We denote by $PMeas(\Omega, \mathcal{F})$ the set of all probability measures on $\mathcal{F}$ over Ω. A *probability mass function* (pmf) is a function $p : \Omega \to [0,1]$ with $\sum_{\omega \in \Omega} p(\omega) = 1$. A family of pmf's can be used to define the behavior of a (memoryless) probabilistic system. In particular, a *discrete time Markov chain* (DTMC) is given by a countable set of *states* S, and a *transition probability* matrix $T : S \times S \to [0,1]$, where $T(s) : S \to [0,1]$ is a pmf for all states $s \in S$, i.e., $T(s, s')$ is the probability for the DTMC to make a transition from state s to state s'.

Probabilistic Rewrite Theories. In *probabilistic rewrite theories* (PRTs) [18] the righthand side t' of a rule $l : t \longrightarrow t'$ **if** *cond* may contain variables $\vec{y}$ that do not occur in t, and that are instantiated according to a probability measure taken from a *family* of probability measures—one for each instance of the variables in t—associated with the rule. Formally, a PRT $\mathcal{R}_\pi$ is a pair $(\mathcal{R}, \pi)$, where $\mathcal{R}$ is a rewrite theory, and π maps each rule r of $\mathcal{R}$, with $vars(t) = \vec{x}$ and $vars(t') \setminus vars(t) = \vec{y}$, to a mapping $\pi_r : [\![cond(\vec{x})]\!] \to PMeas\left(CanGSubst_{E/A}(\vec{y}), \mathcal{F}_r\right)$, where $[\![cond(\vec{x})]\!] = \{\, [\theta]_A \in CanGSubst_{E/A}(\vec{x}) \mid E \cup A \vdash \theta(cond) \,\}$, and $\mathcal{F}_r$ is a σ-algebra over $CanGSubst_{E/A}(\vec{y})$. That is, for each substitution $[\theta]_A$ of the variables in t that satisfies *cond*, we get a probability measure $\pi_r\left([\theta]_A\right)$ for instantiating the variables $\vec{y}$. The rule r together with π_r is called a *probabilistic rewrite rule*, and is written $l : \ t \longrightarrow t'$ **if** *cond* **with probability** π_r. We refer to the specification of the "blackboard game" in Section 3 for an example of the syntax used to specify probabilistic rewrite rules and the probability measure π_r. An *E/A-canonical one-step rewrite* of $\mathcal{R}_\pi$ [18, 1] is a labeled transition $[u]_A \xrightarrow{([\mathbb{C}]_A,\, r,\, [\theta]_A,[\rho]_A)} [v]_A$ with $m \triangleq ([\mathbb{C}]_A,\, r,\, [\theta]_A)$ a R/A-match for $[u]_A$, $[\rho]_A \in CanGSubst_{E/AS}(\vec{y})$, and $[v]_A = [\mathbb{C}(\odot \leftarrow (\theta \cup \rho)(t'))]_A$. To quantify the nondeterminism in the choice of m, the notion of *adversary* is introduced in [18, 1] that samples m from a pmf that depends on the computation history.

A *memoryless* adversary[3] is a family of pmf's $\{\sigma_{[u]_A} : \mathcal{M}([u]_A) \to [0,1]\}_{[u]_A}$, where $\sigma_{[u]_A}(m)$ is the probability of picking the R/A-match m. A consequence of a result in [18] is that executing $\mathcal{R}_\pi$ under $\{\sigma_{[u]_A}\}_{[u]_A}$ is described by a DTMC.

PCTL. The *probabilistic computation tree logic* (PCTL) [16] extends CTL with an operator $\mathcal{P}$ to express properties of DTMCs. We use a subset of PCTL, without time-bounded and steady-state operators. If AP is a set of atomic propositions, ϕ is a *state* formula, and ψ is a *path* formula, PCTL formulas over AP are defined by:

$$\phi ::= \mathit{true} \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathcal{P}_{\bowtie p}(\psi) \qquad \psi ::= \phi\, \mathcal{U}\, \phi \mid \mathbf{X}\phi$$

where $a \in AP$, $\bowtie \in \{<, \leq, >, \geq\}$, and $p \in [0,1]$. PCTL satisfaction is defined over DTMCs, e.g., the meaning of $\mathcal{M}, s \models \mathcal{P}_{<0.05}(\mathit{true}\ \mathcal{U}\ \phi)$ is that ϕ eventually becomes true in less than 5% of all the runs of the DTMC $\mathcal{M}$ from state s.

Statistical Model Checking and VeStA. Traditional model checking suffers from state space explosion problem, whereas *statistical model checking* [20, 28, 26] trades complete confidence for efficiency, allowing the analysis of large-scale probabilistic systems. This technique is based on simulating the model, and on performing statistical hypothesis testing to control the generation of execution traces. The simulation is stopped when a given *level of confidence* is reached for answering the model checking problem. The VeStA tool [27] supports statistical model checking and quantitative analysis of executable specifications in which all nondeterminism is quantified probabilistically. In VeStA, system properties are given in PCTL (or its continuous-time extension CSL), while quantitative analysis queries are given in the QuaTEx logic [1] and ask for the average values of quantities associated with the model—VeStA simulates the model and provides estimates for these averages.

3 A Language for Specifying Memoryless Adversaries

The source of nondeterminism in a probabilistic rewrite theory is picking an R/A-match from a set of possible ones in each state. This section introduces a probabilistic strategy language that can be used to quantify this nondeterminism, i.e., for specifying memoryless adversaries of PRTs.

The probability distribution associated with picking a certain R/A-match $([\mathbb{C}]_A, r, [\theta]_A)$ can be specified using the individual (conditional) distributions for picking the rule r, the context $[\mathbb{C}]_A$, and the substitution $[\theta]_A$, which is more convenient than specifying their joint distribution. That is, by probability theory, any memoryless adversary $\{\sigma_{[u]_A}\}_{[u]_A}$ of a PRT $\mathcal{R}_\pi$ can be decomposed as:

$$\begin{aligned}&\sigma_{[u]_A}([\mathbb{C}]_A, r, [\theta]_A) = \\ &\mathbb{P}\{\text{pick rule } r \mid \text{state is } [u]_A\} \cdot \mathbb{P}\{\text{pick context } [\mathbb{C}]_A \mid \text{state is } [u]_A, \text{ rule is } r\} \\ &\qquad \cdot \mathbb{P}\{\text{pick substitution } [\theta]_A \mid \text{state is } [u]_A, \text{ rule is } r, \text{ context is } [\mathbb{C}]_A\}\end{aligned}$$

[3] This is a slightly modified version of the definition of adversaries in [18, 1].

We denote the factors in this product by $\mathcal{A}_{[u]_A}(r)$, $\mathcal{A}^r_{[u]_A}([\mathbb{C}]_A)$, and $\mathcal{A}^{r,[\mathbb{C}]_A}_{[u]_A}([\theta]_A)$, respectively. They give the probabilities of picking rule r in state $[u]_A$, of using context $[\mathbb{C}]_A$ to match $[u]_A$ with the lefthand side of r, and of using substitution $[\theta]_A$ to match $[u]_A$ with the lefthand side of r in context $[\mathbb{C}]_A$, respectively. We call $\{\mathcal{A}_{[u]_A} : \texttt{enabled}([u]_A) \to [0,1]\}_{[u]_A}$, $\{\mathcal{A}^r_{[u]_A} : C([u]_A, r) \to [0,1]\}_{[u]_A,r}$, and $\{\mathcal{A}^{r,[\mathbb{C}]_A}_{[u]_A}([\theta]_A) : S([u]_A, r, [\mathbb{C}]_A) \to [0,1]\}_{[u]_A,r,[\mathbb{C}]_A}$, resp., the underlying *rule*, *context*, and *substitution adversaries* of the memoryless adversary $\{\sigma_{[u]_A}\}_{[u]_A}$.

It is cumbersome to define *absolute* probabilities for each choice (so that they add up to 1 in each state). If we have rules r_1, r_2, and r_3, and want r_1 to be applied with 3 times as high probability as r_2 (when both are enabled), which should be twice as likely as taking rule r_3, then, for a state $[u]_A$ where all rules are enabled, the probabilities would be $\{r_1 \mapsto 6/9,\ r_2 \mapsto 2/9,\ r_3 \mapsto 1/9\}$, and for a state $[u']_A$ where r_2 is not enabled, the distribution would be $\{r_1 \mapsto 6/7,\ r_3 \mapsto 1/7\}$, etc. This can soon become inconvenient. In our language one therefore instead defines *relative* weights for each rule, context, and substitution. That is, for any state $[u]_A$ in our example, the "weights" of the rules r_1, r_2 and r_3 could be 6, 2, and 1, respectively. Relative weights are therefore needed since the set of possible R/A-matches in a state, whose nondeterministic choice we want to quantify, is only available during the model execution. We therefore build the concrete probability distributions *on-the-fly* in each state during execution, which we sample to obtain the next state.

Language. The language we propose allows specifying memoryless rule, context, and substitution adversaries, using *strategy expressions* of the following forms:

psdrule ⟨*Identifier*⟩ := **given state:** ⟨*StatePattern*⟩
is: ⟨*RuleWeightDist*⟩ [**if** ⟨*Condition*⟩] [[**owise**]]

psdcontext ⟨*Identifier*⟩ := **given state:** ⟨*StatePattern*⟩
rule: ⟨*RulePattern*⟩
is: ⟨*ContextWeightDist*⟩ [**if** ⟨*Cond*⟩] [[**owise**]]

psdsubst ⟨*Identifier*⟩ := **given state:** ⟨*StatePattern*⟩
rule: ⟨*RulePattern*⟩
context: ⟨*ContextPattern*⟩
is: ⟨*SubstWeightDist*⟩ [**if** ⟨*Condition*⟩] [[**owise**]]

with ⟨*StatePattern*⟩ a term $t(\vec{x})$, ⟨*RulePattern*⟩ a rule label or a variable over rule labels, and ⟨*ContextPattern*⟩ a term $c(\vec{y})$ with $\odot \in \vec{y}$ and $\vec{y} \backslash \{\odot\} \subseteq \vec{x}$, or a variable over contexts. The weight expression ⟨*RuleWeightDist*⟩ is either **uniform**, or a list of "**;**"-separated weight assignments of the form ⟨*RuleLabel*⟩ ↦ ⟨*Weight*⟩, where ⟨*Weight*⟩ is a term $w(\vec{x})$ of sort `Rat`; ⟨*ContextWeightDist*⟩ and ⟨*SubstWeightDist*⟩ have similar forms. ⟨*Condition*⟩ (abbreviated to ⟨*Cond*⟩ in the context adversary syntax above) specifies the condition under which the strategy can be applied, and **[owise]** specifies that it should be applied if no other strategy can be applied. We refer to [5] for the detailed syntax and semantics of our language, i.e., how each strategy expression defines an adversary. Our implementation also provides an executable rewriting logic semantics of our language.

PSMaude. Our tool PSMaude [4] extends Maude by adding support for specifying probabilistic rules with fixed-size probability distributions and our strategy language, a probabilistic rewrite command and a statistical PCTL model checker that can analyze a given PRT controlled by given probabilistic strategies.

Given a probabilistic module *SYSTEM-SPEC* that specifies a PRT, probabilistic strategies can be written in modules of the form:

```
(psmod PSTRAT is protecting SYSTEM-SPEC .          --- import system specification
  state StateSort .                                --- sort for system states
  psdrule    RuleStratID    := RuleStratExpr .     --- rule strategy
  psdcontext ContextStratID := ContextStratExpr .  --- context strategy
  psdsubst   SubstStratID   := SubstStratExpr .    --- substitution strategy
  psd StratID := < RuleStratID | ContextStratID | SubstStratID > . --- strategy
endpsm)
```

A strategy definition, introduced with `psd`, associates strategies for rules, contexts, and substitutions with a strategy identifier *StratID*, that can be used in a *probabilistic strategy rewrite* command `(prew [`n`] `s` using StratID .)`, which executes n one-step (probabilistic) rewrites from the state s using the strategy *StratID*. The unbounded version `(uprew `s` using StratID .)` rewrites until a deadlock occurs. The strategies for rules, contexts, and substitutions are introduced with `psdrule`, `psdcontext`, and `psdsubst`, respectively; they can be conditional, with keywords `cpsdrule`, `cpsdcontext`, and `cpsdsubst`, respectively. There may be several definitions for the same strategy identifier, but they should refer to disjoint cases of the arguments. `[owise]`-annotated strategy expressions can be used to specify how the nondeterminism is resolved when no other strategy definition is applicable. It can thus be easily ensured that a probabilistic strategy resolves *all* nondeterminism in a given system specification.

Our statistical model checking command `(smc `s` |= `φ` using StratID .)` allows for further analysis of a specification with given strategies, where φ is a PCTL formula, and satisfaction of atomic propositions is defined in a *state predicate module*:

```
(spmod SYSTEM-PRED is protecting SYSTEM-SPEC . --- import system specification
  smcstate StateSort .                         --- sort for system states
  psp φ1 ... φn : Sort1 ... SortK .            --- parametric state predicates
  var S : StateSort .
  csat S |= φ1(s1, ..., sK) if f(S, (s1, ..., sK)) . --- define their semantics
  ...
endspm)
```

For instance, the parametric state predicate $\varphi_1(s_1, \ldots, s_K)$ holds in a state `S` if and only if the condition $f(\mathtt{S}, s_1, \ldots, s_K)$ in the above `csat` declaration is `true`, where the operator f is defined by means of (possibly conditional) equations.

The command `(set type1 error `b_1` .)` sets the bound on type I errors (the algorithm returns "false" when the property holds), and `(set type2 error `b_2` .)` sets the bound on type II errors (vice versa). By lowering these bounds, a higher confidence on the model checking result is achieved, but more execution samples are generated.

Example. In each step in a probabilistic version of the *blackboard game* [23], in which some numbers are on a blackboard, two arbitrary numbers x and y are replaced by $(x^2 + y)$ `quo` 2 with probability 3/4, and by $(x^3 + y)$ `quo` 2 with probability 1/4, where `quo` denotes integer division. The goal of the game is to obtain the highest possible number at the end. This game is formalized in the following probabilistic module, that also defines an initial state:

```
(pmod BLACKBOARD is protecting RAT .
  sort Blackboard .  subsort Nat < Blackboard .
  op empty : -> Blackboard [ctor] .
  op __ : Blackboard Blackboard -> Blackboard [ctor assoc comm id: empty] .
  vars M N K : Nat .
  prl [play] : M N => (K + N) quo 2
                      with probability K := (M * M -> 3/4 ; M * M * M -> 1/4) .
  op initState : -> Blackboard .
  eq initState = 2 3 5 7 11 13 17 .
endpm)
```

Since the multiset union `__` is associative and commutative (declared with the keywords `assoc` and `comm`), the choice of the numbers x and y from the blackboard is nondeterministic. The choice of the substitution, $\{\,\texttt{M} \mapsto x,\ \texttt{N} \mapsto y\,\}$ or $\{\,\texttt{M} \mapsto y,\ \texttt{N} \mapsto x\,\}$, is then also nondeterministic. We define the following probabilistic strategy `BlackboardStrat` to quantify this nondeterminism:

```
(psmod BLACKBOARD-PROB-STRAT is protecting BLACKBOARD . state Blackboard .
  var B : Blackboard . vars X Y : Nat .
  psdrule   RuleStrat := given state: B  is: (play) -> 1 .
  psdcontext CtxStrat := given state: X Y B  rule: play
                         is: ([] B) -> (1 / (X * Y)) .
  psdsubst   SubStrat := given state: X Y B  rule: play  context: [] B
                         is: {M <- X, N <- Y} -> 9 ;
                             {M <- Y, N <- X} -> 1 if X <= Y .
  psd BlackboardStrat := < RuleStrat | CtxStrat | SubStrat > .
endpsm)
```

The rule strategy `RuleStrat` assigns weight 1 to the only rule `play`. The context strategy `CtxStrat`, selecting in which context the rule `play` applies, assigns for each pair of numbers x and y on the blackboard, the relative weight $1/(x \cdot y)$ to the context that implies that the numbers x and y are replaced by the rule `play`; i.e., it gives a higher weight to contexts corresponding to picking small numbers to replace. The substitution strategy `SubStrat` selects the rule match $\{\texttt{M} \mapsto x,\ \texttt{N} \mapsto y\}$ with 9 times as high probability as $\{\texttt{M} \mapsto y,\ \texttt{N} \mapsto x\}$ when $x \leq y$.

We now explain the strategy `BlackboardStrat` in more detail. For a state $[u]_{ACU} = [\texttt{2 3 5 7 11 13 17}]_{ACU}$, `CtxStrat` assigns weights to each valid context as follows, where ACU refers to the structural axioms for multiset union (Associativity, Commutativity, and Unit (identity)). It first matches the state $[u]_{ACU}$ with the state pattern `X Y B` (where `B` can be `empty`), which gives several matches $\theta_1, \ldots, \theta_N$. Then all valid contexts are generated, which in this example have the form $[\odot\ t]_{ACU}$, with $\odot$ identifying the fragment of $[u]_{ACU}$ that matches the lefthand side of rule `play`, and t is the rest of the state. Next, the

weight of each valid context is computed. Each context $[\odot\ t]_{ACU}$ is unified with the context pattern `[] B`, giving a unique match $\{\texttt{B} \mapsto t\}$. Of all the matches $\theta_1, \ldots, \theta_N$ obtained above, only those with $\texttt{B} \mapsto t$ are kept. The weight associated to $[\odot\ t]_{ACU}$ is then computed by instantiating the weight pattern `1 / (X * Y)` with either one of these last substitutions. (For well-definedness, $\theta_i(\texttt{1 / (X * Y)})$ and $\theta_j(\texttt{1 / (X * Y)})$ should be the same, for all θ_i and θ_j with $\theta_i(\texttt{B}) = \theta_j(\texttt{B})$.) For example, for the context $[\mathbb{C}]_{ACU} = [\odot\ \texttt{3 7 11 13 17}]_{ACU}$ two such matches θ_k with $\theta_k(\texttt{B}) = \texttt{3 7 11 13 17}$ exist: $\theta_1 = \{\texttt{X} \mapsto \texttt{2},\ \texttt{Y} \mapsto \texttt{5},\ \texttt{B} \mapsto \texttt{3 7 11 13 17}\}$ and $\theta_2 = \{\texttt{X} \mapsto \texttt{5},\ \texttt{Y} \mapsto \texttt{2},\ \texttt{B} \mapsto \texttt{3 7 11 13 17}\}$. The weight of the context $[\mathbb{C}]_{ACU}$ is then computed as $\theta_1(\texttt{1 / (X * Y)}) = \texttt{1 / 10}$. Similarly, the weight of the context $[\odot\ \texttt{2 5 7 11 17}]_{ACU}$ is $1/(3 \cdot 13) = 1/39$. After computing the weights of all contexts, they are normalized to obtain a distribution, from which a context is picked.

The substitution strategy `SubStrat` solves the same matching and unification problems as above, but further refines the set of matches to those that satisfy the condition `X <= Y`. For the context $[\mathbb{C}]_{ACU}$ above the only such match is θ_1. The weight distribution pattern associated with `SubStrat` is then instantiated by θ_1 to obtain a concrete weight distribution over the matches of the lefthand side of rule `play`: $\{\texttt{M} \mapsto \texttt{2}, \texttt{N} \mapsto \texttt{5}\} \mapsto 9$; $\{\texttt{M} \mapsto \texttt{5}, \texttt{N} \mapsto \texttt{2}\} \mapsto 1$. By normalizing the associated weights, a probability distribution is obtained, from which a match is picked, e.g., $\eta = \{\texttt{M} \mapsto \texttt{2}, \texttt{N} \mapsto \texttt{5}\}$ with probability $9/10$.

Finally, a match for the probabilistic variable `K` of rule `play` is sampled from the distribution $\pi_r([\eta]_{ACU})$.

We run two simulations using unbounded probabilistic rewriting to show possible final states under the above strategy (the outputs are shown as comments):

```
(uprew initState using BlackboardStrat .) --- 276
(uprew initState using BlackboardStrat .) --- 4457
```

We then define a state predicate `sumGreaterThan(`i`)`, which is true in a state `S` if the sum of all numbers in `S` is larger than i:

```
(spmod BLACKBOARD-PRED is protecting BLACKBOARD . protecting NAT .
  smcstate Blackboard .
  psp sumGreaterThan : Nat . --- declare a parametric state predicate
  var B : Blackboard . var N : Nat .
  csat B |= sumGreaterThan(N) if sum(B) > N . --- define its semantics
  op sum : Blackboard -> Nat .
  eq sum(empty) = 0 .  eq sum(N B) = N + sum(B) .
endspm)
```

We first set a bound of 0.01 on both error probabilities, and then check that the sum on the blackboard never exceeds 10000 with high probability. This returns a positive result, and the estimated probability for the property to hold:

```
(set type1 error 0.01 .)
(set type2 error 0.01 .)
(smc initState |= P>= 0.9 [G ~ sumGreaterThan(10000)] using BlackboardStrat .)
Result Bool: true          Number of samples used: 11176
Confidence: 99%            Estimated probability: 79/87
```

4 Formalizing and Analyzing Cloud Computing Policies

In this section we use cloud computing to illustrate the usefulness of defining different adversaries for the same rewrite theory. The "base" rewrite theory defines all possible behaviors of the cloud and its environment, and each probabilistic strategy corresponds to a particular *load balancing policy* (and assumptions about the environment). We prove the safety of the "base" model, and use simulation and statistical model checking to analyze the QoS of different load balancing policies.

Cloud Computing Scenario. We model a cloud computing system that delivers services to *service users* and *service providers*. A service provider runs web applications on one or more *virtual machines* (VMs) hosted on *physical servers* in the cloud. Service users then use these applications via the cloud service. An example of a user is a person who uses an email application of a provider via a cloud infrastructure.

Servers are grouped into *data centers*, which are grouped into geographical *regions*. Since running applications in regions closer to the users may prove beneficial, we have included a *region selection* filter in our cloud architecture in Fig. 1. A *load balancer* distributes traffic across the data centers in a region.

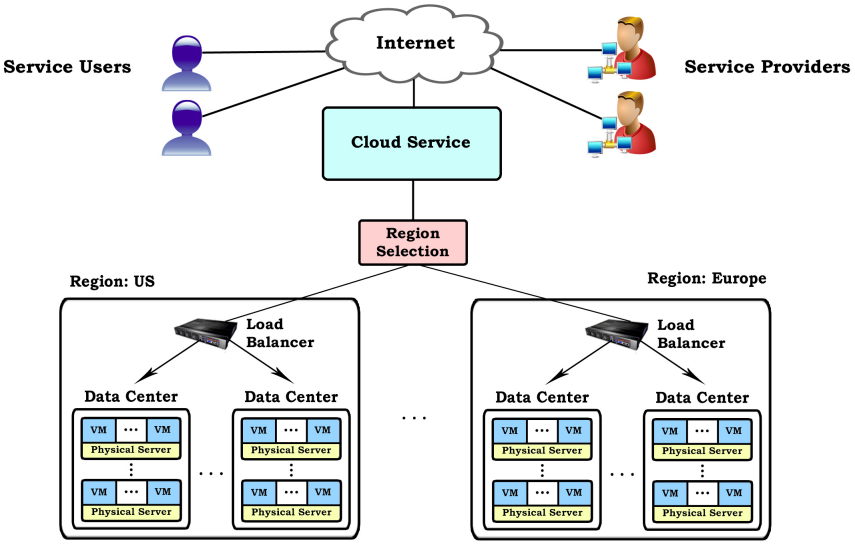

Fig. 1. A cloud computing architecture.

When a *user* sends an application request, the cloud service forwards it to one of the VMs of the provider that owns the application. When a *provider* sends a request to launch a VM in a region, the cloud service forwards it to the region's

load balancer, which chooses a server to host the new VM. Providers may launch a limited number of VMs, and users have a limited number of requests that the cloud can process *simultaneously*.

Formalization. We model the cloud system as a *hierarchical* object-oriented system. Service users, providers, and cloud services are declared as follows:

```
class Node | priority : Nat .
class SUser | location : Location .
class SProvider .
subclass SUser SProvider < Node .
class CService | status : CServiceDataSet, subscr : CServiceDataSet .
```

where `status` and `subscr` contain status and subscription information data about users and providers, of the forms: `noReq(`u`,` k`)`: the number of unresolved requests of user u is k; `noVM(`p`,` l`)`: the number of VMs of provider p is l; `maxReq(`u`,` k`)`: the maximum number of requests that the cloud service may simultaneously process for user u is k; `maxVM(`p`,` l`)`: the maximum number of VMs that provider p can run is l. A region has a `location`, and a `dataCenters` attribute with the data center objects in that region. The `pservers` attribute of a data center object denotes the set of physical server objects in the data center:

```
class Region | location : Location, dataCenters : Configuration .
class DCenter | pservers : Configuration .
class PServer | load : Nat, maxLoad : Nat, nextVMID : Nat, vms : Configuration .
```

with `load` the number of VMs on the server; `maxLoad` the server's capacity; `nextVMID` for generating fresh IDs for new VMs; and `vms` a set of VM objects of the class:

```
class VMachine | owner : Oid, vmReq : OidMSet, vmMaxReq : Nat, running : Bool .
```

with `owner` the ID of the provider that owns the VM; `vmReq` the object IDs of all users whose requests are running on the VM; `vmMaxReq` the maximum number of requests that can be resolved on the VM simultaneously; and `running` says whether the VM is running. We model two types of messages: *i)* user requests `req(`u`,`p`)` from user u to the application of provider p; *ii)* provider requests `launch(`p`,`r`)` from provider p to the cloud service, to launch a new VM in region r.

The following rule models how the cloud service handles a service request from user `O` to the web application of provider `O'` when `O` does not exceed her limit (`A < B`). This request can be forwarded to *any* VM in the system (since `dataCenter`, `pservers`, and `vms` denote *sets* of objects). The user's status is updated and the selected VM updates its `vmReq` attribute by adding the user's object ID `O` to the set `OIDSET`[4]:

```
crl [processUserReq]:
    req(O, O')
    < CSO : CService | status : (noReq(O, A), AS1) , subscr : (maxReq(O, B), AS2) >
```

[4] We do not show the declaration of variables, but follow the Maude convention that variables are in capital letters.

```
   < R : Region | dataCenters : (< DCO : DCenter |
                    pservers : (< PSO : PServer |
                         vms : (< VMO : VMachine | owner : O',
                                    running : true, vmReq : OIDSET, vmMaxReq : D >
                                VM) > PS) > DC) >
 => < CSO : CService | status : (noReq(O, A + 1), AS1) >
   < R : Region | dataCenters : (< DCO : DCenter |
                    pservers : (< PSO : PServer |
                         vms : (< VMO : VMachine | vmReq : (O OIDSET) > VM) > PS) > DC) >
if A < B /\ size(OIDSET) < D .
```

The next rule models how the cloud service handles `launch` requests from a provider `O` for a new VM in a region `R` by launching a VM on *any* server in `R`.

```
crl [processProviderReq]:
    launch(O, R)
    < CSO : CService | status : (noVM(O, A), AS1) , subscr : (maxVM(O, B), AS2) >
    < R : Region | dataCenters : (< DCO : DCenter |
                     pservers : (< PSO : PServer |
                                  load : M , maxLoad : N, nextVMID : NEXTID , vms : VM >
                                 PS) > DC) >
 => < CSO : CService | (status : (noVM(O, (A + 1)), AS1)) >
    < R : Region | dataCenters : (< DCO : DCenter |
                    pservers : (< PSO : PServer | load : (M + 1) , nextVMID : (NEXTID + 1),
                         vms : (< vm(PSO, NEXTID) : VMachine | owner : O, running : true,
                                   vmReq : noid, vmMaxReq : MAXREQ >  VM) > PS) > DC) >
if A < B /\ M < N .
```

Model Checking Safety Properties. To ensure the "safety" of our system we use Maude's `search` command to verify that the cloud is never processing more requests from a user than allowed. For the search to terminate, we use an initial state with two users that can generate 6 and 7 requests. The following Maude command searches for a "bad" state where some user `O'` has more requests running (`M`) than allowed (`N`):

```
(search [1] : initState =>*
   CONFIG  < CSO : CService | status : (noReq(O,M), AS1), subscr : (maxReq(O,N), AS2) >
 such that M > N .)
rewrites: 11312174 in 279134ms cpu (283262ms real) (40525 rewrites/second)
No solution.
```

Load Balancing Policies as Probabilistic Strategies. The above model describes *all possible* treatments of user and provider requests. For better system performance, one could think of different *load balancing policies*, so that, e.g., users may get with high probability a VM closer to their location, or get VMs on servers with the least workload, etc. Likewise, a provider might want VMs uniformly across the regions, or in regions with least workload, etc. Within a region, better-paying providers could be assigned VMs with high probability on stable servers, or on servers with small workload. These policies are specified by different strategies on top of our verified model.

We have defined three strategies: *i)* a strategy that does not take locations or geographical regions into account, i.e., the region to which a user request is

forwarded is chosen uniformly at random; *ii)* a strategy that forwards high priority user requests to the region closest to the user with high probability, and furthermore, treats requests from high priority providers with high probability; and *iii)* a strategy that forwards high priority user requests to a region with small VM load with high probability, and processes requests from high priority providers with high probability. For each of these policies, we define two "subcases": *a)* distribute requests uniformly within the region, and *b)* with high probability, distribute requests to VMs/physical servers with small workload for the high-priority providers/users.

We first quantify the nondeterministic choices related to general/environment assumptions about the cloud system, using the following rule strategy:

```
psdrule RuleStrat := given state: CF
                     is: (resolveUserReq) -> 10 ;
                         (processUserReq) -> 1000  ; (processProviderReq) -> 100 ;
                         (failVM)         -> 1     ; (migrateVM)          -> 1   ;
                         (newUserReq)     -> 100   ; (newProviderReq)     -> 10  .
```

Our load balancing policies are specified by different context strategies for rules `processUserReq` and `processProviderReq`. They resolve the nondeterministic choice of which request to process next, and on which VM/server the request is resolved. We only specify the policy *ii(b)*, and refer to [5] for the definitions of the other policies. The rule `processUserReq` selects *any* user request `req(`u`,` p`)`, and assigns *any* VM to handle the request, in *any* region. For policy *ii(b)*, the context strategy for rule `processUserReq` models that user requests with high priorities `P` are selected with probability proportional to $\mathtt{P}^2$, that the probability of selecting region `R` is inversely proportional to the distance `distance(LOC, LOC')` between the user and `R`, and that requests are sent to the VMs/servers with small load `size(OIDSET)` with high probability:

```
psdcontext CtxStrat2b :=
given state: CF req(O, O') < O : SUser | location : LOC, priority : P >
             < CSO : CService | ATTRSET >
             < R : Region |  location : LOC',
                        dataCenters : (< DCO : DCenter |
                           pservers : (< PSO : PServer | ATTRSET',
                                vms : (< VMO : VMachine | vmReq : OIDSET, ATTRSET'' >
                                       VM) > PS) > DC) >
       rule: processUserReq
         is: (CF < O : SUser | location : LOC, priority : P > [])
              -> ((P * P) / ((1 + size(OIDSET)) * (1 + distance(LOC, LOC')))) .
```

The context strategy for rule `processProviderReq` models that requests from providers with high priorities `P` are selected with high probability proportional to $\mathtt{P}^2$, and resolved by allocating VMs on servers with small load `M` with high probability:

```
psdcontext CtxStrat2b :=
  given state: CF < CSO : CService | ATTRSET >
               launch(O, R) < O : SProvider | priority : P >
               < R : Region |    location : LOC',
```

```
                    dataCenters : (< DC0 : DCenter |
                       pservers : (< PS0 : PServer | load : M, ATTRSET > PS)
                                  > DC) >
rule: processProviderReq
  is: (CF < O : SProvider | priority : P > []) -> ((P * P) / (1 + M)) .
```

The resulting probabilistic strategy is denoted `Strat2b`.

Simulation. We simulate our system using the strategies `Strat2b` and `Strat3b` (specifying policies *ii(b)* and *iii(b)* above). To analyze the QoS from a user U's perspective, we define a QoS measure of a request R as a function of the relative workload of the server S handling the request, and the distance between the user and S:

$$cost(U, R, S) = k \cdot \frac{\#tasksRunning(S)}{capacity(S)} + q \cdot distance(U, R)$$

The total user QoS is the sum of all such single QoS measures in the system.

We simulate the cloud system using `Strat2b` and `Strat3b`, from an initial state with 2 regions, 4 users, and 2 providers in different locations and with different priorities. Users can generate 3, 5, 8, and 3 requests, respectively, whereas providers can generate 20 requests.

```
(prew [200] initState using Strat2b .)   (prew [200] initState using Strat3b .)
```

Fig. 2 shows the total *cumulative* cost over 200 probabilistic rewrite steps (the horizontal axis), for the two strategies (using the same random seed). The smaller values correspond to `Strat2b`, which suggests that `Strat2b` is a better policy in terms of the total user QoS. We next use statistical model checking to show that, with high confidence, this is the case.

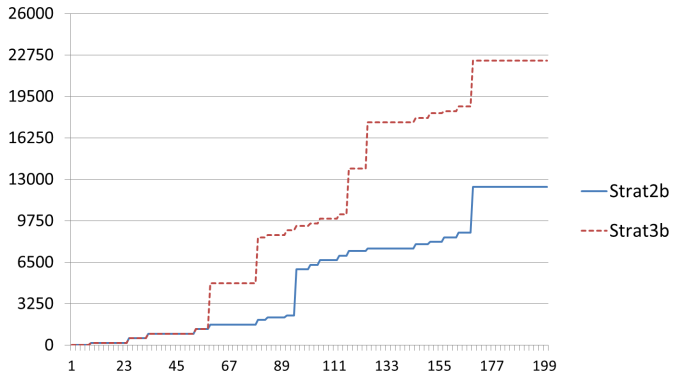

Fig. 2. Comparison of the total cloud cost with each rewrite step, for the two policies.

Statistical Model Checking. We use the same initial state as for the simulation, and define a parametric state predicate `effortGreaterThan` as follows:

```
psp effortGreaterThan : Nat .  var CF : Configuration . var N : Nat .
csat CF |= effortGreaterThan(N) if costCloud(CF) > N .
```

such that `effortGreaterThan(`N`)` is `true` if the cumulative cost of the cloud exceeds N in the current state. We verify that, with confidence 0.9, under the strategy `Strat2b`, the effort always stays below 13000 with high probability:

```
(smc initState |= P> 0.9 [G ~ effortGreaterThan(13000)] using Strat2b .)
rewrites: 36338205 in 1024184ms cpu (1024339ms real) (35480 rewrites/second)
Result Bool: true            Number of samples used: 3520
Confidence: 90 %             Estimated probability: 1
```

and that the effort under `Strat3b` will exceed 15000, with high probability:

```
(smc initState |= P> 0.9 [F effortGreaterThan(15000)] using Strat3b .)
rewrites: 128202820 in 2927619ms cpu (2928049ms real) (43790 rewrites/second)
Result Bool: true            Number of samples used: 7150
Confidence: 90 %             Estimated probability: 1
```

5 Related Work

A number of tools support models that are both probabilistic and nondeterministic, including Markov automata [14], generalized stochastic Petri nets [21], or uniform labeled transition systems [6].

In probabilistic automaton-based models one can use synchronous parallel composition to quantify nondeterministic choices by composing the system with a new "scheduler" component. For example, if the model allows us to nondeterministically select action a or action b, we can quantify this nondeterminism by composing the model with a "scheduler" automaton with transitions $\{s_0 \xrightarrow{\tau\,[1/3]} \mathit{do\text{-}a},\ s_0 \xrightarrow{\tau\,[2/3]} \mathit{do\text{-}b},\ \mathit{do\text{-}a} \xrightarrow{\overline{a}} s_0,\ \mathit{do\text{-}b} \xrightarrow{\overline{b}} s_0\}$; the composed system will then do action a with probability $1/3$ and action b with probability $2/3$. Such "scheduler" components are supported by tools like MODEST [7] and PRISM [19]. Our approach contrasts with this one by: (*i*) having a more explicit separation between model and strategy, since in the above approach the strategy is just another "system" component; (*ii*) supporting a more expressive underlying specification language with unbounded data types, dynamic object/message creation and deletion, and arbitrary complex data types; and (*iii*) providing a more expressive and convenient way of specifying the strategies themselves. It is also unclear to what extent automaton-based approaches can support *hierarchical* systems.

In Uppaal-SMC [12] the nondeterminism concerning *which* component should be executed next is *implicitly* resolved by assigning a stochastic delay to each component; the one with the shortest remaining delay is then scheduled for execution. If multiple transitions are enabled for a component, then one is chosen

uniformly at random. In contrast, our language allows the user to specify the probability distributions that quantify the nondeterminism in the model.

Maude itself has a non-probabilistic strategy language [15] to guide the execution of *non-probabilistic* rewrite theories; i.e., there is no support for quantifying the nondeterminism either in the model or in the strategy. VeStA [27] can analyze fully probabilistic actor PMaude specifications. For *flat* object-oriented systems nondeterminism is typically removed by letting each action be triggered by a message, and letting probabilistic rules add a stochastic delay to each message [1, 3]. The probability of two messages being scheduled at the same "time" is then zero, and this therefore resolves nondeterminism. This method was recently extended to *hierarchical* object-oriented systems [13, 10]. The differences with our work are: (i) whereas we have a clear separation between system model and adversary, the above approach encodes the adversary in the model, and hence clutters it with fictitious clocks and schedulers to obtain a fully probabilistic model; (ii) our implementation supports not only a subset of object-oriented specifications, but the entire class of PRTs with fixed-size probability distributions; and (iii) we add a simulator and statistical model checker to Maude instead of using an external tool.

ELAN [8] is a rewriting language where strategies are first class citizens that can appear in rewrite rules, so there is no separation between system model and strategies. The paper [9] adds a "probabilistic choice" operator $\texttt{PC}(s_1 : p_1, \ldots, s_n : p_n)$ to ELAN's strategy language, where p_i defines the probability of applying strategy s_i. This approach is different from ours in the following ways: there is no separation between "system" and "strategy;" the definition of context and substitution adversaries is not supported and therefore not all nondeterminism in a system can be quantified; and there is no support for probabilistic model checking analysis.

6 Concluding Remarks

In this paper we define what is, to the best of our knowledge, the first language for defining complex probabilistic strategies to quantify the nondeterminism in infinite-state systems with both probabilities and nondeterminism, and that includes the object-oriented systems with dynamic object creation. We propose a modular safety/QoS analysis methodology in which a simple (typically non-probabilistic) "base" model that defines all possible system behaviors can be easily verified for safety, and where different probabilistic refinements can be specified on top of the verified base model. QoS properties of the refinements can then be analyzed by statistical and exact probabilistic model checking.

We have implemented a probabilistic simulator and statistical PCTL model checker for our strategies for all probabilistic rewrite theories with discrete probability distributions. We show the usefulness of our language and methodology on a cloud computing example, where different probabilistic strategies on top of the verified base model define different load balancing policies for the cloud, and show how our tool PSMaude can be used to compare the QoS provided by dif-

ferent policies. This example indicates that we need to integrate timed modeling and analysis into our framework.

We also plan to extend PSMaude to allow the statistical analysis of quantitative temporal expressions specified in the QuaTEx logic [1]. This would allow, e.g., the statistical estimation of *expected values* of particular numerical quantities associated with a probabilistic rewrite theory whose nondeterminism is quantified by a given probabilistic strategy. Another direction for future work is to investigate algorithms for the *exact* (vs. statistical) probabilistic model checking of probabilistic rewrite theories for which *some*, but not necessarily all nondeterminism is quantified by a "partial" probabilistic strategy. Finally, since our proposed probabilistic strategy language only allows defining memoryless adversaries, we also aim to extend its syntax and semantics to allow defining *history-dependent* adversaries.

Acknowledgments. We thank José Meseguer and Roy Campbell for discussions on probabilistic strategies, and gratefully acknowledge partial support for this work by AFOSR Grant FA8750-11-2-0084. We also thank the anonymous reviewers for very useful comments on a previous version of the paper.

References

1. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. ENTCS 153(2) (2006)
2. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: CALCO. LNCS, vol. 6859. Springer (2011)
3. AlTurki, M., Meseguer, J., Gunter, C.A.: Probabilistic modeling and analysis of DoS protection for the ASV protocol. ENTCS 234 (2009)
4. Bentea, L.: The PSMaude tool home page: `http://folk.uio.no/lucianb/prob-strat/`
5. Bentea, L., Ölveczky, P.C.: A probabilistic strategy language for probabilistic rewrite theories and its application to cloud computing. Manuscript: `http://folk.uio.no/lucianb/publications/2012/pstrat-cloud.pdf`
6. Bernardo, M., Nicola, R., Loreti, M.: Uniform labeled transition systems for nondeterministic, probabilistic, and stochastic processes. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) Trustworthly Global Computing, LNCS, vol. 6084, pp. 35–56. Springer Berlin Heidelberg (2010)
7. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MODEST: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. 32(10), 812–830 (2006)
8. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.E., Ringeissen, C.: An overview of ELAN. Electronic Notes in Theoretical Computer Science 15 (1998)
9. Bournez, O., Kirchner, C.: Probabilistic rewrite strategies. Applications to ELAN. In: RTA. LNCS, vol. 2378. Springer (2002)
10. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: Modelling and analyzing adaptive self-assembly strategies with Maude. In: WRLA. LNCS, vol. 7571, pp. 118–138. Springer (2012)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)

12. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: CAV. LNCS, vol. 6806. Springer (2011)
13. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: FASE. LNCS, vol. 7212. Springer (2012)
14. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: Logic in Computer Science. pp. 342–351 (2010)
15. Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. Electronic Notes in Theoretical Computer Science 174(11), 3–25 (2007)
16. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6 (1994)
17. Katelman, M., Meseguer, J., Hou, J.C.: Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In: FMOODS. LNCS, vol. 5051. Springer (2008)
18. Kumar, N., Sen, K., Meseguer, J., Agha, G.: Probabilistic rewrite theories: Unifying models, logics and tools. Technical report UIUCDCS-R-2003-2347, Department of Computer Science, University of Illinois at Urbana-Champaign (2003)
19. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. LNCS, vol. 6806. Springer (2011)
20. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94(1), 1–28 (1991)
21. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with generalized stochastic Petri nets. SIGMETRICS Performance Evaluation Review 26(2), 2 (1998)
22. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theoretical Computer Science 285(2) (2002)
23. Martí-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. Electronic Notes in Theoretical Computer Science 117, 417–441 (2005)
24. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1) (1992)
25. Meseguer, J.: A rewriting logic sampler. In: ICTAC. LNCS, vol. 3722. Springer (2005)
26. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: CAV. LNCS, vol. 3576. Springer (2005)
27. Sen, K., Viswanathan, M., Agha, G.A.: VeStA: A statistical model-checker and analyzer for probabilistic systems. In: QEST'05. IEEE Computer Society (2005)
28. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: CAV. LNCS, vol. 2404. Springer (2002)