Finite Horizon Analysis of Stochastic Systems with the Murφ Verifier*

Giuseppe Della Penna[1], Benedetto Intrigila[1,**], Igor Melatti[1], Enrico Tronci[2], and Marisa Venturini Zilli[2]

[1] Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
{dellapenna,intrigila,melatti}@di.univaq.it
[2] Dip. di Informatica Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy
{tronci,zilli}@dsi.uniroma1.it

Abstract. Many reactive systems are actually *Stochastic Processes*. Automatic analysis of such systems is usually very difficult thus typically one simplifies the analysis task by using simulation or by working on a simplified model (e.g. a *Markov Chain*).
We present a *Finite Horizon Probabilistic Model Checking* approach which essentially can handle the same class of stochastic processes of a typical simulator. This yields easy modeling of the system to be analysed together with formal verification capabilities. Our approach is based on a suitable *disk based* extension of the Murφ verifier.
Moreover we present experimental results showing effectiveness of our approach.

1 Introduction

Correctness of digital hardware, embedded software and protocols can often be verified with *Model Checking* techniques [5,9,14,13,18,26] by modeling such systems as *Nondeterministic Finite State Systems* (NFSS).

However, there are many reactive systems that exhibit uncertainty in their behavior, i.e. which are stochastic systems. Examples of such systems are: fault tolerant systems, randomized distributed protocols and communication protocols. Typically stochastic systems cannot be conveniently modeled using NFSS. However they can often be modeled as *Stochastic Processes* [19].

Unfortunately, automatic analysis of stochastic processes is quite hard, apart from some noticeable special classes of stochastic processes. For this reason typically approximated approaches are used. Namely: *simulation* or *model approximation*. *Simulation* carries out an approximate analysis on the given stochastic

* This research has been partially supported by MURST projects: MEFISTO and SAHARA

** Corresponding Author: Benedetto Intrigila. Tel: +39 0862 43 31 32. Fax: +39 0862 43 31 80

C. Blundo and C. Laneve (Eds.): ICTCS 2003, LNCS 2841, pp. 58–71, 2003.

process. *Model approximation* carries out an exact analysis on a simplified (approximated) model of the given stochastic process. For example, *Markov Chains* [3,11] can be used to approximate a given stochastic process.

Automatic analysis of *Markov Chains* can be effectively performed by using *Probabilistic Model Checkers* [28,6,17,23,12,25,4,7,8,2,15,27].

Probabilistic Model Checkers have been developed also for some particular class of *Stochastic Processes* [10], namely those in which the probability of an outgoing transition from state s is a function of the sojourn time in state s (semi-Markov Processes).

Stochastic Simulators [19] typically can handle fairly general *stochastic systems*. However, from a simulator we can only get information about the *average behavior* of the system at hand, whereas from a model checker we also get information about *low probability* events.

In this paper we focus on *Discrete Time Stochastic Processes* (SP). Our goal is to compute the probability that a given SP reaches an *error state* in at most k steps starting from a given *initial state* (*Finite Horizon Verification*).

We will present an approach and a tool to carry out *Finite Horizon Verification* of a class of SP that is essentially as large as the class of SP that can be handled by many simulators (e.g. [27,19]). To the best of our knowledge, this is the first time that such an approach is presented. Our results can be summarized as follows.

1. We present (Section 3) *Probabilistic Rule Based Transition Systems*(PRBTS) and show (Section 4) how PRBTS can be used to model a fairly large class of *Finite State* SP (*Discrete Time Stochastic Processes*). By using *finite precision real numbers* as in [21] (and as in any simulator) we can also handle *Discrete Time Hybrid Stochastic Processes*, i.e. stochastic processes which have continuous (i.e. finite precision real) as well as discrete state variables.
2. PRBTS can be used as a *low level* language to define stochastic systems. This is useful to separate the task of designing high level languages for system descriptions from the task of designing *Verification Engines*. As an example, we show (Section 5) how a *high level Communicating Processes* definition of a stochastic system can be translated, in linear time, into a *low level* PRBTS definition of the same system.
3. We show (Section 7) how FHP-Murφ [22], a suitable *disk based* extension of the Murφ verifier [18], can be used for automatic *Finite Horizon Verification* of PRBTS.

   Indeed, using FHP-Murφ, PRBTS can be used as a *low level* language to define stochastic systems whereas FHP-Murφ can be used as a low level *Verification Engine* for *Finite Horizon Verification* of stochastic systems.
4. We show (Section 7) effectiveness of our approach by presenting experimental results on automatic analysis of two nontrivial stochastic systems using with FHP-Murφ.

   Our experimental results show that FHP-Murφ can handle more general models than state-of-the-art *Probabilistic Model Checkers* like PRISM [24,2,16] or TwoTowers [27].

On the other hand PRISM as well as TwoTowers can verify more general properties (e.g. all PCTL [12] properties for PRISM) than FHP-Murφ. In fact FHP-Murφ can only handle *Finite Horizon Verification*.

2 Basic Notation

We give some basic definitions on Finite State/Discrete Time General Stochastic Processes. For more details on stochastic processes see, e.g., [20].

Definition 1.

1. *A* Finite State/Discrete Time Stochastic Process (*shortened* SP *in the following*) *is a triple* $\mathcal{X} = (S, \mathbf{P}, q)$ *where* S *is a finite set (of* states*),* $q \in S$ *is the initial state,* $Seq(S)$ *is the set of all finite sequences of elements of* S*, and* $\mathbf{P} : S \times Seq(S) \times S \to [0,1]$ *is a transition probability function, i.e. for all* $s \in S$*,* $\pi \in Seq(S)$*,* $\sum_{t \in S} \mathbf{P}(s, \pi, t) = 1$*. (We included the* initial state *q in the SP definition to simplify our exposition.)*
2. *An* execution sequence (*or* path) *in the SP* $\mathcal{X} = (S, \mathbf{P}, q)$ *is a nonempty (finite or infinite) sequence* $\pi = s_0 s_1 s_2 \ldots$ *where* s_i *are states and*

$$\mathbf{P}(s_i, s_0 \ldots s_{i-1}, s_{i+1}) > 0$$

for $i = 0, 1, \ldots$*. If* $\pi = s_0 s_1 s_2 \ldots$ *we write* $\pi(k)$ *for* s_k*, and we write* $\pi|k$ *for the sequence* $s_0 s_1 s_2 \ldots s_{k-1}$*. The* length *of a finite path* $\pi = s_0 s_1 s_2 \ldots s_k$ *is* k *(number of transitions), whereas the length of an infinite path is* ∞*. We denote with* $|\pi|$ *the length of* π*. We denote with* $Path(\mathcal{X}, s)$ *the set of infinite paths* π *in* $\mathcal{X}$ *s.t.* $\pi(0) = s$*. If* $\mathcal{X} = (S, \mathbf{P}, q)$ *we write also* $Path(\mathcal{X})$ *for* $Path(\mathcal{X}, q)$*.*
3. *For* $s \in S$ *we denote with* $\sum(s)$ *the smallest* σ*-algebra on* $Path(\mathcal{X}, s)$ *which, for any finite path* ρ *starting at* s*, contains the basic cylinders* $\{ \pi \in Path(\mathcal{X}, s) \mid \rho \text{ is a prefix of } \pi \}$*. The probability measure* Pr *on* $\sum(s)$ *is the unique measure with* $Pr\{ \pi \in Path(\mathcal{X}, s) \mid \rho \text{ is a prefix of } \pi \} = \mathbf{P}(\rho) = \prod_{i=0}^{k-1} \mathbf{P}(\rho(i), \rho|i, \rho(i{+}1)) = \mathbf{P}(\rho(0), \epsilon, \rho(1))\, \mathbf{P}(\rho(1), \rho|1, \rho(2)) \cdots \mathbf{P}(\rho(k{-}1), \rho|(k{-}1), \rho(k))$*, where* $k = |\rho|$ *and* ϵ *is the empty sequence.*

We recall that a *Markov Chain* is a particular SP, such that the probability transition function $\mathbf{P}(s, \pi, t)$ actually does not depend on π ("lack of memory") and therefore reduces to a *Stochastic Matrix* (see [3]).

Given a SP, we want to compute the probability that a path of length k starting from the initial state q reaches a state s satisfying a given boolean formula ϕ.If ϕ models an *error condition*, this computation allows us to compute the probability of reaching an error condition in at most k transitions.

Problem 1. Let $\mathcal{X} = (S, \mathbf{P}, q)$ be a SP, $k \in \mathbb{N}$, and ϕ be a boolean function on S. We want to compute: $P(\mathcal{X}, k, \phi) = Pr((\exists i \leq k\ \phi(\pi(i))) \mid \pi \in \mathrm{Path}(\mathcal{X}))$. That is, we want to compute the probability of reaching a state satisfying ϕ in *at most* k steps in the SP $\mathcal{X}$ (starting from the initial state q).

Problem 1 can be very difficult both from a computational and from an analytical point of view [4,6,7]. So, the first task is to single out a (large enough) class of *tractable* SP. Moreover, we need to better specify the computational model we want to use. We introduce this model in Section 3. Then, in Section 4 we will show how we intend to cope with our verification problem.

3 Probabilistic Rule Based Transition Systems

Definition 2. *A* Probabilistic Rule Based Transition System *(PRBTS) $\mathcal{S}$ is a 3-tuple $(S, \texttt{Rules}, q)$, where: S is a finite set (of states), $q \in S$ and* `Rules` *is a finite set of pairs (p, f), with p being a function from S to $[0,1]$ and f being a function from S to S and $\forall s \in S \sum_{(p,f)\in\texttt{Rules}} p(s) = 1$.*

Definition 3. *Let $\mathcal{S} = (S, \texttt{Rules}, q)$ be a PRBTS. An* execution sequence *in $\mathcal{S}$ is a nonempty (finite or infinite) sequence $\pi = s_0 s_1 s_2 \ldots$ where s_i are states and for every $i = 0, 1, \ldots$ there exists a pair $(p, f) \in$ `Rules`, such that $f(s_i) = s_{i+1}$ and $p(s_i) > 0$.*

As expected, to a PRBTS we can univocally associate a Markov Chain. This can be done as follows.

Definition 4. *Let $\mathcal{S} = (S, \texttt{Rules}, q)$ be a PRBTS. The Markov Chain $\mathcal{S}^{mc} = (S, \mathbf{P}, q)$ associated to $\mathcal{S}$ is defined as follows: $\mathbf{P}(s,t) = \sum_{(p,f)\in\texttt{Rules} s.t. f(s)=t} p(s)$ (taking as 0 summation on an empty set).*

Proposition 1. *Let $\mathcal{S} = (S, \texttt{Rules}, q)$ be a PRBTS. Then, the Markov Chain $\mathcal{S}^{mc}$ associated to $\mathcal{S}$ is well defined.*

4 From Stochastic Processes to PRBTS

As we discussed in Section 1, we cannot hope to analyze all possible SP. So, we restrict our analysis to SP such that their transition probabilities depend only on some fixed characteristics of the process history. We formalize this as follows.

Definition 5. *Let the SP $\mathcal{X} = (S, \mathbf{P}, q)$ be given. We say that $\mathcal{X}$ has* finite character *n iff there exists an equivalence relation R on $Seq(S)$ of finite index n (that is with n equivalence classes) such that for every π_1, $\pi_2 \in Seq(S)$*

$$\textit{if } R(\pi_1, \pi_2) \textit{ then } \forall s, t \in S.\, \mathbf{P}(s, \pi_1, t) = \mathbf{P}(s, \pi_2, t)$$

Now we show that to a finite character SP $\mathcal{X}$ we can associate a PRBTS $\mathcal{S}$, in such a way that the verification Problem 1 for $\mathcal{X}$ can be reduced to that for $\mathcal{S}$.

Proposition 2. *Let the SP* $\mathcal{X} = (S, \mathbf{P}, q)$ *be of finite character* n *w.r.t. an equivalence relation* R*. Let moreover* $Q_0, \ldots, Q_{n-1}$ *be an enumeration of the equivalence classes of* R*. Then there exists a PRBTS* $\mathcal{S} = (S_1, \texttt{Rules}, q_1)$*, such that:*

1. $S_1 = S \times \mathbf{n}$*, where* $\mathbf{n}$ *denotes the set* $\{0, \ldots, n-1\}$*;*
2. *if* π *is any sequence in Path*$(\mathcal{X})$*, such that* $\pi \in Q_i$ *and* $\pi_1 = \pi s$ *is in* Q_j*, where by* πs *we denote the concatenation of* s *to the sequence* π*, and* $\mathbf{P}(s, \pi, t) > 0$*, then*
   - *there exists at least one pair* (p, f) *in* `Rules` *such that* $f((s,i)) = (t,j)$ *and* $p((s,i)) > 0$*,*
   - $\sum_{(p,f) \in \texttt{Rules}\ s.t. f((s,i))=(t,j)} p(s) = \mathbf{P}(s, \pi, t)$*;*
3. $q_1 = (q, i_0)$*, where* $q \in Q_{i_0}$*;*
4. *Problem 1 on* $\mathcal{X}$ *with respect to* ϕ *can be reduced to compute:* $P(\mathcal{S}^{mc}, k, \phi_1) = Pr_{\mathcal{S}^{mc}}((\exists i \leq k\ \phi_1(\pi(i))) \mid \pi \in \mathrm{Path}(\mathcal{S}^{mc}))$ *where* $\forall j \in \mathbf{n}$*,* $\phi_1((s,j)) = \phi(s)$*, that is* $P(\mathcal{X}, k, \phi) = P(\mathcal{S}^{mc}, k, \phi_1)$*.*

Proof. (Sketch) It is easy to see that a PRBTS $\mathcal{S}$, verifying the required conditions, can be specified from $\mathcal{X}$: simply insert in `Rules` a suitable pair (p, f) of functions, for every transition $\mathbf{P}(s, \pi, t) > 0$, taking into account to choose one representative for each equivalence class. As an example, given $\mathbf{P}(s, \pi, t) > 0$ with $\pi \in Q_i$ and $\pi s \in Q_j$, set f as the constant function on S_1 returning always (t, j), and set p as the function that returns $\mathbf{P}(s, \pi, t)$ for input (s, i) and 0 otherwise. For the last point, observe that for every such $\mathcal{S}$, the associated Markov Chain $\mathcal{S}^{mc}$ gives rise to essentially the same probability measure of $\mathcal{X}$ on cylinders and therefore on every set. Indeed, given a path $\pi \in \mathrm{Path}(\mathcal{S}^{mc})$ the indexes in π give no information, since they are univocally determined by the path π itself.

We stress that a PRBTS is *always defined by a program* of a suitable (e.g. C-like) programming language. This allows us to specify functions $(p, f) \in$ `Rules` inside the program as *procedures*. This makes their formulation parametric and concise. On the basis of such considerations, we state the following claim:

Claim. A *rule based (i.e. PRBTS oriented)* approach to SP specification is *in many cases* exponentially shorter than a *Markov Chain based* specification approach. By a *Markov Chain based* specification approach we mean any language requiring *in many cases* an explicit (i.e. tabular) definition of the stochastic matrix of the input Markov Chain.

In fact, by comparing the protocol LQS modeled in Section 7.1 (with FHP-Murφ, so with PRBTS) with the model of the same protocol in PRISM (that it is not included here, for space reasons: see [29]), we can see that the former is much shorter than the latter, since it does not grow with the parameter `ITEM_Q`.

One may wonder whether this is only a problem of language expressiveness. In a sense, this is not the case, since PRISM needs to store in memory the *complete* Markov Chain stochastic matrix. On the opposite, FHP-Murφ treats the Markov Chain exactly with the transition rules given in the model and it does not need to generate all the transition matrix.

5 From Communicating Stochastic Processes to PRBTS

As an example of usage of PRBTS as a low level definition language for SP, in this Section we show how the definition of an SP $\mathcal{S}$ specified by *Communicating Stochastic Processes* can be translated into a suitable PRBTS.

Definition 6. *A* System of Communicating Stochastic Processes *(SCSP) $\mathcal{S}$ is a 4-tuple $(n, S, \mathbf{q}, \mathcal{R})$, where:*
n is an integer (denoting the number of processes in our system);
$S = S_1 \times \ldots \times S_n$ is the Cartesian product of finite sets (of states) S_i, $i = 1, \ldots, n$;
$\mathbf{q} = (q_1, \ldots, q_n) \in S$;
$\mathcal{R} = \langle \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ is a n-tuple of sets $\mathcal{R}_i$ $i = 1, \ldots, n$ s.t. $\mathcal{R}_i$ is a finite set of pairs (p, f) where p is a function from S to $[0, 1]$, f is a function from S to S_i, and $\forall i \in \{1, \ldots, n\}$ $\forall s \in S$ $\sum_{(p,f)\in\mathcal{R}_i} p(s) = 1$.

In the following we denote with boldface letters (e.g. $\mathbf{x}$) elements of $S = S_1 \times \ldots \times S_n$ and with x_i the i-th component of $\mathbf{x}$. We can define the transition relation of a SCSP assuming that processes are scheduled with uniform probability ($1/n$ if we have n processes).

Definition 7. *Let $\mathcal{S} = (n, S, \mathbf{q}, \mathcal{R})$ be a SCSP. The Markov Chain $\mathcal{S}^{mc} = (S, \mathbf{P}, \mathbf{q})$ associated to $\mathcal{S}$ is defined as follows:*
$\mathbf{P}(\mathbf{s}, \mathbf{t}) = \sum_{i=1}^{i=n} \sum_{(p,f)\in\mathcal{R}_i \text{ s.t. } (s_1,\ldots,s_{i-1},f(\mathbf{s}),s_{i+1},\ldots,s_n)=\mathbf{t}} (\frac{1}{n} \cdot p(\mathbf{s}))$
(taking as 0 summations on empty sets).

Essentially PRBTS are (probabilistic) *shared variable* concurrent programs. Thus it is not surprising [1] that a SCSP can be transformed into a PRBTS using a suitable uniform probability *scheduler*. The following definition shows how this can be done (e.g. along the lines in PRISM [24]).

Definition 8. *Let $\mathcal{S} = (n, S, \mathbf{q}, \mathcal{R})$ be a SCSP. We denote with $\Gamma(\mathcal{S})$ the PRBTS $(S, \mathbf{q}, \texttt{Rules})$ defined as follows:* $\texttt{Rules} = \cup_{i=1}^{i=n} \cup_{(p,f)\in\mathcal{R}_i} \{(\lambda\mathbf{x}.(\frac{1}{n} \cdot p(\mathbf{x})), f)\}$

The following proposition follows immediately from the construction in Definition 8.

Proposition 3. *Let $\mathcal{S}$ be a SCSP. Then $\mathcal{S}^{mc} = \Gamma(\mathcal{S})^{mc}$*

Remark 1. Note that the PRBTS transformation of a SCSP is not limited to the case in which the processes are scheduled with a uniform probability. In fact, it is sufficient to modify Definition 8 in this way: $\texttt{Rules} = \cup_{i=1}^{i=n} \cup_{(p,f)\in\mathcal{R}_i} \{(\lambda\mathbf{x}.(s(i) \cdot p(\mathbf{x})), f)\}$, where s is a function from $\{1, \ldots, n\}$ to $[0, 1]$ denoting the scheduling probability of the process $i \in \{1, \ldots, n\}$ (obviously, s must be such that $\sum_{i=1}^{n} s(i) = 1$).

6 Defining Probabilistic Systems with the Murφ Verifier

We want to extend the input language of the Murφ verifier to allow definition of SP using PRBTS. Since Murφ input language defines NFSS, our main modification to Murφ input language consists of adding transition probabilities to transition rules.

In this Section we show how we modified Murφ input language to achieve the above goal thus defining FHP-Murφ input language. The length of FHP-Murφ *finite horizon* is passed on the command line to FHP-Murφ.

6.1 FHP-Murφ Input Language

We modify Murφ input language in the following parts: 1. We add a probability specification to each start state; 2. We change the semantics of rules; 3. We only allow one invariant to which we add a probability bound.

To handle *Discrete Time Hybrid Stochastic Processes* it is useful to have state variables ranging on real numbers. For this reason in the following we will consider the Murφ version enhanced with *finite precision real numbers*, as described in [21].

To add probabilities in definitions of startstates, we modify the `startstate` nonterminal production of the Murφ language grammar (Chapter 7 of the documentation [18]) as follows: `<startstate> ::=`
`startstate [<string>] [<realexpr>] [{<decl>} begin] [<stmts>] end`
where the expression `realexpr` must evaluate to a real number in $[0, 1]$, and defaults to 1 when it is not specified. If we are given h startstates with probabilities $p_1, \ldots, p_h$, then $\sum_{i=1}^{h} p_i$ has to be 1, or FHP-Murφ will return an error.

To add probabilities on rules, we modify the semantics of the `simplerule` nonterminal production of the Murφ language grammar (Chapter 7 of the documentation [18]) as follows. The original production, without priority and fairness (not modified in our work), was
`<simplerule> ::= rule [ <expr> ] ==> [ <decl> begin ] [ stmts ] end`.

In FHP-Murφ, we simply require the expression after the keyword `rule` (i.e. `expr`) to be a real expression valued in $[0, 1]$, instead of a boolean as it is for Murφ. FHP-Murφ does not allow simultaneous use of both boolean and probability based rules.

The above modification to `<simplerule>` has a deep impact on Murφ semantics. In fact, with boolean rules, each state has a set of enabled transitions, leading to other states; the activation of a rule only depends on its condition being true or false. In our probabilistic setting, each Murφ rule defines a pair (p, f) of the PRBTS being defined.

Finally, we modify the `invariant` nonterminal production of the Murφ language grammar (Chapter 7 of the documentation [18]) as follows:
`<invariant> ::= invariant [ <string> ] <realexpr> <booleanexpr>`
where `<realexpr>` has to be a real valued expression in $[0, 1]$, while `<booleanexpr>` has to be a boolean valued expression.

```
type real_type : real(4, 10);
var x : real_type;

startstate "init"  begin x := 1.0; end;

rule "reset" (x = 0.0? 1.0 : 0.0) ==> begin x := 1.0; end;
rule "beetwen 0 and x" (x > 0.0? x : 0.0) ==> begin x := x/10; end;
rule "beetwen x and 1" (x > 0.0? 1.0 - x : 0.0) ==> begin x := (1.0 + x)/2.0; end;

invariant "never reaches 0.0" 0.0 (x != 0.0)
```

Fig. 1. An example of FHP-Murφ input file

In FHP-Murφ the invariant statement `invariant` p γ requires that with probability at least p the following holds: "all states reachable in at most k steps from an initial state satisfy γ" (k is FHP-Murφ horizon).

This is equivalent to say that the probability of reaching in at most k steps from an initial state a state not satisfying γ is less than $(1 - p)$.

6.2 A Toy Example

Consider the SP $\mathcal{S}$ defined as follows. Initially $\mathcal{S}$ is in state 1. If $\mathcal{S}$ is in a state $x > 0$, then with probability x $\mathcal{S}$ moves to state $x/10$, and with probability $(1 - x)$ $\mathcal{S}$ moves to state $(1 + x)/2$. If $\mathcal{S}$ is in state 0 then $\mathcal{S}$ deterministically moves to state 1. In Fig. 1 we give the FHP-Murφ definition for $\mathcal{S}$.

The FHP-Murφ invariant in Fig. 1 requires that, with probability at least 0.0 (i.e. always), in all the states, that are reachable in at most k transitions (horizon), $x \neq 0$ holds. That is, the probability that we reach, within horizon k, state 0, is less than 0. That is, state 0 is not reachable in $\mathcal{S}$.

From definition of $\mathcal{S}$ should be quite clear that indeed state 0 is not a reachable state for $\mathcal{S}$. However, since we are using finite precision real numbers, state 0 may be reached because of numerical approximations.

In Fig. 1, since the precision of x is 10^{-9} (with this precision, we have $10^{-10} = 0$), we will reach the state 0 if the horizon is a $k \geq 10$. For example, if $k = 10$, then the probability to reach state 0 is 10^{-45}.

7 Two Protocols in FHP-Murφ

In this Section we show how FHP-Murφ (*Finite Horizon Probabilistic* Murφ) [22], a suitable *disk based* extension of the Murφ verifier [18], can be used for automatic *Finite Horizon Verification* of PRBTS.

More specifically, we give two examples of our approach describing the behavior of two different queueing systems, showing their implementation in FHP-Murφ and sketching why they are more naturally described in FHP-Murφ than in PRISM.

Both examples describe queue systems with a certain probability that an element in the queue decides to leave its slot without having being served. This results in an error state.

7.1 A Length-Based Queue System

The first system models a "Length-Based" Queue System (LQS in the following), and it has the dynamics described below. In a generic state s, the following moves are allowed:

1. An enqueue operation. This operation is possible only if the queue is not full;
2. A dequeue operation. This operation is possible only if the queue is not empty;
3. Each element in the queue can leave its slot (this results in an error state);
4. The system may remain in the same state s.

The probabilities of the preceding moves are as follows. Let n be the number of queue slots. Suppose that, in state s, h operations are allowed. We have that $1 \leq h \leq 3 + n$, since each of the at most n elements in the queue can go in an error state. Then the probability of the first two moves (if they are allowed) is $\frac{1}{h}$. The probability that a queue element i enters an error state is $\frac{1-e^{-j}}{h}$, where j is the number of elements preceding i in the queue (i.e. the number of dequeue operations that i must wait for before it is its turn). This means that the more elements preceding i, the higher the probability that i leaves the queue. Finally, the probability that no operation is performed is the complement to 1 of the sum of the other defined probabilities.

The implementation of such a system in FHP-Murφ is quite simple. The queue is modeled with a circular array managed by two pointers, `head` and `tail`. For each slot in the queue, we memorize if it is in a correct state or in an error state (i.e. the element has left).

In Figure 2 we show the two main functions, `prob_trans` and `make_trans`, and how they are called by the `rule ''main''`.

Function `prob_trans` returns the outgoing probabilities from the current state s. The parameter `i` is needed to identify which of the moves allowed in s is the one to be calculated. Note that the function $\frac{1-e^{-j}}{h}$, where j is the number of elements preceding an element in the queue, is calculated by the function `prob_err`.

Function `make_trans` changes state s so as to generate a next state. It uses the parameter `i` in the same manner as `prob_trans`.

The `ruleset` in Figure 2 calls the `rule ''main''` with the different values for the variable `i` which are needed in functions `prob_trans` and `make_trans`.

Finally, the invariant to be checked states that the probability of the event "for all states s that are reachable in a finite number of steps k, s is not an error state" must be at least 0, where k is a parameter of the verification. Having set the probability to be $p \geq 0$ (which is always true) forces FHP-Murφ to always reach the horizon k (if we had set it to be $p \geq \gamma$, with $0 < \gamma \leq 1$, the visit would have stopped when p had become less than γ).

```
function prob_trans(i : trans_possib_type) : real_type; begin
  tmp := 0; /* number of moves except enqueue and dequeue */
  trans_possib := 1; /* total number of possible moves */
  calc_trans_possib(trans_possib, tmp);
  if (i >= trans_possib) then return 0.0;
/* i ranges on the max transitions number,
whilest they are not always all possible */
  else
    if (i < tmp) then return 1.0/trans_possib;
    else if (i = trans_possib - 1) then return 1.0/trans_possib - sum_prob_prec();
    else return prob_err(i - tmp)/trans_possib;
    endif; endif; endif; end;

procedure make_trans(i : trans_possib_type); begin
  /* the first part is the same as prob_trans */
  tmp := 0; trans_possib := 1; calc_trans_possib(trans_possib, tmp);
  if (i<trans_possib) then /* now, instead of giving probabilities, moves are done */
    if (!queue_empty() & i = 0) then /* dequeue */
      q[head] := noerr;
      if (head = ITEM_Q - 1) then head := 0;
      else head := head + 1; endif;
    else if (!queue_full() & (tmp = 1 ? i = 0 : i = 1)) then /* enqueue */
      q[tail] := noerr;
      if (tail = ITEM_Q - 1) then tail := 0;
      else tail := tail + 1; endif;
    else if (i != trans_possib - 1) then /* gone away */
      q[i - tmp] := err;
    endif; endif; endif; endif; end; /* if i = trans_possib - 1 no action is done */

ruleset i : trans_possib_type do /* general rule for the whole system */
  rule "main" prob_trans(i) ==> begin make_trans(i); end; end;

invariant "queue ok" 0.0 forall i : queue_range do q[i] != err endforall);
```

Fig. 2. FHP-Murφ implementation sketch for LQS

7.2 A Time-Based Server-Queue System

The second system models a "Time-Based" Server-Queue System (TSQS in the following), and it has the sequent behavior. In a generic state s, there are two different set of allowed moves. The first set just consists of the enqueue, the dequeue, the server status change and the null operations, with uniform probability.

The server status is given by a counter ranging from 0 to `MAX_COUNT_S`, modeling the time of service. If the server counter is 0, the server is free, then a dequeue (on a nonempty queue) can be made. In this case, the server counter is set to `MAX_COUNT_S`. If the server counter is greater than 0, then it is reset to 0 with probability proportional to the current server counter, and it is simply decremented with a complementary probability.

This models the fact that the higher the time of service, the higher the probability of returning free.

The second set of moves consists in updating a counter associated to each element in the queue, modeling the time spent by the element in the queue. When this counter reaches a given maximum value (`MAX_COUNT_Q`), we are in an error state. The updating phase consist in $n + 1$ possible transitions, where n is the number of elements currently in the queue: each of the element counters can immediately reach `MAX_COUNT_Q` with probability directly proportional to

```
function prob_trans(i : trans_possib) : real_type;
begin
  num_trans_possib := 1; calc_trans_possib(trans_possib);
  if (i >= num_trans_possib) then return 0.0;
  else /* mod_glob distinguish the two set of moves */
    if (mod_glob = 0) then
      if (s > 0 & i < 2) then
        if (i = 1) then return (s/MAX_COUNT_S)/(num_trans_possib - 1);
        else return (1.0 - s/MAX_COUNT_S)/(num_trans_possib - 1);
        endif;
      else return 1.0/(s > 0? num_trans_possib - 1 : num_trans_possib);
      endif;
    else
      if (i!=num_trans_possib-1) then return (q[slot(i)]/MAX_COUNT)/num_trans_possib;
      else return 1.0/trans_possib - sum_prob_prec();
      endif; endif; endif;
end;

procedure make_trans(i : trans_possib);
begin
  num_trans_possib := 1; calc_trans_possib(trans_possib);
  if (i < num_trans_possib) then
    if (mod_glob = 0) then
      if (s > 0 & i < 2) then s := (i = 1? s - 1 : 0);
      else if (!queue_empty() & s = 0 & i < 1) then
        .   .   .   .   /* dequeue operation */
        s := MAX_COUNT_S;
      else
        if (!queue_full()&(s>0?i<3:(!queue_empty()?i<2:i<1))) then
        .   .   .   .   /* enqueue operation */
        endif; endif; endif;
    else
      if (i != num_trans_possib - 1) then
        /* function slot(i) return the i-th element in the queue */
        q[slot(i)] := MAX_COUNT;
      endif;
      for k : queue_range do
        if (in_queue(k) & q[k] != MAX_COUNT) then q[k] := q[k] + 1;
        endif; endfor;
      /* if i = trans_possib - 1 no action is done */
    endif; endif;
  mod_glob := (mod_glob + 1)%2; /* switch between the two set of moves */
end;

invariant "queue ok" 1.0
(forall i : queue_range do q[i] != MAX_COUNT endforall);
```

Fig. 3. FHP-Murφ implementation sketch for TSQS

the current counter value, while all the other counters are simply incremented. Moreover, the last possibility is that all counters are simply incremented.

This models the fact that the higher the time spent in queue, the higher the probability to go away without being served.

Also the FHP-Murφ implementation of TSQS is simple, and it is sketched in Figure 3. The data structures are essentially the same as in LQS: the only modification consists in maintaining a counter (and not a boolean) for each slot, and in adding a counter to model the server. The structure of the code is the same as in Figure 2, so we only give functions `prob_trans` and `make_trans`.

Note that both these protocols are more difficult to write in PRISM input language. In fact, PRISM only allows constant probabilities to be defined on

ITEM_Q	Horizon	Memory (disk)	Visited	Time	Probability
4	10	0	104	3.900	0.2843699449
4	20	0	264	6.450	0.6043041472
5	10	0	126	3.930	0.3189541147
5	20	0	375	6.790	0.6333385081

Fig. 4. Results for LQS on a INTEL Pentium III 750Mhz with 128MB of RAM. Murφ options: `-b` (bit compression), `-c` (40 bit hash compaction), `-m80` (use 80 MB of RAM). Memory occupations are in MB, time is in seconds.

ITEM_Q	MAX_COUNT_Q	MAX_COUNT_S	Horizon	Memory (disk)	Visited	Time	Probability
5	3	3	10	0	114	13.090	0.595936214
5	3	3	20	0	518	20.850	0.9432926435
10	20	20	30	0	705081	2243.830	0.7360071576
10	20	20	40	139.810176	20072051	65949.160	0.885392219
>10						> 1 day	

Fig. 5. Results for TSCS on a INTEL Pentium III 750Mhz with 128MB of RAM. Murφ options: `-b` (bit compression), `-c` (40 bit hash compaction), `-m200` (use 200 MB of RAM). Memory occupations are in MB, time is in seconds.

transitions. On the other hand, here we have that the transition probabilities depends on the current state. Hence, to implement these protocols in PRISM, we are forced to list the values of the parameters from which they depend (e.g., in LQS, we have to list all the possible values representing the number of elements preceding the current one, asking for each of them if it is the correct value [29]), and then to tabulate, for each of them, the transition probability values. On the opposite, in FHP-Murφ we have been able to describe the transition probabilities in a uniform way.

7.3 Experimental Results

In Figures 4 and 5 we report the results obtained verifying, respectively, LQS and TSCS with FHP-Murφ. For each verification we report the values of the parameters from which the protocol depends (i.e. `ITEM_Q` for LQS, indicating the number of available slots in the queue, `ITEM_Q`, `MAX_COUNT_Q` and `MAX_COUNT_S` for TSCS), the finite horizon of the verification, the memory (on disk), the visited states, the time required by the verification and the final probability (of violating the invariant). Observe that, in TSCS, we were able to cope with quite large numbers of visited states. In fact, being the FHP-Murφ verification algorithm disk-based, almost any verification can be performed, if one waits for a suitable amount of time. This is symbolized by the last row of Figure 5.

8 Conclusions

We presented (Section 3) *Probabilistic Rule Based Transition Systems* (PRBTS) and showed (Section 4) how PRBTS can be used to model a fairly large class of *Finite State Discrete Time Stochastic Processes* as well as *Discrete Time Hybrid Stochastic Processes* (by approximating reals with *finite precision real numbers*).

PRBTS can be used as a *low level* language for *Stochastic Verification Engines*. As an example (Section 5) we showed how a high level definition of a stochastic system based on *Systems of Communicating Stochastic Processes* can be translated into a PRBTS definition.

We showed (Section 7) how FHP-Murφ [22], a suitable *disk based* extension of the Murφ verifier [18] can be used for automatic *Finite Horizon Verification* of PRBTS.

We showed (Section 7) effectiveness of our approach by presenting experimental results on automatic analysis with FHP-Murφ of two nontrivial stochastic systems. Our experimental results show that FHP-Murφ can handle more general models than state-of-the-art *Probabilistic Model Checkers* like PRISM [24,2,16] or TwoTowers [27]. On the other hand PRISM as well as TwoTowers can verify more general properties (e.g. all PCTL [12] properties for PRISM) than FHP-Murφ.

Future works include extending our approach to more general properties than *Finite Horizon Verification*, e.g. PCTL formulas with unbounded `until`. Moreover, it would be interesting to compare our approach with the discounting theory in [30]. In fact, this approach, where the future becomes less and less relevant, seems to fit well with a finite horizon point of view.

Acknowledgments

We are grateful to the anonymous referees for their help in improving a previous version of this paper.

References

1. Krzysztof R. Apt and Ernst-Rudinger Olderog. *Verification of Sequential and Concurrent Programs*. Text and Monographs in Computer Science. Springer–Verlag, 1991.
2. C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. *Automata, Languages and Programming*, pages 430–440, 1997.
3. E. Behrends. *Introduction to Markov Chains*. Vieweg, 2000.
4. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98, 1992.
6. C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Proc. of FOCS'88*, pages 338–345. IEEE CS Press, 1988.
7. C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4):857–907, 1995.
8. L. de Alfaro. Formal verification of performance and reliability of real-time systems. Technical report, Stanford University, 1996.

9. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992.
10. H. Hermanns G.G. Infante Lopez and J. Katoen. Beyond memoryless distribution: Model checking semi-markov chains. In *Proc. of PAPM-PROBMIV*, number 2165 in LNCS, pages 57–70. Springer, 2001.
11. H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. Elsevier, 1994.
12. H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6:512–535, 1994.
13. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991.
14. G. J. Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
15. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In P. Kemper, editor, *Proc. Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 7–12, September 2001. Available as Technical Report 760/2001, University of Dortmund.
16. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with prism: A hybrid approach. In *Proc. TACAS'02*, volume 2280. LNCS, Springer Verlag, April 2002.
17. K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94:1–28, 1991.
18. url: `http://sprout.stanford.edu/dill/murphi.html`.
19. Barry L. Nelson. *Stochastic Modeling: Analysis And Simulation*. Dover Publications, 1995.
20. A. Papoulis. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill Series in System Sciences, 1965.
21. G. Della Penna, B. Intrigila, I. Melatti, M. Minichino, E. Ciancamerla, A. Parisse, E. Tronci, and M. V. Zilli. Automatic verification of a turbogas control system with the murphi verifier. In *Proc. of 6th International Workshop on: Hybrid Systems: Computation and Control (HSCC)*, LNCS, Prague, Czech Republic, April 2003. Springer.
22. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Finite horizon analysis of markov chains with the murphi verifier. *Submitted for publication*, 2003.
23. A. Pnueli and L. Zuck. Probabilistic verification. *Information and Computation*, 103:1–29, 1993.
24. url: `http://www.cs.bham.ac.uk/~dxp/prism/`.
25. R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In *Proc. of CONCUR*, number 836 in LNCS, pages 381–496. Springer, 1994.
26. url: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.
27. url: `http://www.sti.uniurb.it/bernardo/twotowers/`.
28. M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. of FOCS'85*, pages 327–338. IEEE CS Press, 1985.
29. url: `http://www.di.univaq.it/melatti/ICTCS03/`.
30. L. de Alfaro and T. A. Henzinger and R. Majumdar. Discounting the Future in Systems Theory. ICALP 2003: 1022-1037