International Journal on Software Tools for Technology (2006) 8(4/5): 397–409
DOI 10.1007/s10009-005-0216-7

SPECIAL SECTION ON RECENT ADVANCES IN HARDWARE VERIFICATION

Giuseppe Della Penna · Benedetto Intrigila · Igor Melatti · Enrico Tronci · Marisa Venturini Zilli

Finite horizon analysis of Markov Chains with the Murφ verifier

Published online: 4 April 2006

Abstract In this paper we present an explicit disk-based verification algorithm for Probabilistic Systems defining *discrete time/finite state* Markov Chains. Given a Markov Chain and an integer k (horizon), our algorithm checks whether the probability of reaching an error state in at most k steps is below a given threshold. We present an implementation of our algorithm within a suitable extension of the Murφ verifier. We call the resulting probabilistic model checker FHP-Murφ (*Finite Horizon Probabilistic* Murφ). We present experimental results comparing FHP-Murφ with (a finite horizon subset of) PRISM, a state-of-the-art symbolic model checker for Markov Chains. Our experimental results show that FHP-Murφ can handle systems that are out of reach for PRISM, namely those involving arithmetic operations on the state variables (e.g. hybrid systems).

Keywords Automatic verification · Model checking · Markov chains · Probabilistic model checking · Probabilistic verification

1 Introduction

Model-checking techniques [9, 17, 22, 23, 29, 34] are widely used to verify the correctness of digital hardware, embedded software and protocols by modeling such systems as *Nondeterministic Finite State Systems* (NFSSs).

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 NFSSs, but they can often be modeled by *Markov Chains* [3, 19]. Roughly speaking, a Markov Chain can be seen as an automaton labelled with (outgoing) probabilities on its transitions.

For stochastic systems, correctness can only be stated using a probabilistic approach, e.g. using a *Probabilistic Logic* (e.g., [11, 20, 36]). This motivates the development of *Probabilistic Model Checkers* [2, 12, 24], i.e. model-checking algorithms and tools whose goal is to automatically verify (probabilistic) properties of stochastic systems (typically Markov Chains). For example, a probabilistic model checker may automatically verify a system property like "the probability that a message is not delivered after 0.1 s is less than 0.80."

Many methods have been proposed for probabilistic model checking (e.g. [4, 11, 14, 20, 21, 26, 31, 33, 36]).

To the best of our knowledge, currently, the state-of-the-art probabilistic model checker is PRISM [2, 25, 32]. PRISM overcomes the limitations due to the use of linear algebra packages in Markov Chain analysis by using *Multi Terminal Binary Decision Diagrams* (MTBDDs) [10] (sometimes also referred as ADDs [1]), a generalization of *Ordered Binary Decision Diagrams* (OBDDs) [8] allowing real numbers in the interval [0, 1] on terminal nodes. More precisely, PRISM can carry out the required Markov Chain analysis using a matrix-based approach (based on linear algebra packages), a symbolic approach (based on the CUDD package [13]) as well as a hybrid approach. The user can choose the best approach for the problem at hand.

Here we are mainly interested in the automatic analysis of *discrete time/finite state* Markov Chains modeling *Discrete Time Hybrid Systems*. Such Markov Chains can in principle be analyzed using PRISM. However, our experience is

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

This paper is a journal version of the conference paper [16].

G. Della Penna (✉) · I. Melatti
Dipartimento di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
E-mail: {dellapenna, melatti}@di.univaq.it

E. Tronci · M. Venturini Zilli
Dipartimento di Informatica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Rome, Italy
E-mail: {tronci, zilli}@di.uniroma1.it

B. Intrigila
Dipartimento di Matematica Pura ed Applicata, Università di Roma "Tor Vergata", Via della Ricerca Scientifica 1, 00133 Rome, Italy
E-mail: intrigil@mat.uniroma2.it

that, using PRISM on our systems, quite soon we run into a *state explosion* problem, i.e. we run out of memory because of the huge OBDDs built during the model-checking process. This is due to the fact that hybrid systems dynamics typically entails many arithmetical operations on the state variables. This makes life very hard for OBDDs, thus making the usage of a symbolic probabilistic model checker (e.g. like PRISM) on such systems rather problematic.

Indeed, our experience shows that *Explicit Model Checking* can outperform *Symbolic Model Checking* in automatic analysis of *Hybrid Control Systems* [15]. This suggested us to explore the possibility of devising an explicit disk-based algorithm for automatic *finite horizon* safety analysis of Markov Chains. In this paper we present our algorithm as well as experimental results showing its effectiveness. Our results can be summarized as follows.

– In Sects. 3 and 4 we present an *explicit* algorithm for automatic verification of discrete time/finite-state Markov Chains. Given a Markov Chain $\mathcal{M}$, our algorithm checks whether the probability of reaching a given state s within k steps is less than a given bound p. Our algorithm is *disk based*, thus, because of the large size of modern hard disks, state explosion is hardly a problem for us. Computation time instead is our bottleneck: our algorithm can trade RAM memory with computation time, i.e. the more RAM available the faster our computation. To the best of our knowledge, this is the first time that such a disk-based algorithm for probabilistic model checking is proposed.
– In Sect. 5, we present an implementation of our algorithm within the Murφ verifier [29]. We call the resulting probabilistic model checker FHP-Murφ (*Finite Horizon Probabilistic* Murφ).
– In Sect. 6.1, we present experimental results comparing FHP-Murφ with PRISM on two suitably modified versions of the dining philosophers protocol included in the PRISM distribution. Our experimental results show that FHP-Murφ can handle systems that are out of reach for PRISM. However, as long as PRISM does not hit state explosion, PRISM is faster than FHP-Murφ (as to be expected).

  Note, however, that PRISM can handle more general models than FHP-Murφ, and can verify more general properties (namely all PCTL [20] properties) than FHP-Murφ. In fact, FHP-Murφ can only verify finite horizon safety properties for Markov Chains, a subclass (although an important one) of the verification tasks that PRISM can handle.
– In Sect. 6.2, we present experimental results on using FHP-Murφ for a probabilistic analysis of a "real-world" hybrid system, namely the Turbogas Control System of the co-generative power plant described in Ref. [15]. Because of the arithmetic operations involved in the definition of the system dynamics, this hybrid system is out of reach for OBDDs (and thus for PRISM), whereas FHP-Murφ can complete the (finite horizon) verification within reasonable time.

2 Basic notation

Let S be a finite set of *states*. We regard functions from S to the real interval [0, 1] and functions from $S \times S$ to [0, 1] as row vectors and as matrices, respectively.

If $\mathbf{x}$ is a vector and $s \in S$ we also write $\mathbf{x}_s$ or $(\mathbf{x})_s$ for $\mathbf{x}(s)$. Similarly, if $\mathbf{P}$ is a matrix and $s, t \in S$ we also write $\mathbf{P}_{s,t}$ or $(\mathbf{P})_{s,t}$ for $\mathbf{P}(s, t)$.

On vectors and matrices we use the standard matrix operations. Namely: $\mathbf{xP}$ is the row vector $\mathbf{y}$ s.t. $\mathbf{y}_s = \sum_{j \in S} \mathbf{x}_j \mathbf{P}_{j,s}$ and $\mathbf{AB}$ is the matrix $\mathbf{C}$ s.t. $\mathbf{C}_{s,t} = \sum_{j \in S} \mathbf{A}_{s,j} \mathbf{B}_{j,t}$. We define $\mathbf{A}^n$ in the usual way, i.e. $\mathbf{A}^0 = \mathbf{I}$, $\mathbf{A}^{n+1} = \mathbf{A}^n \mathbf{A}$, where $\mathbf{I}$ (*the identity matrix*) is the matrix defined as follows: $\mathbf{I}(s, j) =$ *if* $(s = j)$ *then* 1 *else* 0.

Finally, we denote with $\mathcal{B}$ the set $\{0, 1\}$ of Boolean values. As usual 0 stands for *false* and 1 stands for *true*.

We give some basic definitions on Markov Chains; for further details see, e.g. [3]. A *distribution* on S is a function $\mathbf{x} : S \to [0, 1]$ s.t. $\sum_{i \in S} \mathbf{x}(i) = 1$. Thus a distribution on S can be regarded as a $|S|$-dimensional row vector $\mathbf{x}$. A distribution $\mathbf{x}$ *represents* the state $s \in S$ iff $\mathbf{x}(s) = 1$ (thus $\mathbf{x}(i) = 0$ when $i \neq s$). If distribution $\mathbf{x}$ represents $s \in S$, by abuse of language we also write $\mathbf{x} \in S$ to mean that distribution $\mathbf{x}$ represents a state and we use $\mathbf{x}$ in place of the element of S represented by $\mathbf{x}$. In the following we often represent states using distributions. This allows us to use matrix notation to define our computations.

Definition 1 A *Discrete Time Markov Chain* (just *Markov Chain* in the following) is a triple $\mathcal{M} = (S, \mathbf{P}, q)$ where S is a finite set (of *states*), $q \in S$ and $\mathbf{P} : S \times S \to [0, 1]$ is a transition matrix, i.e. for all $s \in S$, $\sum_{t \in S} \mathbf{P}(s, t) = 1$. (We included the *initial state* q in the Markov Chain definition since in our context this will often shorten the notation).

Given the distribution $\mathbf{x}$, the distribution $\mathbf{y}$ obtained by one execution step of Markov Chain $\mathcal{M} = (S, \mathbf{P}, q)$ is computed as $\mathbf{y} = \mathbf{xP}$. In particular, if $\mathbf{x}(s) = 1$ we have that $\forall t[\mathbf{y}(t) = (\mathbf{P})_{s,t}]$.

Definition 2 An *execution sequence* (or *path*) in the Markov Chain $\mathcal{M} = (S, \mathbf{P}, q)$ is a nonempty (finite or infinite) sequence $\pi = s_0 s_1 s_2 \ldots$ where s_i are states and $\mathbf{P}(s_i, s_{i+1}) > 0$, $i = 0, 1, \ldots$

– If $\pi = s_0 s_1 s_2 \ldots$ we write $\pi(k)$ for s_k.
– We write $\pi|k$ for the sequence $s_0 s_1 s_2 \ldots s_{k-1}$.
– The *length* of a path π is denoted with $|\pi|$. 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 ω.
– We denote with Path$(\mathcal{M}, s)$ the set of infinite paths π in $\mathcal{M}$ s.t. $\pi(0) = s$. If $\mathcal{M} = (S, \mathbf{P}, q)$ we write also Path$(\mathcal{M})$ for Path$(\mathcal{M}, q)$.
– We denote with Path$_k(\mathcal{M}, s)$ the set of infinite paths π in $\mathcal{M}$ s.t. $\pi(0) = s$ and $|\pi| = k$. In the same way, Path$_{\leq k}(\mathcal{M}, s)$ is the set of infinite paths π in $\mathcal{M}$ s.t. $\pi(0) = s$ and $|\pi| \leq k$.

Definition 3 Let X be a set. Then a σ-algebra F is a nonempty collection of subsets of X such that the following holds

1. The empty set is in F.
2. If A is in F, then so is the complement of A.
3. If A_n is a sequence of elements of F, then the union of the A_ns is in F.

If S is any collection of subsets of X, then we can always find a σ-algebra containing S, namely the power set of X. By taking the intersection of all σ-algebras containing S, we obtain the smallest such σ-algebra. We call the smallest σ-algebra containing S the σ-algebra generated by S.

Definition 4 For $s \in S$ we denote with $\sum(s)$ the smallest σ-algebra on $\text{Path}(\mathcal{M}, s)$ which, for any finite path ρ starting at s, contains the all the paths in $\{\pi \in \text{Path}(\mathcal{M}, s) | \rho \text{ is a prefix of } \pi\}$.

The probability measure Prob on $\sum(s)$ is the unique measure with

$$\begin{aligned}&\Pr(\{\pi \in \text{Path}(\mathcal{M}, s) | \rho \text{ is a prefix of } \pi\})\\ &\quad= \Pr(\rho) = \prod_{i=0}^{k-1} \mathbf{P}(\rho(i), \rho(i+1))\\ &\quad= \mathbf{P}(\rho(0), \rho(1))\mathbf{P}(\rho(1), \rho(2)) \ldots \mathbf{P}(\rho(k-1), \rho(k))\end{aligned} \tag{1}$$

where $k = |\rho|$.

3 Finite horizon safety verification of Markov Chains

Given a Markov Chain, we want to compute the probability that a path of length k starting from a given initial state q reaches a state s satisfying a given Boolean formula ϕ (i.e., such that $\phi(s) = 1$). If ϕ models an *error condition* the above computation allows us to compute the probability of reaching an error in at most k transitions.

Problem 1 Let $\mathcal{M} = (S, \mathbf{P}, q)$ be a Markov Chain, $k \in \mathbb{N}$, and ϕ be a Boolean function on S. We want to compute $P(\mathcal{M}, k, \phi) = \Pr((\exists i \le k\ \phi(\pi(i))) \mid \pi \in \text{Path}(\mathcal{M}))$, that is the probability of reaching a state satisfying ϕ in *at most* k steps in the Markov Chain $\mathcal{M}$ (starting from $\mathcal{M}$ initial state q).

Definition 5 Let $\mathcal{M} = (S, \mathbf{P}, q)$ be a Markov Chain and let ϕ be a Boolean function on S, i.e. $\phi : S \to \mathcal{B}$. We define the Markov Chain $\mathcal{M}_\phi$ as $\mathcal{M}_\phi = (S, \mathbf{P}_\phi, q)$, where for all $s, t \in S$,

$$\mathbf{P}_\phi(s,t) = \begin{cases} \mathbf{P}(s,t) & \text{if } \neg\phi(s) \\ 1 & \text{if } \phi(s) \wedge (s = t) \\ 0 & \text{if } \phi(s) \wedge (s \neq t) \end{cases} \tag{2}$$

In other words, the Markov Chain $(S, \mathbf{P}_\phi, q)$ is obtained from $(S, \mathbf{P}, q)$ by removing all outgoing edges from any state s satisfying ϕ (error state) and replacing such outgoing edges with just one edge leading back to s. Thus, once an error state is entered there is no way to leave it. This, in turn, means that for $(S, \mathbf{P}_\phi, q)$ the probability of reaching in *exactly* k steps a state satisfying ϕ is exactly the same as the probability of reaching in *at most* k steps a state satisfying ϕ. Note that, according to Definition 1, $(S, \mathbf{P}_\phi, q)$ is indeed a Markov Chain.

From the above considerations it follows that $P(\mathcal{M}, k, \phi)$ can be computed from $\mathbf{P}_\phi$ as shown in Proposition 1. Essentially Proposition 1 is a specialization to our finite horizon case of known results on PCTL Model Checking of Markov Chains (e.g., [2, 20]). In order to give a formal proof of Proposition 1, let us first introduce the following Lemma 2.

Lemma 1 *Let $\mathcal{M} = (S, \mathbf{P}, q)$ be a Markov Chain, let $k \in \mathbb{N}$ be an integer, and let $\phi : S \to \mathcal{B}$ be an atomic proposition. Then,*

$$\begin{aligned}&\text{Prob}\{\pi \in \text{Path}(\mathcal{M}) \mid \exists i \le k : \phi(\pi(i))\}\\ &\quad= \text{Prob}\{\pi \in \text{Path}(\mathcal{M}_\phi) \mid \phi(\pi(k))\}\end{aligned}$$

Proof Note that, for all $k \in \mathbb{N}$, we can limit to finite paths, so $\text{Prob}\{\pi \in \text{Path}(\mathcal{M}) \mid \exists i \le k : \phi(\pi(i))\} = \text{Prob}\{\pi \in \text{Path}_k(\mathcal{M}) \mid \exists i \le k : \phi(\pi(i))\}$ and $\text{Prob}\{\pi \in \text{Path}(\mathcal{M}_\phi) \mid \phi(\pi(k))\} = \text{Prob}\{\pi \in \text{Path}_k(\mathcal{M}_\phi) \mid \phi(\pi(k))\}$. Thus, it is sufficient to prove that

$$\begin{aligned}&\text{Prob}\{\pi \in \text{Path}_k(\mathcal{M}) \mid \exists i \le k : \phi(\pi(i))\}\\ &\quad= \text{Prob}\{\pi \in \text{Path}_k(\mathcal{M}_\phi) \mid \phi(\pi(k))\}\end{aligned}$$

To this end, we will use the following objects:

- $\phi_k(\text{Path}_k(\mathcal{M})) = \{\pi \in \text{Path}_k(\mathcal{M}) \mid \exists i \le k : \phi(\pi(i))\}$; this is the set Path_k without the paths where ϕ never holds;
- $m_k(\phi, \pi) = \min\{0 \le i \le k \mid \phi(\pi(i))\}$, where $\pi \in \phi_k(\text{Path}_k(\mathcal{M}))$; this is a function that, given ϕ and a path π where ϕ holds, returns the index of the first state of π where ϕ holds;
- $\text{Path}_\phi(\mathcal{M}) = \{\rho = s_0 \ldots s_i \mid \exists \pi \in \phi_k(\text{Path}_k(\mathcal{M})) : (\rho \text{ is a prefix of } \pi \wedge i = m_k(\phi, \pi)))\}$; for each path π in $\phi_k(\text{Path}_k(\mathcal{M}))$, this set contains the prefix of π that ends in the first state satisfying ϕ.

To shorten formulas, we define $A = \phi_k(\text{Path}_k(\mathcal{M}))$, $B = \text{Path}_\phi(\mathcal{M})$, $D = \phi_k(\text{Path}_k(\mathcal{M}_\phi))$, $m = m_k(\phi, \pi)$ and $C = \text{Path}_{k - m_k(\phi,\pi)}((S, \mathbf{P}, \pi(m_k(\phi, \pi))))$. Hence we have that

$$\begin{aligned}&\text{Prob}\{\pi \in \text{Path}_k(\mathcal{M}) \mid \exists i \le k : \phi(\pi(i))\}\\ &\quad= \text{Prob}\{\pi \in \text{Path}_k(\mathcal{M}) \mid \pi \in A\}\end{aligned} \tag{3}$$

$$\sum_{\pi \in A} \prod_{j=0}^{k-1} \mathbf{P}(\pi(j), \pi(j+1)) \tag{4}$$

$$\sum_{\pi \in B}\left[\left(\prod_{j=0}^{m-1} \mathbf{P}(\pi(j), \pi(j+1))\right) \cdot \cdot \left(\sum_{\rho \in C} \prod_{h=0}^{k-m-1} \mathbf{P}(\rho(h), \rho(h+1))\right)\right] \tag{5}$$

$$\sum_{\pi \in B} \prod_{j=0}^{m-1} \mathbf{P}(\pi(j), \pi(j+1)) \tag{6}$$

$$\sum_{\pi \in B}\left[\left(\prod_{j=0}^{m-1} \mathbf{P}(\pi(j), \pi(j+1))\right)\left(\prod_{h=m}^{k-1} 1\right)\right] \tag{7}$$

$$\sum_{\pi \in D} \prod_{j=0}^{k-1} \mathbf{P}_{\phi}(\pi(j), \pi(j+1)) \tag{8}$$

$$\text{Prob}\{\pi \in \text{Path}_k(\mathcal{M}_{\phi}) \mid \exists i \leq k : \ \phi(\pi(i))\} \tag{9}$$

$$\text{Prob}\{\pi \in \text{Path}_k(\mathcal{M}_{\phi}) \mid \phi(\pi(k))\} \tag{10}$$

In the step from (3) to (4) we used Definition 4, while in transforming (3) to (6) we used the fact that for all $s \in S$, $n \in \mathbb{N}$ and $1 \leq k \leq n$,

$$\sum_{\pi \in \text{Path}_n(\mathcal{M}, s)} \prod_{j=0}^{k-1} \mathbf{P}(\pi(j), \pi(j+1)) = 1$$

Moreover, the final equivalence (from (9) to (10)) is due to Definition 5: in fact, a $\pi \in \text{Path}_k(\mathcal{M}_{\phi})$, when reaches a state s in which ϕ holds, is forced to infinitely cycle on s with probability 1 (every other transition from s has probability 0, so it is not activated). □

Proposition 1 *Let* $\mathcal{M} = (S, \mathbf{P}, \mathbf{q})$, *and let* ϕ *be a Boolean function on* S. *Then* $P(\mathcal{M}, k, \phi) = \Pr((\exists i \leq k\ \phi(\pi(i))) \mid \pi \in \text{Path}(\mathcal{M})) = \sum_{s \mid \phi(s)} (\mathbf{q}\mathbf{P}_{\phi}{}^{k})_s$

Proof By Lemma 1, it is sufficient to prove that

$$\text{Prob}\{\pi \in \text{Path}(\mathcal{M}_{\phi}) \mid \phi(\pi(k))\} = \sum_{s \mid \phi(s)} (\mathbf{q}\mathbf{P}_{\phi}{}^{k})_s$$

From Markov Chain theory [3], we have that, for all $s \in S$, $\text{Prob}\{\pi \in \text{Path}(\mathcal{M}_{\phi}) \mid \pi(k) = s\} = (\mathbf{q}\mathbf{P}_{\phi}{}^{k})_s$. So, we have that

$$\begin{aligned}
&\text{Prob}\{\pi \in \text{Path}(\mathcal{M}_{\phi}) \mid \phi(\pi(k))\} \\
&\quad = \text{Prob}\{\pi \in \text{Path}(\mathcal{M}_{\phi}) \mid \vee_{s \mid \phi(s)}\, \pi(k) = s\} \\
&\quad = \sum_{s \mid \phi(s)} \text{Prob}\{\pi \in \text{Path}(\mathcal{M}_{\phi}) \mid \pi(k) = s\} \\
&\quad = \sum_{s \mid \phi(s)} (\mathbf{q}\mathbf{P}_{\phi}{}^{k})_s
\end{aligned}$$

□

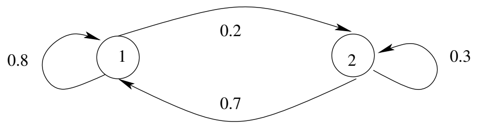

Fig. 1 A Markov Chain

Example 1 Consider the Markov Chain $\mathcal{M} = (S, \mathbf{P}, \mathbf{q})$ with $S = \{1, 2\}$, $\mathbf{P} = [\begin{smallmatrix} 0.8 & 0.2 \\ 0.7 & 0.3 \end{smallmatrix}]$ and $\mathbf{q} = [1\ 0]$ (i.e. distribution $\mathbf{q}$ denotes state 1). The usual automata-like representation for $\mathcal{M}$ is given in Fig. 1. Let ϕ be defined as follows: $\phi(s) = (s = 2)$, i.e. only state 2 satisfies ϕ.

Then $\mathbf{P}_{\phi} = [\begin{smallmatrix} 0.8 & 0.2 \\ 0.0 & 1.0 \end{smallmatrix}]$ and from Proposition 1 we have: $P(\mathcal{M}, 1, \phi) = 0.2$, $P(\mathcal{M}, 2, \phi) = 0.36$, $P(\mathcal{M}, 3, \phi) = 0.488$.

4 Probabilistic finite-state systems

The Markov Chain definition given in Definition 1 is appropriate to study mathematical properties of Markov Chains. However, Markov Chains arising from probabilistic concurrent systems are usually defined using a suitable programming language rather than a stochastic matrix. As a matter of fact the (huge) size of the stochastic matrix of concurrent systems is one of the main obstructions to overcome in probabilistic model checking.

A Markov Chain is presented to a model checker by defining (using a suitable programming language) a *next state* function that returns the needed information about the immediate successors of a given state. The following definition formalizes this notion.

Definition 6 A *Probabilistic Finite State System* (PFSS) $\mathcal{S}$ is a 4-tuple $(S, q, \mathcal{A}, \texttt{next})$, where S is a finite set (of states), $q \in S$, $\mathcal{A}$ is a finite set of labels and $\texttt{next}$ is a function taking a state s as argument and returning a set $\texttt{next}(s)$ of triplets $(t, a, p) \in S \times \mathcal{A} \times [0, 1]$ s.t. $\sum_{(t,a,p) \in \texttt{next}(s)} p = 1$.

We can associate a Markov Chain to a PFSS in a unique way.

Definition 7 1. Let $\mathcal{S} = (S, q, \mathcal{A}, \texttt{next})$ be a PFSS. The Markov Chain associated to $\mathcal{S}$ is $\mathcal{S}^{mc} = (S, \mathbf{P}, q)$, where $\mathbf{P}(s, t) = \sum_{(t,a,p) \in \texttt{next}(s)} p$.

2. Given $k \in \mathbb{N}$ and a Boolean function ϕ on S we write $P(\mathcal{S}, k, \phi)$ for $P(\mathcal{S}^{\text{mc}}, k, \phi)$ as defined in Problem 1.

Problem 1 for PFSSs becomes: given a PFSS $\mathcal{S}$, compute $P(\mathcal{S}, k, \phi)$. We want to compute $P(\mathcal{S}, k, \phi)$ without generating the transition matrix for Markov Chain $\mathcal{S}^{\text{mc}}$: using Proposition 1 this can be done as shown in Proposition 2. To prove Proposition 2, let us first introduce the following Lemma 2.

```
/* For i = 1, ... k + 1, Q(i) is a queue of
  state-probability pairs (s, p) */
P((S, q, 𝒜, next), k, ϕ) {
 if ϕ(q)
  return(1);
 i = 0; r = 0;
 Enqueue(Q(i), (q, 1));
 foreach i = 0 ... k - 1 do {
  /* BF level i begins */
  while(Q(i) is nonempty) {
   (s, p) = Dequeue(Q(i));
   foreach (s′, a, p′) in next(s) do {
    if ϕ(s′) {
     r = r + p*p′;
    } else Enqueue(Q(i + 1), (s′, p*p′));
   }
  }
 }
 return(r);
}
```

Fig. 2 Computation of $P(\mathcal{S}, k, \phi)$

Lemma 2 *Let $\mathcal{S} = (S, q, \mathcal{A}, \mathtt{next})$ be a PFSS, $k \in \mathbb{N}$ be a nonnegative integer and ϕ be a Boolean function on S such that $\phi(q) = 0$. Finally, let* `P` *be the function that computes $P(\mathcal{S}, k, \phi)$ as in Fig. 2. Then, for all $0 \le i \le k$ the following holds.*

At the beginning of level i in function $\mathtt{P}(\mathcal{S}, k, \phi)$, `Q(i)` *contains n pairs $(s_1, p_1), \ldots, (s_n, p_n)$, where n is the number of the paths $\pi_1, \ldots, \pi_n$ such that, for all $1 \le m \le n$, π_m satisfies the following properties:*

1. *$|\pi| = i$;*
2. *$\pi(0) = q$;*
3. *$\pi(i) = s_m$;*
4. *$\mathbf{P}(\pi) = p$;*
5. *for all $0 \le j \le i$, $\phi(\pi(j)) = 0$.*

Proof The proof is done by induction on i. As induction basis we have that, at the beginning of BF level 0, `Q(i)` contains only the pair $(q, 1)$, and in fact the only path of length 0 starting from q is q itself. Moreover, $\phi(q) = 0$ and $\mathbf{P}(q) = 1$ (i.e. the path consisting only of q has probability 1).

As induction step, we suppose our statement valid for i, and we prove it for $i + 1$. To this end, let $\pi_1, \ldots, \pi_n$ be the paths of length $i + 1$ starting from q such that, for all $1 \le m \le n$ and $0 \le j \le i + 1$, $\phi(\pi_m(j)) = 0$. Moreover, let `Q(i)` be the queue at the beginning of level i.

By induction hypothesis we have that, for all $1 \le m \le n$, the pairs $(\pi_m(i), \mathbf{P}(\pi_m|(i + 1)))$ (containing the last but one state of π_m together with the probability of π_m without the last state) were stored in the queue `Q(i)`. During the level i computation, these pairs are dequeued and expanded to create the queue `Q(i+1)`. Now, suppose by absurd that exists a $1 \le m \le n$ such that $(\pi_m(i + 1), \mathbf{P}(\pi_m))$ is not enqueued in `Q(i+1)` during the level i computation. Since, by induction hypothesis, $(\pi_m(i), \mathbf{P}(\pi_m|(i+1)))$ is on `Q(i)`, we have that $(\pi_m(i+1), \mathbf{P}(\pi_m))$ will be eventually enqueued in `Q(i+1)`. In fact, $\mathbf{P}(\pi_m) = \mathbf{P}(\pi_m|(i + 1))\mathbf{P}(\pi_m(i), \pi_m(i + 1))$ and `P` makes exactly this computation, with $\mathtt{p} = \mathbf{P}(\pi_m|(i + 1))$ (it is taken from `Q(i)`) and $\mathtt{p}' = \mathbf{P}(\pi_m(i), \pi_m(i + 1))$. Moreover, no error states are enqueued at this level, so by induction hypothesis for all $1 \le m \le n$ and $0 \le j \le i + 1$, $\phi(\pi_m(j)) = 0$.

On the other hand, suppose by absurd that a pair $(\overline{s}, \overline{p})$ is enqueued in `Q(i+1)`, but for all $1 \le m \le n$ $(\overline{s}, \overline{p}) \ne (s_m, p_m)$. By induction hypothesis, `Q(i)` contains only pairs $(\pi_m(i), \mathbf{P}(\pi_m|(i+1)))$ satisfying conditions 1–5. Thus, since only the states in `Q(i)` are expanded, we have that there exists $1 \le m_1, \ldots, m_h \le n$ such that $\overline{s} = \pi_{m_l}(i+1) = s_{m_l}$ for all $1 \le l \le h$. Hence, $\overline{p} \ne p_{m_l} = \mathbf{P}(\pi_{m_l})$ has to hold for all $1 \le l \le h$. However, take a $1 \le l \le h$; then, $\overline{p}$ is computed as $\mathtt{p} * \mathtt{p}' = \mathbf{P}(\pi_{m_l}|(i + 1))\mathbf{P}(\pi_{m_l}(i), \pi_{m_l}(i + 1)) = \mathbf{P}(\pi_{m_l}) = p_{m_l}$ (where the last equivalence holds for the proof of the first part of this lemma). This contradiction ends the proof. □

Proposition 2 *Let $\mathcal{S} = (S, q, \mathcal{A}, \mathtt{next})$ be a PFSS, $k \in \mathbb{N}$ and ϕ be a Boolean function on S. Then $P(\mathcal{S}, k, \phi)$ can be computed as shown in Fig. 2.*

Proof Let $\mathcal{M} = \mathcal{S}^{mc}$. By Proposition 1, we have to show only that the final value of `r` is $\sum_{s \mid \phi(s)} (\mathbf{qP_\phi}^k)_s$. To this end, let

- $\Pi(s, k, r) = \{\pi \in \mathrm{Path}_k(\mathcal{M}_\phi, s) \mid \pi(k) = r\}$, and
- $\Pi(s, \phi, k, r) = \{\pi \in \mathrm{Path}_{\le k}(\mathcal{M}, s) \mid \exists\rho \in \mathrm{Path}_k(\mathcal{M}_\phi, s)\exists 0 \le i \le k : \rho(i) = r \wedge \forall j < i\ \rho(j) \ne r \wedge \pi = \rho|(i + 1) = \rho_0 \ldots \rho_i\}$

(i.e., $\Pi(s, \phi, k, r)$ contains all the paths of length at most k leading from s to r without going through error states). Then, we have that

$$\sum_{s \mid \phi(s)} (\mathbf{qP_\phi}^k)_s = \tag{11}$$

$$\sum_{s \mid \phi(s)} \sum_{\pi \in \Pi(q,k,s)} \mathbf{P}_\phi(\pi) = \tag{12}$$

$$\sum_{s \mid \phi(s)} \sum_{\pi \in \Pi(q,\phi,k,s)} \mathbf{P}_\phi(\pi) = \tag{13}$$

$$\sum_{s \mid \phi(s)} \sum_{\pi \in \Pi(q,\phi,k,s)} \mathbf{P}(\pi) \tag{14}$$

Here, the step from (11) and (12) is a known property of Markov Chains [3], whereas step from (13) and (14) is due to the definition of $\mathbf{P}_\phi$ (Definition 5).

To complete the proof, we only have to observe that, by Lemma 2, (14) is exactly the computation carried out for the variable `r` in function `P`. In fact, the queue contains, for each level i, the paths in $\{\pi \in \Pi(q, \phi, i, s) \mid |\pi| = i \text{ and } \phi(s) = 0\}$, while the ones in $\{\pi \in \Pi(s, \phi, i, r) \mid \exists 1 \le j \le i : \phi(\pi(j)) = 1\}$ are used to increment `r`. □

Remark 1 Given a PFSS $\mathcal{S} = (S, q, \mathcal{A}, \texttt{next})$, $k \in \mathbb{N}$, a Boolean function ϕ on S and a probability threshold p, in Sect. 5, exploiting Proposition 2, we will present an efficient disk-based algorithm to check if it holds that $P(\mathcal{S}, k, \phi) < p$. In other words, our algorithm checks the validity of *Finite Horizon Probabilistic* (FHP) *Safety Properties*, which are a very important class of properties. This motivates our disk-based algorithm.

Of course a FHP safety property can be easily defined with a PCTL [20] formula, namely $P_{<p}[\text{true}\ U^{\leq k}\phi]$, thus also the probabilistic model checker PRISM [32] can be used to verify such properties. Note, however, that PRISM can handle *all* PCTL formulas, whereas our algorithm can only handle FHP safety properties. In particular, PRISM can verify *unbounded horizon* properties like $P_{<p}[\text{true}\ U\phi]$ (*the probability of reaching a state satisfying ϕ is less than p*), which cannot be handled with our algorithm.

```
/* k is the horizon, i.e. the max allowed
 number of steps to reach an error state
 */
int k;
bool Phi(state s);
state_prob_label_triplets next(state s);
FIFO_Queue Q_old, Q_new;
Cache M;
/* prob_Phi incrementally stores the
 probability of reaching an error state in
  at most k steps */
double prob_Phi;
/* max_prob_Phi is the max allowed value
 for prob_Phi */
double max_prob_Phi;
```

Fig. 3 Functions and data structures

5 Analyzing probabilistic systems with the Murφ verifier

Building on the computation scheme in Fig. 2, in the following we describe an efficient disk-based algorithm to verify FHP-safety properties, as well as an implementation of such an algorithm within the Murφ verifier. We call the resulting tool FHP-Murφ (*Finite Horizon Probabilistic* Murφ).

5.1 Functions and data structures

The FHP-Murφ input defines a PFSS $\mathcal{S} = (S, q, \mathcal{A}, \texttt{next})$ to which we will refer in the sequel. The FHP-Murφ input language is the same as the Murφ one [29], only FHP-Murφ has probabilities rather than Booleans on rule guards.

In particular, the FHP-Murφ input language allows to define the Boolean function ϕ on S, the probability threshold β s.t. $P(\mathcal{S}, k, \phi) < \beta$ must hold (Remark 1), and the initial state q of $\mathcal{S}$. Note that Murφ can have a set of initial states; however, without loss of generality, in the following we assume to have just one initial state.

Figure 3 shows the declarations of the functions and data structures used in FHP-Murφ:

- The constant `k` (implementing k) is our verification horizon and is given to FHP-Murφ as a command line parameter.
- The function `Phi()` implements ϕ.
- The function `next()` is the nextstate function of the PFSS $\mathcal{S}$, defined by FHP-Murφ input, which takes a state s as argument and returns the set $\texttt{next}(s)$ of triplets (t, a, p) s.t. s goes to t with probability p and label a.
- The queues `Q_old` and `Q_new` are used to store distributions, thus queue elements are pairs (s, p) where s is a state and p is the probability of reaching s from the initial state of $\mathcal{S}$. Such queues play, respectively, the same role as queues $Q(i)$ and $Q(i + 1)$ in the while loop of Fig. 2. Note that queues `Q_old` and `Q_new` are the only place in which *state explosion* may occur in our algorithm. For this reason we implement them on disk analogously to [35]. This allows us to handle fairly large state spaces.
- The cache `M` contains pairs (s, p) as for queues `Q_old` and `Q_new`. Note that, however, only s is used to determine the cache entry in which a particular pair (s, p) has to be stored. In the following, we use the expression `M[h]` to refer to the pair (s, p) stored in entry `h` of `M`. In particular, we write `M[h].state` to denote s and `M[h].prob` to denote p.
- `prob_Phi` accumulates the probability of reaching an error state (i.e. a state s s.t. `Phi(`s`) = true`) and is incremented every time an error state is reached.
- Constant `max_prob_Phi` (implementing β) defines our probability threshold, i.e. the max allowed value for the probability `prob_Phi`.

5.2 Functions `Search()` and `Insert()`

The main function `Search()`, shown in Fig. 4, efficiently implements the computation described in Fig. 2.

Function `Insert()`, also shown in Fig. 4, uses a cache table `M` in RAM to save queue space and thus computation time. Every time it is necessary to enqueue a new pair (state s, probability p), `Insert(s, p)` is called. If a pair (s, p') is already stored in cache `M`, we simply update the stored probability p' in `M`, adding p to it; otherwise, we store (s, p) in `M`. If this causes a collision (i.e., if the slot of `M` in which we have to put (s, p) is already occupied), we call function `Checktable()` to empty `M` and thus free the needed cache slot.

If we were not using `M`, for each state s at level i we would have w copies of s in the queue, where w is the number of paths of length i leading to state s from initial state q, whileas using `M` our queue contains one or slightly

```
int Search() {
 if (Phi(q))
  return(1);
 prob_Phi = 0;
 /* enqueue initial state q */
 Enqueue(Q_old, (q, 1));
 for (level = 1; level <= k; level++) {
  clear cache table M;
  while (Q_old is not empty) {
   (s, p) = Dequeue(Q_old);
   foreach (s', a, p') in next(s) {
    if (Phi(s')) {
     prob_Phi = prob_Phi + p*p';
     if (prob_Phi >= max_prob_Phi)
     /* property does not hold */
      return(0);
    } else Insert(s', p*p');
   } /* foreach */
  } /* while */
  /* level terminated, Q_old is empty  */
  Checktable();
  swap Q_new with Q_old ;
  /* now, Q_new is empty */
 } /* for */
 return(1); /* property holds */
}  /* Search() */

Insert(state s, double p) {
 if (s is in M) {
  h = hash(s);
  prob = M[h].prob + p;
  /* new probability of s is prob */
  M[h] = (s, prob);
 } else {
  collision = Insert_in_table(s, p);
  if (collision) {
   Checktable();
   /* there is space to insert now */
   Insert_in_table(s, p);
  } /* if collision */
 } /* else */
} /* Insert() */
```

Fig. 4 Functions `Search()` and `Insert()`

more than one (depending on how large is `M`) copy of s. This saves queue space as well as computation time. Hence, the more RAM available for `M`, the less our duplicated states, queue sizes, number of states to be explored and, finally, our computation time. For this reason `M` should be as large as possible.

5.3 Functions `Insert_in_table()` and `Checktable()`

Function `Insert_in_table()`, shown in Fig. 5, tries to insert a (state s, probability p) pair in the cache `M`.

```
bool Insert_in_table(state s, double p) {
 h = hash(s);
 if (M[h] is free) {
  M[h] = (s, p);
  return true;
 } else return (M[h].state == s);
} /* Insert_in_table() */

Checktable() {
 move M in Q_new and clear M;
} /* Checktable() */
```

Fig. 5 Functions `Insert_in_table()` and `Checktable()`

`Insert_in_table(s,p)` first calculates the hash value `h` of `s`. Then, if `M[h]` is a free cache slot, the function inserts `s` and `p` in `M[h]` and returns true. If `M[h]` is not free, `Insert_in_table()` returns false without inserting `s` and `p` in `M`.

Function `Checktable()`, also shown in Fig. 5, simply flushes `M` into `Q_new`. It is actually the only function that enqueues values in `Q_new` and is used by function `Insert()` to free `M` when a collision occurs. `Checktable()` is also called at the end of the `while` in function `Search()` (Fig. 4) to enqueue in `Q_new` the states visited after the last call to function `Insert()`, so that all states reached in the current level will be expanded in the next one.

6 Experimental results

To show the effectiveness of our approach we run two kind of experiments. First, in Sect. 6.1, we compare FHP-Murφ with the probabilistic model checker PRISM [32]. Second, in Sect. 6.2, we run FHP-Murφ on a quite large probabilistic hybrid system. Since our main goal is to use FHP-Murφ on hybrid systems, this second kind of evaluation is very interesting for us.

All the experiments presented were carried out on a Dual-Pentium III 500 MHz machine with 2GB of RAM, using FHP-Murφ bit compression (option `-b`, which allocates to each state variable the smallest number of bits needed to hold its possible values) and PRISM default options.

6.1 Probabilistic dining philosophers

In this section, we give our experimental results on using FHP-Murφ on the probabilistic protocols included in the PRISM distribution [32]. We do not consider the protocols that lead to Markov Decision Processes or to Continuous Time Markov Chains, since FHP-Murφ cannot deal with them. Hence we only consider Pnueli and Zuck [30] and Lehmann and Rabin [27, 28] probabilistic dining philosophers protocols.

```
module phil1
 p1: [0..10] init 0;
 .   .   .   .   .
 [] p1=6 -> (p1'=1);
 [] p1=7 -> (p1'=1);
 .   .   .   .   .
 [] p1=10 -> (p1'=0)
endmodule
```

Fig. 6 Pnueli–Zuck algorithm fragment to be modified in PRISM

```
module phil1
 p1: [0..10] init 0;
 cont1: [0..MAX_CONT] init 0;
 .   .   .   .   .
 [] p1=6 & cont1!=MAX_CONT ->
    (p1'=1) & (cont1'=cont1+1);
 [] p1=6 & cont1=MAX_CONT -> (p1'=1);
 [] p1=7 & cont1!=MAX_CONT ->
    (p1'=1) & (cont1'=cont1+1);
 [] p1=7 & cont1=MAX_CONT -> (p1'=1);
 .   .   .   .   .
 [] p1=10 -> (p1'=0) & (cont1'=0);
endmodule
```

Fig. 7 Pnueli–Zuck algorithm modified fragment in PRISM

Moreover, we modify PRISM definitions for such protocols in order to have a finite horizon property to verify with FHP-Murφ. In fact, FHP-Murφ is unable to verify the PCTL properties for these protocols included in the PRISM distribution, since they are not of the required (*finite horizon probabilistic safety*) form $P_{<p}[\text{true } U^{\leq k}\phi]$. Our modifications to the PRISM protocols consist in adding variables to count the number of times that a philosopher fails in getting both forks before he succeeds. We then verify that these counters are always less than a given maximum threshold (MAX_CONT in the following) with a given probability. This corresponds to verifying *quality of service* properties, which are very frequent in practice. For example, in the Pnueli–Zuck protocol, we changed the code fragment in Fig. 6 with the one in Fig. 7.

The FHP-Murφ definitions for such protocols have been obtained by translating into FHP-Murφ their PRISM (modified) definitions so that for each protocol, FHP-Murφ and PRISM definitions specify exactly the same Markov Chain.

We want to know the probability $P(\texttt{MAX\_CONT}, k)$ of a counter reaching MAX_CONT in at most k (horizon) steps. We set $k = 20$ as our finite horizon (this value occurs in a property of the Lehmann-Rabin protocol in the PRISM distribution [32]).

Figure 8 shows the PCTL property to be verified, stating that the probability that a counter reaches MAX_CONT has to be at most p. We set $p = 1$ since for computing

```
P>=1.0 [true U<=20 ((cont1 = MAX_CONT) |
 (cont2 = MAX_CONT) | (cont3 = MAX_CONT))]
```

Fig. 8 PCTL formula to be verified on the Pnueli–Zuck algorithm in PRISM

```
function calc_prob
    (i : 1..NPHIL; c : 0..10) : prob;
/* probability that p[i] becomes c, NPHIL
 is the number of philosophers */
begin
 switch p[i]
/* p[1] corresponds to PRISM p1, p[2] to
 PRISM p2 etc. */
  .   .   .   .   .
  case 6:
   if (c = 1) then return 1.0 / NPHIL;
   else return 0.0; endif
  case 7:
   if (c = 1) then return 1.0 / NPHIL;
   else return 0.0; endif
  .   .   .   .   .
 endswitch;
end;

ruleset philosophers : 1..NPHIL do
 ruleset next : 0..10 do
  rule "next"
  calc_prob(philosophers, next) ==> begin
   p[i] := c;
/* cont[1] corresponds to PRISM cont1,
 cont[2] to PRISM cont2 etc. */
   if (c = 1 & (p[i] = 6 | p[i] = 7) &
    (cont[i] != MAX_CONT))
    then cont[i] := cont[i] + 1;
   endif;
   if (p[i] = 10 & c = 0) then
    cont[i] := 0;
   endif;
  end;
 end;
end;

invariant "starvation" 0.0
 forall i : 1..NPHIL do
  (cont[i] != MAX_CONT)
 endforall;
```

Fig. 9 Pnueli–Zuck algorithm in FHP-Murφ

$P(\texttt{MAX\_CONT}, k)$ the value of p does not matter. In Fig. 9 we have the corresponding FHP-Murφ code.

FHP-Murφ invariant `invariant p` γ requires that, with probability at least p, "all the states reachable in at most k steps from the initial state satisfy γ" (k is FHP-Murφ horizon). Thus, using the notation of Sect. 5 we have that

Table 1 Results for the modified Pnueli–Zuck protocol

`NPHIL`	`MAX_WAIT`	Probability	Murφ memory	PRISM memory	Murφ time	PRISM time
3	3	7.335194164e-05	200	0.9057	51.970	1.487
3	4	6.883132778e-10	200	1.6844	52.610	2.507
4	3	1.88985976e-06	200	28.1066	242.940	28.72
4	4	2.910383046e-12	200	66.2659	244.170	71.112
5	3	9.164495139e-08	200	916.8246	1408.290	1023.468
5	4	4.194304e-14	200	N/A	1412.210	N/A
8	3	1.210429649e-10	1000	N/A	213790.740	N/A

FHP-Murφ configuration: `-m200` (use 200 Mb of RAM), `-maxl20` (the finite horizon is 20). The last verification had `-m1000` (use 1 Gb of RAM). Memory occupations are in Mb, time is in seconds

Table 2 Results for the modified Lehmann–Rabin protocol

`NPHIL`	`MAX_WAIT`	Probability	Murφ memory	PRISM memory	Murφ time	PRISM time
3	3	4.8039366e-06	800	39.0625	1040.330	84.556
3	4	0.0	800	70.1483	1041.700	121.147
4	3	5.609882064e-08	800	N/A	34307.740	N/A

FHP-Murφ configuration: `-m800` (use 800 Mb of RAM), `-maxl20` (the finite horizon is 20). Memory occupations are in Mb, time is in seconds

$\phi = \neg\gamma$ and the probability threshold (`max_prob_Phi` in Fig. 3) is $(1 - \mathtt{p})$.

Note that in Fig. 9 the probability threshold for FHP-Murφ invariant is 0, so that FHP-Murφ will not stop verification before completing all levels of the BF computation. This forces FHP-Murφ to compute $P(\mathtt{MAX\_CONT}, k)$.

To assess FHP-Murφ effectiveness, in Tables 1 and 2, we compare the results obtained with FHP-Murφ and with PRISM on, respectively, the Pnueli–Zuck and the Lehmann–Rabin protocols (modified as described above), setting $k = 20$ (the finite horizon is 20). In the *PRISM Memory* and *PRISM Time* columns, N/A means that PRISM was unable to complete the verification; in this case, also the options `-m` (totally MTBDD-based verification algorithm) and `-s` (algebraic verification algorithm) have been used, with the same result.

From Table 1 we can see that, for the Pnueli–Zuck algorithm, when `NPHIL` = 5 (5 philosophers) and `MAX_CONT` is 4, PRISM is unable to complete any verification within 2GB of RAM, independently on which of the three PRISM verification algorithms (totally MTBDD-based, algebraic or hybrid) is chosen. Similarly, for the Lehmann-Rabin algorithm, in Table 2 we see that when `NPHIL` is 4 and `MAX_WAIT` is 3 PRISM is unable to complete the verification task in the same environment as above.

FHP-Murφ was always able to complete all the given verifications tasks. However, as it can be seen from Tables 1 and 2, for the verification tasks in which PRISM terminates, PRISM is always faster than FHP-Murφ.

As for the numerical quality of FHP-Murφ, we have that when both PRISM and FHP-Murφ terminate they both give the same value for $P(\mathtt{MAX\_CONT}, k)$ (column *Probability* in Tables 1 and 2).

To sum up, our experimental results show that, for probabilistic protocols involving arithmetical computations, FHP-Murφ has to be considered among the available (and valuable) tools for automatic finite horizon analysis of safety properties.

6.2 Analysis of a probabilistic hybrid system with FHP-Murφ

In this section we show our experimental results on using FHP-Murφ for the analysis of a *real-world* hybrid system, namely, the control system for the gas turbine of a 2 MW *electric co-generative power plant* (*ICARO*) in operation at the ENEA Research Center of Casaccia (Italy).

Our control system (*Turbogas Control System*, TCS in the following) is the heart of ICARO and is indeed its most critical subsystem. Unfortunately, TCS is also the largest ICARO subsystem, thus making the use of model checking for such hybrid system a challenge.

Depending on the operating conditions (e.g. *startup*, *normal*, *shutdown*) ICARO models are widely different. Here we only focus on *normal* operating conditions, i.e. the situation in which ICARO is running at its *nominal* (setpoint) power. In particular, our model cannot be used to study system behaviour during transient operating modes (e.g. at startup or shutdown).

ICARO plant consists of many subsystems. Here we only focus on one of the many subsystems of ICARO (e.g. see [5–7]). Namely we focus on the *Gas Turbine* ICARO subsystem that we call in the following ICARO *Turbogas Control System* (TCS). TCS is the heart of ICARO and is indeed ICARO most critical subsystem. Unfortunately, TCS is also the largest ICARO subsystem, thus making the use of model checking for such hybrid system a challenge.

Unless otherwise stated, all our data (e.g. block diagrams, parameter values, etc) are taken from the ICARO documentation [18].

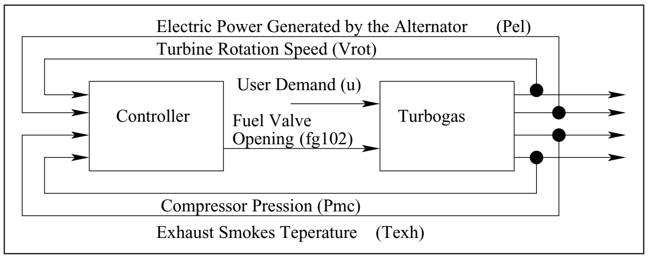
Electric Power Generated by the Alternator (Pel)
Turbine Rotation Speed (Vrot)
User Demand (u)
Controller
Fuel Valve Opening (fg102)
Turbogas
Compressor Pression (Pmc)
Exhaust Smokes Teperature (Texh)

Fig. 10 High-level block diagram of ICARO Turbogas Control System

TCS is a *control system*, that is a (hybrid) system in which we can distinguish two subsystems: the *plant* (i.e. the controlled system) and the *controller* (which sends commands to the plant in order to meet given requirements on the whole system behaviour). Figure 10 shows the high level block diagram for TCS.

The block named *Turbogas* in Fig. 10 models the *Gas Turbine* module. As a matter of fact this module consists of many subsystems (e.g. the compressor, the combustion chamber, the turbine itself and the generator). For our purposes here we can simply look at its input-output model. The module has the following input variables.

– Variable $fg102$ takes value in the real interval [0,1]. This variable gives the opening fraction of the turbogas fuel gas valve (namely valve FG102). It takes value 0 when the valve FG102 is fully closed (no fuel can flow trough the valve) and value 1 when the it is fully opened. This is a *control variable*, i.e. a variable whose value can be chosen by the controller so as to achieve predefined goals.
– Variable u models the *User Demand* of electric power. This variable has to be considered as a *disturbance*, i.e. a variable whose value we (i.e. the controller) cannot choose.

The output variables of the module are the following ones.

– P_{el}, the *Electric power* generated by the alternator.
– V_{rot}, the *Rotation speed* of the gas turbine.
– T_{exh}, the *Temperature* of the exhaust smokes.
– P_{mc}, the *Pressure* of the compressor.

For the purposes of our analysis we used the ODE (*Ordinary Differential Equation*) model, shown in Fig. 11,

$$\begin{aligned}
\dot{P}_{el}(t) &= \alpha_{1,1}P_{el}(t) + \alpha_{1,2}fg102(t) + \alpha_{1,3}u(t)\\
\dot{T}_{exh}(t) &= \alpha_{2,1}T_{exh}(t) + \alpha_{2,2}fg102(t) + \alpha_{2,3}(P_{el}(t) - P^0_{el})\\
&\quad + \alpha_{2,4}(P_{mc}(t) - P^0_{mc})\\
\dot{V}_{rot}(t) &= \alpha_{3,1}V_{rot}(t) + \alpha_{3,2}fg102(t) + \alpha_{3,3}(P_{el}(t) - P^0_{el})\\
P_{mc}(t) &\in [MIN\_P_{mc}, MAX\_P_{mc}]\\
\mid \dot{P}_{mc}(t) \mid &\leq MAX\_D\_P_{mc}\\
u(t) &\in [0, MAX\_U]\\
\mid \dot{u}(t) \mid &\leq MAX\_D\_U
\end{aligned}$$

Fig. 11 Turbogas ODE model used for our analysis

to link turbogas input variables with output variables. Of course such a model is only valid in a neighborhood of the setpoint.

Note that, according to the model in Fig. 11, the compressor pressure P_{mc} can change value *nondeterministically* as long as it satisfies the constraints given in Fig. 11. We do not need a more detailed model here since the compressor pressure is only used as input to the fuel gas valve controller whose requirements do not involve the compressor pressure.

We do not know *in advance* the user demand u. However, by making some hypothesis on the user demand u dynamics we can follow for the user demand model in Fig. 11 the same approach we used for the compressor pressure. Namely, we simply ask that the user demand $u(t)$ be in the interval $[0, MAX\_U]$ (the user demand is always non-negative since users cannot produce electric power) and the *variation speed* of the user demand $\dot{u}(t)$ be at most MAX_D_U. Note that for the model in Fig. 11 the only *input* variable is $fg102$, all other variables (i.e. P_{el}, V_{rot}, T_{exh}, P_{mc}, u) are *state* as well as *output* variables.

The block named *Controller* in Fig. 10 is the fuel gas valve controller of the Turbogas. Figure 12 shows the detailed block diagram for this component.

From Fig. 12 it is clear that the turbogas controller output is obtained as the minimum (block `MIN`) of the outputs of the three subsystems N1 Governor, *Power Limiter*, *Exhaust Temperature Limiter*.

All the controller subsystems are built from the *elementary cell* shown in Fig. 13. In this cell we have the simultaneous presence of linear blocks (e.g. the integrator block labeled $1/s$, *saturation* blocks, *test for* > 0 and logical (`AND`) blocks. This makes the elementary cell a hybrid system. Since all subsystems in TCS are based on such kind of cell, it turns out that TCS itself is a (quite big) hybrid system.

The *N1 Governor* block computes the power demand with the goal of maintaining the turbine rotation speed within given bounds.

The *Power Limiter* block computes the power demand with the goal of maintaining the electric power generated within given bounds.

The *Exhaust Temperature Limiter* block computes the power demand with the goal of maintaining the temperature of the exhaust smokes within given bounds.

The subsystem `MIN` in Fig. 12 computes the minimum among its inputs. Moreover, the block `MIN` returns the name (index) of the winner (i.e. of the input which attained the minimum value) on the output labeled `WINNER`.

The block *Limiter* in Fig. 12 saturates the power demand to 12MW.

The block *Adjust* together with the `OFFSET` parameter in Fig. 12 translates the power demand from the Limiter block into a fuel valve opening command, i.e. into a real number in the range [0,1].

The goal of the turbogas controller is to set the turbogas control variable $fg102$ so as to keep the value of turbogas

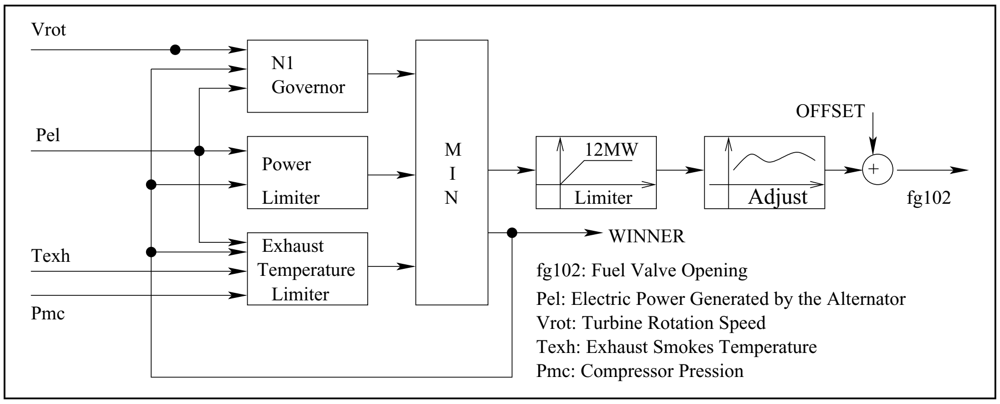
Vrot
Pel
Texh
Pmc
N1 Governor
Power Limiter
Exhaust Temperature Limiter
MIN
12MW
Limiter
Adjust
OFFSET
fg102
WINNER
fg102: Fuel Valve Opening
Pel: Electric Power Generated by the Alternator
Vrot: Turbine Rotation Speed
Texh: Exhaust Smokes Temperature
Pmc: Compressor Pression

Fig. 12 Turbogas fuel gas valve controller

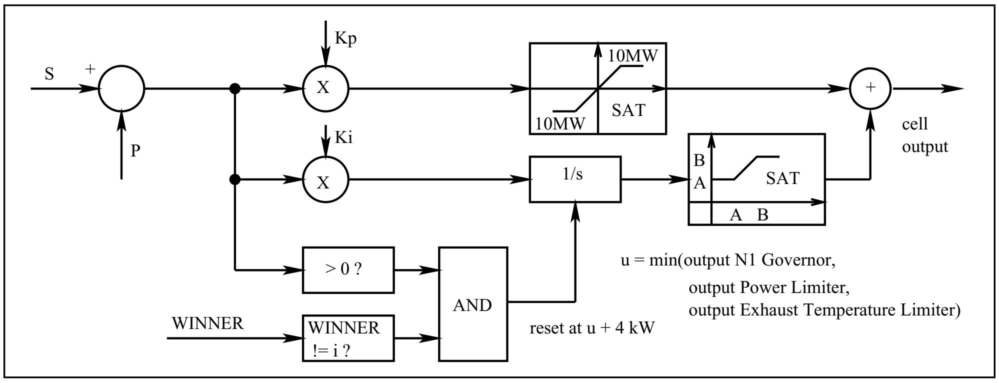
S
P
Kp
Ki
X
10MW
10MW
SAT
1/s
B
A
SAT
A B
cell output
> 0 ?
WINNER
WINNER != i ?
AND
reset at u + 4 kW
u = min(output N1 Governor, output Power Limiter, output Exhaust Temperature Limiter)

Fig. 13 Elementary cell used for the construction of turbogas controller subsystems: N1 Governor, Power Limiter, Exhaust Temperature Limiter. Cell Inputs: `S`, `P`, `WINNER`. Cell Parameters: `i`, `A`, `B`. Known Constants: `Kp`, `Ki` (from [18])

- Electric Power setpoint value: P^0_{el}=2000 (KW).
- Exhaust Smokes Temperature setpoint value: T^0_{exh}=552 (C).
- Turbine Rotation Speed setpoint value: V^0_{rot}=75 (RPM)
- Compressor Pression setpoint value: P^0_{mc}=12 (Bar)

Fig. 14 Turbogas setpoint values

$$1300 \leq P_{el}(t) \leq 2500$$
$$200 \leq T_{exh}(t) \leq 580$$
$$40 \leq V_{rot}(t) \leq 120$$

Fig. 15 Turbogas Control System requirements

output variables P_{el}, V_{rot}, T_{exh} as close as possible to their *setpoint* (as shown in Fig. 14) and always within the limits given in Fig. 15, notwithstanding variations in the user demand u. Such limits are our requirements, i.e. the properties that we will have to check during verification.

In [15] it is shown that by adding finite precision real numbers to Murφ, we can use it to automatically verify TCS. In particular, in [15] it has been shown that, if the speed of variation of the user demand for electric power (`MAX_D_U` in the following) is greater than or equal to 25 (KW/sec), TCS fails in maintaining the ICARO parameters within the required safety ranges. A TCS state in which one of the ICARO parameters is outside its given safety range is of course considered an *error state*.

However, in [15] the user demand has been modeled rather roughly, using nondeterministic automata. Here we show that using FHP-Murφ we can define and, more importantly, automatically analyze a more accurate model for the user demand by modeling it using a Markov Chain. To do this we define a function $p(u, i)$ as follows:

$$p(u,i) = \begin{cases} 0.4 + \beta\dfrac{(u-u_0)|u-u_0|}{M^2} & \text{if } i = -1 \\ 0.2 & \text{if } i = 0 \\ 0.4 + \beta\dfrac{(u_0-u)|u-u_0|}{M^2} & \text{if } i = +1 \end{cases} \qquad (15)$$

where M = `MAX_U` (maximum user demand value), β is a suitable *elasticity constant* and $u_0 = \frac{M}{2}$ is the *user demand setpoint*.

Table 3 Results for the Turbogas control system

MAX_D_U	Reachable States	Rules Fired	Finite Horizon	CPU Time	Probability
25	3018970	8971839	1600	68562.570	7.373291768e-05
35	2226036	6602763	1400	50263.020	1.076644427e-04
45	1834684	5439327	1300	41403.150	9.957147381e-05
50	83189	246285	900	2212.360	3.984375e-03

FHP-Murφ configuration: `-m500` (use 500 Mb of RAM). Time is given in seconds

Denoting with $u(t)$ the user demand value at time t we can define the (stochastic) dynamics for the user demand as follows:

$$u(t+1) = \begin{cases} \max(u(t)-\alpha, 0) & \text{with prob. } p(u(t), -1) \\ u(t) & \text{with prob. } p(u(t), 0) \\ \min(u(t)+\alpha, M) & \text{with prob. } p(u(t), +1) \end{cases} \tag{16}$$

where α = MAX_D_U.

In this way we have that, the further $u(t)$ from u_0, the higher the probability to return towards u_0, i.e. to decrement $u(t)$ if $u(t) > u_0$ and to increment it otherwise.

To see that (16) is indeed a Markov Chain, it is sufficient to observe that, $\forall\beta$, the sum of the outgoing transitions is obviously 1. Moreover, since $\frac{(u(t)-u_0)|u(t)-u_0|}{M^2} \leq 1$, as long as $-0.4 \leq \beta \leq 0.4$ holds, all probability values are between 0 and 1.

With FHP-Murφ the definition of the Markov Chain (16), starting from the TCS model, is quite simple. This is done in Fig. 16, where `user_demand(u, d_u)` computes $p(u, d\_u)$ (15) and function `main` updates the system state, in particular updates `u` as described in (16).

In Fig. 3 we report the results ofsome verification runs done by FHP-Murφ with $\beta = 0.4$. We are interested in cases where the error probability is greater than zero, and from the results in [15] we know that this is the case if we choose MAX_D_U greater than or equal to 25 and the horizon value no smaller than $|\text{PathErr}(n)|$, where $\text{PathErr}(n)$ is the shortest path to a TCS error state when MAX_D_U $= n$. Thus, in our experiments here we choose our horizon k to be equal to $100 \cdot \lceil \frac{|\text{PathErr}(n)|}{100} \rceil$. In this way we check the error probability in the error neighborhood.

Table 3 allows us to evaluate the probability of reaching an error state when MAX_D_U is greater than or equal to 25. Note that such a probability is rather small, suggesting that in many cases setting MAX_D_U to 25 may be acceptable. Note that this kind of evaluations were not possible with the nondeterministic verification of TCS carried out in [15].

```
/* user demand disturbance: takes values
  -1, 0 and 1 */
ruleset d_u : -1..1 do
 rule "time step" user_demand(u, d_u) ==>
  main(u, d_u);
end;
```

Fig. 16 Probabilistic *user demand* ruleset in FHP-Murφ

7 Conclusions

In this paper we presented (Sects. 3 and 4) an *explicit* disk-based verification algorithm for Probabilistic Systems defining *discrete time/finite state* Markov Chains. Given a Markov Chain and an integer k (horizon), our algorithm checks that the probability of reaching a given error state in at most k steps is below a given probability threshold.

We described (Sect. 5) an implementation of our algorithm within a suitable extension of the Murφ verifier that we call FHP-Murφ (*Finite Horizon Probabilistic*-Murφ).

Finally, we showed (Sect. 6) experimental results comparing FHP-Murφ with (a finite horizon subset of) PRISM, a state-of-the-art symbolic model checker for Markov Chains. Our experimental results show that FHP-Murφ can handle systems that are out of reach for PRISM, namely those involving arithmetic operations on the state variables (e.g. hybrid systems).

Future work includes extending our approach to other models (e.g. Continuous Time Markov Chains) as well as to other kinds of PCTL formulas, e.g. formulas with unbounded until.

References

1. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: ICCAD '93: Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, pp. 188–191. IEEE Computer Society Press, Los Alamitos, CA, USA (1993)
2. Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M., Ryan, M.: Symbolic model checking for probabilistic processes. In: Degano, P., Gorrieri, P., Marchetti-Spaccamela, A. (eds.) Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, Proceedings, vol. 1256 of Lecture Notes in Computer Science, pp. 430–440. Springer, Berlin (1997)
3. Behrends, E.: Introduction to Markov Chains, Vieweg. Germany (2000)
4. Bianco, de Alfaro: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) Foundations of Software Technology and Theoretical Computer Science, 15th Conference, Bangalore, India, Proceedings, vol. 1026 of Lecture Notes in Computer Science, pp. 499–513. Springer, Berlin (1995)

5. Bobbio, A., Ciancamerla, E., Franceschinis, G., Gaeta, R., Minichino, M., Portinale, L.: Methods of increasing modelling power for safety analysis, applied to a turbine digital control system. In: Anderson, S., Bologna, S., Felici, M. (eds.) Computer Safety, Reliability and Security 21st International Conference, SAFECOMP 2002, Catania, Italy, Proceedings, vol. 2434 of Lecture Notes in Computer Science, pp. 212–223. Springer, Berlin (2002)
6. Bobbio, A., Ciancamerla, E., Gribaudo, M., Horvath, A., Minichino, M., Tronci, E.: Model Checking based on fluid petri nets for the temperature control system of the icaro co-generative Planti. In: Anderson, S., Bologna, S., Felici, M. (eds.) Computer Safety, Reliability and Security, 21st International Conference, SAFECOMP 2002, Catania, Italy, Proceedings, vol. 2434 of Lecture Notes in Computer Science, pp. 273–283. Springer, Berlin (2002)
7. Bobbio, A., Bologna, S., Minichino, M., Ciancamerla, E., Incalcaterra, P., Kropp, C., Tronci, E.: Advanced techniques for safety analysis applied to the gas turbine control system of Icaro co generative plant. In: Proceedings of X Convegno TESEC, Genova, Italy (2001)
8. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
9. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
10. Clarke, E.M., McMillan, K.L., Zhao, X., Fujita, M., Yang, J.: Spectral transforms for large Boolean functions with applications to technology mapping. In: Proceedings of the 30th International on Design automation conference, pp. 54–60. ACM Press, New York (1993)
11. Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: Proceedings of the IEEE Conference on Decision and Control, pp. 338–345. IEEE Press, Piscataway, NJ (1988)
12. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM. **42**(4), 857–907 (1995)
13. CUDD Web Page: http://vlsi.colorado.edu/∼fabio/ (2004)
14. de Alfaro, L.: Formal verification of performance and reliability of real-time systems. Technical Report STAN-CS-TR-96-1571, Stanford University (1996)
15. Della Penna, G., Intrigila, B., Melatti, I., Minichino, M., Ciancamerla, E., Parisse, A., Tronci, E., Venturini Zilli, M.: Automatic verification of a turbogas control system with the murφ verifier. In: Maler, O., Pnueli, A. (eds.) Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003 Prague, Czech Republic, Proceedings, vol. 2623 of Lecture Notes in Computer Science, pp. 141–155. Springer, Berlin (2003)
16. Della Penna, G., Intrigila, B., Melatti, I., Tronci, E., Venturini Zilli, M.: Finite horizon analysis of markov chains with the Murφ verifier. In: Geist, D., Tronci, E. (eds.) Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, Proceedings, vol. 2860 of Lecture Notes in Computer Science, pp. 394–409. Springer (2003)
17. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer and Processors, pp. 522–525. IEEE Computer Society, Washington, DC (1992)
18. ENEA: Proprietary ICARO Documentation (2001)
19. Hansson, H.: Time and Probability in Formal Design of Distributed Systems. Elsevier, Amsterdam (1994)
20. Hansson, H., Jonsson, B.: A logic for reasoning about time and probability. Formal Aspects Comput **6**(5), 512–535 (1994)
21. Hart, S., Sharir, M.: Probabilistic temporal logics for finite and bounded models. In: Proceedings of the sixteenth annual ACM symposium on Theory of computing, pp. 1–13. ACM Press, New York (1984)
22. Holzmann,G.J.: Design and Validation of Computer Protocols. Prentice-Hall, Upper Saddle River, NJ (1991)
23. Holzmann, G.J.: The spin model checker. IEEE Trans. Software Eng. **23**(5), 279–295, (1997)
24. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J.T., Harder, U. (eds.) Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, London, UK, Proceedings, vol. 2324 of Lecture Notes in Computer Science, pp. 200–204. Springer, Berlin (2002)
25. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. In: Katoen, J.-P., Stevens, P. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002, Proceedings, vol. 2280 of Lecture Notes in Computer Science, pp. 52–66. Springer, Berlin (2002)
26. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. **94**(1), 1–28 (1991)
27. Lehmann, D., Rabin, M.: On the advantages of free choice: A symmetric fully distributed solution to the dining philosophers problem (extended abstract). In: Proceedings of 8th Symposium on Principles of Programming Languages, pp. 133–138 (1981)
28. Lynch, N., Saias, I., Segala, R.: Proving time bounds for randomized distributed algorithms. In: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing, pp. 314–323. ACM Press, New York (1994)
29. Murphi Web Page: http://sprout.stanford.edu/dill/murphi.html (2004)
30. Pnueli, A., Zuck, L.: Verification of multiprocess probabilistic protocols. Distrib. Comput. **1**(1), 53–72 (1986)
31. Pnueli, A., Zuck, L.D.: Probabilistic verification. Inf. Comput. **103**(1), 1–29 (1993)
32. PRISM Web Page: http://www.cs.bham.ac.uk/∼dxp/prism/ (2004)
33. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. In: Jonsson, B., Parrow, J. (eds.) CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, Proceedings, vol. 836 of Lecture Notes in Computer Science, pp. 481–496. Springer, Berlin (1994)
34. SPIN Web Page: http://spinroot.com (2004)
35. Tronci, E., Della Penna, G., Intrigila, B., Venturini Zilli, M.: Exploiting transition locality in automatic verification. In: Margaria, T., Melham, T.F. (eds.) Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, Proceedings, vol. 2144 of Lecture Notes in Computer Science, pp. 259–274. Springer, Berlin (2001)
36. Vardi, M.: Automatic verification of probabilistic concurrent finite-state programs. In: 26th Annual Symposium on Foundations of Computer Science, pp. 327–338, IEEE CS Press, Portland, OR (1985)