Bounded Probabilistic Model Checking 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. In this paper we present an explicit verification algorithm for Probabilistic Systems defining *discrete time/finite state* Markov Chains. We restrict ourselves to verification of *Bounded* PCTL formulas (BPCTL), that is, PCTL formulas in which all *Until* operators are bounded, possibly with different bounds. This means that we consider only paths (system runs) of bounded length. Given a Markov Chain $\mathcal{M}$ and a BPCTL formula Φ, our algorithm checks if Φ is satisfied in $\mathcal{M}$. This allows to verify important properties, such as reliability in *Discrete Time Hybrid Systems*.
We present an implementation of our algorithm within a suitable extension of the Murφ verifier. We call FHP-Murφ (*Finite Horizon Probabilistic* Murφ) such extension of the Murφ verifier.
We give 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 effectively handle verification of BPCTL formulas for systems that are out of reach for PRISM, namely those involving arithmetic operations on the state variables (e.g. hybrid systems).

1 Introduction

Model checking techniques [5, 12, 19, 18, 25, 32] are widely used to verify 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. However, they can often be modeled by *Markov Chains* [2, 15]. Roughly speaking,

* Corresponding Author: Igor Melatti. Tel: +39 0862 43 3189 Fax: +39 0862 43 3057

a Markov Chain can be seen as an automaton labeled 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. [33, 8, 14]). This motivates the development of *Probabilistic Model Checkers* [9, 1, 20], i.e. of 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 seconds is less than 0.80". Note that, following [20, 21], we are using the expression "probabilistic model checking" to mean model checking of probabilistic systems.

Many methods have been proposed for probabilistic model checking, e.g. [11, 3, 8, 14–16, 22, 31, 33].

To the best of our knowledge, currently, the state-of-the-art probabilistic model checker is PRISM [30, 1, 21]. PRISM overcomes the limitations due to the use of linear algebra packages in Markov Chain analysis by using *Multi Terminal Binary Decision Diagrams* (MTBDDs) [7], a generalization of *Ordered Binary Decision Diagrams* (OBDDs) [4] allowing real numbers in the interval $[0, 1]$ on terminal nodes. Roughly speaking, PRISM can use three approaches to Markov Chain analysis. Namely: a sparse matrix based approach (based on linear algebra packages), a symbolic approach (based on the CUDD package [10]) and a hybrid approach, which uses MTBDDs to represent the system transition matrix and sparse matrix algorithms to carry out the (quantitative) probabilistic analysis [21]. As shown in [21], PRISM hybrid approach is faster than probabilistic model checkers based only on MTBDDs (e.g., ProbVerus [34]) and avoids the state explosion problem of probabilistic model checkers based only on sparse matrices (e.g., ETMCC [17] or the algorithms in [14, 15]).

Here we are mainly interested in 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 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 usage of a symbolic probabilistic model checker (e.g. like PRISM) on such systems rather problematic.

To this end in [27] is presented an explicit disk based algorithm for automatic *Finite Horizon* safety analysis of Markov Chains. The algorithm in [27] has been implemented in the probabilistic model checker FHP-Murφ (Finite Horizon Probabilistic Murφ) [6].

The experimental results in [27] show that FHP-Murφ outperforms PRISM on (discrete time) hybrid systems verification. Note however that PRISM can handle all PCTL [14] logic, whereas FHP-Murφ only handles *finite horizon safety properties* (e.g. like "the probability of reaching an error state in k steps is less than a given threshold"). Moreover, in [28] it is shown that FHP-Murφ input language is more natural than the PRISM one in order to specify many Stochastic Systems.

Unfortunately there are many interesting (finite horizon) properties that cannot be expressed as safety properties. For example reliability and robustness properties like: *"the probability of reaching within k_1 steps an undesired state, which will not be left with high probability within k_2 steps, is low"* cannot be verified using the algorithm given in [27]. By an *undesired state* we mean a state in which the system should *not* be, e.g. a state in which the system cannot stay for a too long time, otherwise a damage occurs. Of course such properties can also be handled by PRISM, however then we hit the state explosion barrier quite soon when handling hybrid systems (our goal here).

The above considerations suggest extending FHP-Murφ capabilities so as to handle all *Bounded PCTL* (BPCTL) properties. That is, PCTL properties in which all *Until* operators are bounded, possibly with different bounds. In other words, we consider only paths (system runs) of bounded length. Clearly BPCTL allows us to define reliability properties and, indeed, much more. Our results can be summarized as follows.

– We present (Section 3) a new explicit verification algorithm for *finite state/discrete time* Markov Chains. Our present algorithm can handle *all* BPCTL formulas, whereas the one presented in [27] can only handle safety properties. Moreover, our present algorithm *is not a simple extension of the one in* [27], since, to handle the reliability properties we are interested in, which result in BPCTL properties with nested *Untils*, we had to completely re-engineer it. Namely, the BF (*Breadth First*) visit of the state transition graph in [27] has been changed into a DF (*Depth First*) visit (see Section 3), with an *ad-hoc* caching strategy that allows to better handle BPCTL properties with nested *Untils*. Finally, our algorithm is *disk based*, therefore, because of the large size of modern hard disks, memory is hardly a problem for us. Computation time instead is our bottleneck. However, our algorithm can trade RAM with computation time, i.e. the more RAM available the faster our computation (see Section 3.1). To the best of our knowledge, this is the first time that such an algorithm for probabilistic model checking is proposed.
– We present (Section 3.2) an implementation of our algorithm within the FHP-Murφ verifier.
– We present (Section 4.1) experimental results comparing our *BPCTL enhanced* FHP-Murφ with PRISM on the two probabilistic dining philosophers protocols included in the PRISM distribution, and also on the two modified version of the same protocols presented in [27]. Our experimental results show that BPCTL enhanced 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 our FHP-Murφ (as to be expected).
– We present (Section 4.2) experimental results on using BPCTL enhanced 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 [26]. Because of the arithmetic operations involved in the definition of system dynamics, this hybrid system is out of reach for OBDDs (and thus

for PRISM), whereas FHP-Murφ can complete verification of interesting reliability properties within a reasonable amount of time.

2 Basic Notation

2.1 Markov Chains

Let S be a finite set. 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 $\mathbf{x}(s)$. 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.

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. [2].

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 *state* $j \in S$ iff $\mathbf{x}(j) = 1$ (thus $\mathbf{x}(i) = 0$ when $i \neq j$).

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. *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 our notation.)*

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. 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 $|\pi|$ the length of π. We denote with $Path(\mathcal{M}, s)$ the set of infinite paths π in $\mathcal{M}$ s.t. $\pi(0) = s$, whereas $Path_k(\mathcal{M}, s)$ is set of paths π in $\mathcal{M}$ s.t. $\pi(0) = s$ and $|\pi| = k$. If $\mathcal{M} = (S, \mathbf{P}, q)$ we write also $Path(\mathcal{M})$ for $Path(\mathcal{M}, q)$.*
   Moreover, we say that a state $s \in S$ is reachable *in k steps when it exists a path $\pi \in Path_k(\mathcal{M})$ such that $\pi(k) = s$.*
3. *For $s \in S$ we denote with $\sum(s)$ the smallest σ-algebra on $Path(\mathcal{M}, s)$ which, for any finite path ρ starting at s, contains the basic cylinders $\{ \pi \in Path(\mathcal{M}, s) \mid \rho$ is a prefix of $\pi \}$. The probability measure $Prob$ on $\sum(s)$ is the unique*

measure with $Prob(\{\pi \in Path(\mathcal{M}, s) | \rho \text{ is a prefix of } \pi\}) = \mathbf{P}(\rho) = \prod_{i=0}^{k-1} \mathbf{P}(\rho(i), \rho(i{+}1)) = \mathbf{P}(\rho(0), \rho(1))\mathbf{P}(\rho(1), \rho(2)) \cdots \mathbf{P}(\rho(k{-}1), \rho(k))$, *where* $k = |\rho|$.

E.g. given a 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 $\mathbf{y} = \mathbf{xP}$ and $\mathbf{x}(s) = 1$ we have that $\forall t[\mathbf{y}(t) = (\mathbf{P})_{s,t}]$.

The Markov Chain definition 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.

Thus 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 2. *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 3. *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* $\mathbf{P}(s, t) = \sum_{(t,a,p) \in \texttt{next}(s)} p$.

Moreover, a state sequence $\pi = s_0 s_1 s_2 \ldots$ *is a path in* $\mathcal{S}$ *iff it is a path in* $\mathcal{S}^{mc}$.

2.2 BPCTL

In this Section we give syntax (Definition 4) and semantics (Definition 5) for BPCTL (*Bounded PCTL*). BPCTL formulas only consider PCTL formulas in which all *Until* operators are bounded, possibly with different bounds. This means that we consider only paths (system runs) of bounded length.

Definition 4. *Let* AP *be a finite set of atomic propositions, i.e. of functions* $p : S \to \{0, 1\}$. *The BPCTL language* $\mathcal{L}_{BPCTL}$ *is the language generated by the following grammar:*

$$\Phi ::= tt \mid p \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid [\mathbf{X}\Phi]_{\sqsupseteq\alpha} \mid [\Phi_1 \, \mathbf{U}^{\leq k} \, \Phi_2]_{\sqsupseteq\alpha}$$

where $\alpha \in [0, 1]$ *and* $k \in \mathbb{N}$ *and the symbol* $\sqsupseteq$ *is one of the symbols* $>, \geq$.

Definition 5. *Let* $\mathcal{M} = (S, \mathbf{P}, q)$ *be a Markov Chain. Then, the satisfaction relation* $\models \subseteq S \times \mathcal{L}_{BPCTL}$ *is defined, for all* $s \in S$, *as follows:*

– $s \models tt$;

– $s \models p$ *iff* $p(s) = 1$;
– $s \models \Phi_1 \wedge \Phi_2$ *iff* $s \models \Phi_1$ *and* $s \models \Phi_2$;
– $s \models \neg\Phi$ *iff* $s \not\models \Phi$;
– $s \models [\mathbf{X}\Phi]_{\sqsupseteq\alpha}$ *iff* $Prob\{\pi \in Path(\mathcal{M}, s) \mid \pi(1) \models \Phi\} \sqsupseteq \alpha$;
– $s \models [\Phi_1\ \mathbf{U}^{\leq k}\ \Phi_2]_{\sqsupseteq\alpha}$ *iff* $Prob\{\pi \in Path(\mathcal{M}, s) \mid \exists h \leq k$ *s. t.* $[\pi(h) \models \Phi_2$ *and* $\forall i < h\ \pi(i) \models \Phi_1]\} \sqsupseteq \alpha$.

Moreover, let F *be a BPCTL formula. Then,* $\mathcal{M} \models F$ *iff* $q \models F$.
Finally, let $\mathcal{S}$ *be a PFSS. Then,* $\mathcal{S} \models F$ *iff* $\mathcal{S}^{mc} \models F$.

Finally, we give two definitions that will be useful in the following.

Definition 6. *Let* $\mathcal{M} = (S, \mathbf{P}, q)$ *be a Markov Chain,* Φ, Ψ *be BPCTL formulas,* $k \in \mathbb{N}$ *and* $s \in S$. *Then we write* $P_s[\Phi\ \mathbf{U}^{\leq k}\ \Psi]$ *for* $Prob\{\pi \in Path(\mathcal{M}, s) \mid \pi \models \Phi\ \mathbf{U}^{\leq k}\ \Psi\}$.

Definition 7. *A BPCTL formula* Φ *is said to be a* $\mathbf{U}$*-formula iff there are BPCTL formulas* Φ_1, Φ_2 *s.t.* $\Phi \equiv [\Phi_1\ \mathbf{U}^{\leq k}\ \Phi_2]_{\sqsupseteq\alpha}$.

Remark 1. From Definition 5, we can intuitively see that the truth value for $s \models \Phi$ can be evaluated by taking into account only paths of finite length k, provided that k is large enough and computing, when needed, $Prob\{\pi \in \mathrm{Path}_k(\mathcal{M}, s) \mid \mathcal{P}(\pi)\}$ (for some path property $\mathcal{P}$) as $\sum_{\pi\ s.t.\ \mathcal{P}(\pi)} \mathbf{P}(\pi)$.

If we denote with $s \models_k \Phi$ the semantics that considers only paths of length k, then we have the following theorem.

Theorem 1. *Let* $\mathcal{M} = (S, \mathbf{P}, q)$ *be a Markov Chain and* Φ *be a BPCTL formula. Then, there exists a* k *such that, for all* $s \in S$ *and for all* $h \geq k$, $s \models \Phi$ *iff* $s \models_h \Phi$.

The reader is referred, for the mathematical details, to the online appendices of the present paper [13] (Appendix A).

3 Explicit BPCTL Model Checking

In this Section we present an explicit algorithm to verify if a PFSS $\mathcal{S} = (S, q, \mathcal{A}$, `next`$)$ satisfies a given BPCTL formula F ($\mathcal{S} \models F$).

By Definition 5, it is clear that the most difficult case in the verification of F is to compute the truth value of a **U**-formula. In [27], we solved this problem by implementing a BF (*Breadth First*) visit of the $\mathcal{S}$ state space. However, a BF visit it is not effective when dealing with nested **U**-formulas, which are exactly the kind of formulas defining the robustness and reliability properties we are interested in.

In fact, suppose that $F \equiv [tt\ \mathbf{U}^{\leq k_1}\ \Phi_1]_{\sqsupseteq\alpha}$, with $\Phi_1 \equiv [tt\ \mathbf{U}^{\leq k_2}\ \phi]_{\sqsupseteq\beta}$, being ϕ and atomic proposition. To determine if a state s is such that $s \models F$, we have to check, for all states t that are reachable from s in at most k_1 steps, if $t \models \Phi_1$; that is, we have to start, for all t, a nested BF visit. However, it is possible to avoid some of these nested visits. Now we will show how, using a DF visit and a cache, we reach this goal.

Our idea is that, using a DF visit, it is possible to compute, for all states r reached during the computation of $P_t[tt\ \mathbf{U}^{\leq k_2}\ \phi]$ (needed to check if $t \models \Phi_1$), the value $P_r[tt\ \mathbf{U}^{\leq k_2 - h}\ \phi]$, being h the number of transitions steps leading from t to r; this comes from the recursiveness of the DF visit itself. So, if we save in a cache slot r, Φ_1, h and $P_r[tt\ \mathbf{U}^{\leq k_2 - h}\ \phi]$, then this latter value may be used to avoid a nested DF visit when r is possibly reached again.

Note that the BF visit proposed in [27] is only able to say if the state s from which we start the visit is such that $s \models F$ (or $s \models \Phi_1$), so it is hard to apply such a caching strategy to the algorithm proposed in [27].

The rest of this Section is organized as follows. In Sect. 3.1 we give a formal description of our algorithm, explaining it also by means of a simple running example; in Section 3.2 we explain how we implemented the algorithm in the Murφ verifier.

3.1 Explicit Verification of BPCTL Formulas

In this Section we give a formal description of the algorithm, verifying a generic BPCTL formula. Let $\mathcal{S} = (S, q, \mathcal{A}, \mathtt{next})$ be a PFSS and F be a BPCTL formula. We want to check if $\mathcal{S} \models F$ (see Definition 5) holds.

```
/* the main function */
bool BPCTL(PFSS S, formula F) {
 return BPCTL_rec(S, q, F); } /* BPCTL() */

bool BPCTL_rec(PFSS S, state s, formula F) {
 if (F ≡ 1) return true;
 else if (F is an atomic proposition p) return p(s) = 1;
 else if (F ≡ ¬F1) return !BPCTL_rec(S, s, F1);
 else if (F ≡ F1 ∧ F2)
  return BPCTL_rec(S, s, F1) && BPCTL_rec(S, s, F2);
 else if (F ≡ [X Φ]⊒α) return evalX(S, s, F);
 else if (F ≡ [Φ U Ψ]⊒α) return evalU(S, s, F); } /* BPCTL_rec() */


bool evalX(PFSS S, state s, formula F) {
 Let F ≡ [X Φ]⊒α;
 sum = 0; /* accumulates the probability to see Φ in 1 step from s */
 for each (s_next, a, p_next) in next(s) {
  if (BPCTL_rec(S, s_next, Φ)) sum = sum + p_next;
 } /* for */
 return (sum ⊒ α); } /* evalX() */
```

Fig. 1. Functions `BPCTL`, `BPCTL_rec` and `evalX`

The main function `BPCTL`, taking $\mathcal{S}$ and F and returning `true` iff $\mathcal{S} \models \Phi$, is in Figure 1. This function uses, as auxiliary functions, the following ones:

- `BPCTL_rec` (also in Figure 1), is a recursive function that calls itself or the other auxiliary functions, as needed by the syntactical structure of the given BPCTL formula;
- `evalX` (also in Figure 1), is dedicated to the evaluation of formulas of form $[\mathbf{X}\ \Phi]_{\sqsupseteq\alpha}$;
- `evalU` (Figure 2), is dedicated to the evaluation of formulas of form $[\Phi\ \mathbf{U}^{\leq k}\ \Psi]_{\sqsupseteq\alpha}$;
- `DF_Search` (also in Figure 2), is a recursive auxiliary function for `evalU`, computing a finite horizon DF visit of the PFSS $\mathcal{S}$.

```
cache C;

bool evalU(PFSS 𝒮, state s, formula F){
 Let F ≡ [Φ U^≤k Ψ]_⊒α;
 {valid, result} = try_to_evaluate(C, s, F);
 if (valid)
  /* this means that function try_to_evaluate has been able to
    evaluate if s ⊨ F by using only the cache */
  return result;
 else {
  prob = DF_Search(𝒮, s, F, 0);
  return prob ⊒ α; } /* else */ } /* evalU */

double DF_Search(PFSS 𝒮, state s, formula F, int horizon) {
 if (BPCTL_rec(𝒮, s, Ψ)) prob = 1.0;
 else if (!BPCTL_rec(𝒮, s, Φ)) prob = 0.0;
 else { prob = 0.0;
  if (horizon < k) {
   for all (s_next, a, p_next) in next(s){
    prob_tmp = present_cache(C, s, F, k - horizon));
    if (prob_tmp == -1) /* value not found */
     prob = prob + p_next*DF_Search(𝒮, s_next, F, horizon + 1);
    else
     prob = prob + p_next*prob_tmp;
 } /* for */ } /* if */ } /* else */
 /* s exploration ended, the computed value can be inserted in C */
 insert_cache(C, s, F, k, prob);
 return prob; } /* DF_Search */
```

Fig. 2. Functions `evalU` and `DF_Search`

Correctness of the algorithm. The reader is referred, for the proof of the algorithm correctness, to the online appendices of the present paper [13] (Appendix B).

We illustrate how our algorithm works by means of the simple PFSS $\mathcal{S}$ shown in Figure 3. Given the BPCTL formula $F \equiv [tt\ \mathbf{U}^{\leq 2}\ \phi]_{\geq 0.5}$, where ϕ is an atomic proposition such that $\phi(s_1) = \phi(s_4) = \phi(s_7) = 1$ and is 0 on the other states (as shown in Figure 3), we want to verify if $s_0 \models F$.

Then, from `BPCTL`$(\mathcal{S}, F)$, going through `BPCTL_rec` and `evalU`, `DF_Search`$(\mathcal{S}, s_0, F, 0)$ is called, and the DF visit of $\mathcal{S}$ begins. Supposing that $\texttt{prob}_{s_i}$ is the value for the variable `prob` when `DF_Search` is called on the state s_i, we have that $\texttt{prob}_{s_0} = 0$, and a recursive call to s_1 is made. Here, $\texttt{prob}_{s_1} = 1$ (since $\phi(s_1) = 1$), and no recursive call is made; on the return to the previous call (on s_0), we have $\texttt{prob}_{s_0} = \frac{1}{3} \times 1$. Then, a recursive call on s_2 is made, from which other two recursive calls are made, first on s_3 and then on s_4. None of these two calls makes other recursive calls: s_3 because is called with $\texttt{horizon} = 2$, s_4 because $\phi(s_4) = 1$. This latter call will set $\texttt{prob}_{s_4} = 1$ and then $\texttt{prob}_{s_2} = \frac{1}{2}$. Now, we return in the visit to s_0, and now $\texttt{prob}_{s_0} = \frac{1}{3} + \frac{1}{3}\texttt{prob}_{s_2} = \frac{1}{3} + \frac{1}{3}\frac{1}{2} = 0.5$. Then, other two nested calls on s_5 and s_6 are made but, since none of the two satisfies ϕ, $\texttt{prob}_{s_0}$ does not change. So, the final result is that $s_0 \models F$, since $0.5 \geq 0.5$. Note that s_7 and s_8 are never reached: the former because is only reachable from s_1, that is not expanded since $\phi(s_1) = 1$, the latter because is beyond the F horizon, i.e. it is not reached in at most 2 steps from s_0.

For what concerns the cache, it is organized as follows. Each cache slot contains a state s (`C[h].state` in Figure 4) together with a $\mathbf{U}$-formula $F \equiv$

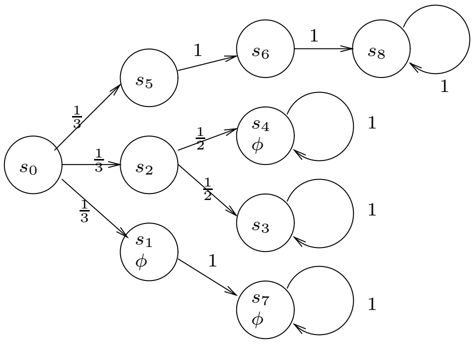

Fig. 3. A PFSS simple example

$[\Phi\ \mathbf{U}^{\leq h}\ \Psi]_{\sqsupseteq\alpha}$ (`C[h].form`), an integer h (`C[h].horizon`) and the respective $P_s[\Phi\ \mathbf{U}^{\leq h}\ \Psi]$ (`C[h].prob`). In this way, we exploit the recursiveness of the DF visit, which allows us to compute, for every state t reached during the computation of $s \models F$, the number $P_t[\Phi\ \mathbf{U}^{\leq h}\ \Psi]$, where h is the number of steps that t needs to reach the horizon (i.e. $h = k - j$, being j the number of steps from s to t). In this way, we avoid to perform already done computations, so saving time with a fixed amount of memory.

```
{bool, bool} try_to_evaluate(cache C, state s, formula F) {
 Let F ≡ [Φ U^{≤k} Ψ]_{⊒α};
 /* The 4 fields of each cache slot h have the following meaning: let
  C[h].form=[Φ1 U^{≤i} Φ2]_{⊒β}; then, C[h].prob = P_{C[h].state}[Φ1 U^{≤C[h].horizon} Φ2]
 */
 if (<s, F> is not in C) return {false, _};
 else {
  for all j such that (C[j].state == s && C[j].form == F) {
   if (C[j].horizon == k) return {true, C[j].prob ⊒ α};
   else if (C[j].horizon > k && !(C[j].prob ⊒ α))
    return {true, false};
   else if (C[j].horizon < k && C[j].prob ⊒ α)
    return {true, true};
  } /* for */
  return {false, _}; } /* else */ } /* try_to_evaluate */
```

Fig. 4. Function `try_to_evaluate`

This saving may take place in two ways:

- when `evalU` calls `try_to_evaluate` (see Figure 4), to attempt to avoid a call to `DF_Search`. This function is based on the fact that $P_s[\Phi\ \mathbf{U}^{\leq h_1}\ \Psi] \leq P_s[\Phi\ \mathbf{U}^{\leq h_2}\ \Psi]$ for all $s \in S$, BPCTL formulas Φ, Ψ and $h_1 \leq h_2$. This implies that, even if the searched pair (state, formula) is not present in the cache with the required horizon, we can say if $s \models F$: as an example, this happens if the horizon stored in the cache is less than the required one, but the stored probability is already greater than α;
- when `DF_Search` calls `present_cache`, to attempt to avoid a recursive call. In this case, the recursive call can be avoided only if we find in the cache the exact entry for the state, the formula and the horizon, since here we need the exact probability value. To exemplify this, consider again Figure 3, and

suppose that there were two transitions from s_0 to s_2, both with probability $\frac{1}{6}$. Then, the second recursive call on s_2 is avoided, since the first call has put on the cache the value $P_{s_2}[tt\ \mathbf{U}^{\leq 1}\ \phi]$, which is returned by function `present_cache`.

Finally, function `insert_cache`, being slightly different from the usual implementation, is in Figure 5. In this function, when a free cache slot has not been found, a slot is overwritten only if it refers to the same pair (state, formula), and has minor horizon. In this way, we overwrite only information obtained with less computation resources. Moreover, the hash collision chain due to open addressing is dynamically extended when too many insertions fail. It is so clear that our algorithm trades memory with time: if we are given more memory, we will have a larger cache, which will be able to store more probabilities, thus avoiding more recursive calls to the DF visit.

```
int M = max length of the open addressing cache collision chain;

void insert_cache(cache C, state s, formula F, int hor, double prob) {
 h = hash(s);
 while (C[h] is not empty) {
  h = hash(s);
  if (more than M times in this while) break;
 } /* while */
 if (C[h] is empty) return C[h].prob;
 if (the previous while has been broken too many often w.r.t. the
  number of calls to insert_cache) M = M*2;
 if (C[h] contains s and F && C[h].horizon < hor)
  overwrite C[h] with {s, F, hor, prob} } /* insert_cache */
```

Fig. 5. Function `insert_cache`

3.2 Implementation within the Murφ Verifier

We implemented the algorithm given in Section 3 within the Murφ verifier. We started from FHP-Murφ [27], a probabilistic version of Murφ in which only a subset of BPCTL formulas can be verified.

Since FHP-Murφ already allows specification of PFSSs, the input language has been modified only to allow definition of BPCTL formulas.

On the other hand, the verification algorithm has been implemented along the lines shown in Figure 1, 2 and 4. The only adjustment is in function `evalU`, that cannot be implemented using standard C recursion. So, a stack has been implemented to explicitly handle the recursive calls. Since we are in a bounded framework, the stack size is limited, and is given by the following definition.

Definition 8. *Let stack_size*: $\mathcal{L}_{BPCTL} \to \mathbb{N}$ *be the function returning the stack size that is needed to verify a BPCTL formula* Φ. *Then stack_size is defined as follows:*

- $stack\_size(tt) = stack\_size(p) = 0$
- $stack\_size(\Phi_1 \wedge \Phi_2) = \max\{stack\_size(\Phi_1), stack\_size(\Phi_2)\}$
- $stack\_size([\mathbf{X}\Phi]_{\sqsupseteq\alpha}) = stack\_size(\neg\Phi) = stack\_size(\Phi)$;
- $stack\_size([\Phi_1\ \mathbf{U}^{\leq k}\Phi_2]_{\sqsupseteq\alpha}) = k + \max\{stack\_size(\Phi_1), stack\_size(\Phi_2)\}$

Hence, the amount of memory needed by the verification task is fixed, so we have that our real bottleneck is time, and not memory. However, to handle the case in which we need more memory than the available one, we implemented the swap-to-disk mechanism *stack cycling*, which is also implemented in the DF-based verifier SPIN [19]. With this technique, only a part of the stack is maintained in RAM: when there is a push or a pop operation outside of the stack part in RAM, then a disk block (containing a certain number of states) is used to store or retrieve the desired states. This mechanism avoids too frequent disk accesses due to repetition of push-pop operations.

In this way, we use the RAM to store part of the DF stack and of our cache. Our experiments show that typically we can take the RAM size for the DF stack as inversely proportional to the number of nested **U**-formulas, since, in this case, it is important to have a large cache in order to speed up the verification process.

4 Experimental Results

To show the effectiveness of our approach we run two kinds of experiments.

First, in Section 4.1, we compare verifications of BPCTL formulas done with FHP-Murφ with verifications of the same models done with the probabilistic model checker PRISM [30].

Second, in Section 4.2, we run FHP-Murφ to verify a robustness property on a quite large probabilistic hybrid systems. Since our main goal is to use FHP-Murφ to prove hybrid systems robustness properties, this second kind of evaluation is very interesting for us.

4.1 Probabilistic Dining Philosophers

In this Section we give our experimental results on using FHP-Murφ on the probabilistic protocols included in PRISM distribution [30]. 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-Zuck [29] and Lehmann-Rabin [23, 24] probabilistic dining philosophers protocols. For both of these protocols, we use two versions: the one which can be found in the PRISM distribution, and the modified version allowing *quality of service* properties verifications, as it is described in [27].

For what concerns the BPCTL properties to be verified, we proceed as follows. For the models in the PRISM distribution, we choose one of the relative BPCTL properties and we modify it so as to obtain an equivalent BPCTL property. In fact, the PRISM BPCTL properties about these protocols are of the type $\phi_P \equiv p \rightarrow [tt\ \mathbf{U}^{\leq k}\ q]_{\geq \alpha}$, where p and q are atomic propositions. However, these formulas are not evaluated on the initial state (which is the standard PCTL semantics), but on all reachable states. To obtain a comparable result with FHP-Murφ, we verify the property $\Phi_M \equiv [tt\ \mathbf{U}^{\leq d}\ (p \wedge \neg[tt\ \mathbf{U}^{\leq k} B]_{\geq \alpha})]_{\leq 0}$, where d is the diameter of the protocol state space, i.e. the length of maximum path between two states. In this way, we have that $q \models \Phi_M$ iff, for all reachable states s, $s \models \Phi_P$.

Our results are in Figure 6 and 7 (with $k = 20$). Note that, in these set of experiments, which do not involve mathematical operations, PRISM works better, while FHP-Murφ take too much time to complete the verifications.

NPHIL	Result	Murφ Mem (MB)	PRISM Mem (MB)	Murφ Time (s)	PRISM Time (s)
5	false	5.0e+2	1.701300e+00	3.41117000e+03	1.318000e+00
6	false	5.0e+2	1.430420e+01	>3.0000000e+05	1.260300e+01

Fig. 6. Results for the Pnueli-Zuck protocol as it is found in the PRISM distribution. We use a machine with 2 processors (both INTEL Pentium III 500Mhz) and 2GB of RAM. Murφ options: `-m500` (use exactly 500MB of RAM). PRISM options: default options.

NPHIL	Result	Murφ Mem (MB)	PRISM Mem (MB)	Murφ Time (s)	PRISM Time (s)
3	true	5.0e+2	1.419900e+00	1.79230000e+03	2.018000e+00
4	true	5.0e+2	2.355610e+01	1.42337890e+05	1.034140e+02

Fig. 7. Results for the Lehmann-Rabin protocol as it is found in the PRISM distribution.The fields have the same meaning of Fig. 6.

NPHIL	MAX_WAIT	Result	Murφ Mem (MB)	PRISM Mem (MB)	Murφ Time (s)	PRISM Time (s)
5	3	false	5.0e+2	9.168246e+02	1.28381900e+04	1.196793e+03
5	4	false	5.0e+2	N/A	1.27377300e+04	N/A

Fig. 8. Results for the Pnueli-Zuck protocol as it was modified in [27]. The fields have the same meaning of Fig. 6. N/A means that PRISM was unable to complete the verification; in this case, also the `-m` and `-s` (totally MTBDD and algebraic verification algorithm respectively) have been used, with the same result.

NPHIL	MAX_WAIT	Result	Murφ Mem (MB)	PRISM Mem (MB)	Murφ Time (s)	PRISM Time (s)
3	4	true	5.0e+2	7.014830e+01	5.00634000e+03	5.359870e+02
4	3	true	5.0e+2	N/A	1.11480680e+05	N/A

Fig. 9. Results for the Lehmann-Rabin protocol as it was modified in [27]. The fields have the same meaning of Fig. 6.

For the modified models, we verify a reliability property. In fact, in this models, there is a set of error states, i.e. those satisfying a special atomic proposition ϕ_{err} (informally, *"a philosopher does not eat for a too long time, and dies for starvation"*). To define our reliability property, we introduce a new atomic proposition ϕ_{und}, which is a weaker version of ϕ_{err} in the sense that, for all states s, if s satisfies ϕ_{err}, then it satisfies also ϕ_{und} (informally, *"a philosopher does not eat for a long time, and he is in danger"*). So, our undesired states are those satisfying ϕ_{und}. The reader is referred, for a detailed description of ϕ_{err} and ϕ_{und}, to the online appendices of the present paper [13] (Appendix C).

Now, we want to say that, when an undesired state s is reached, then the system almost always reaches, from s and in a few steps, a non-error state. Then, our property states that there is a low probability that if we reach in k_1 steps a state s such that $\phi_{und}(s)$ holds, and there is not a high probability of reaching, from s and in $k_2 = \frac{k_1}{10}$ steps, a state t such that $\neg\phi_{err}(t)$. The corresponding BPCTL formula is $[tt\ \mathbf{U}^{\leq k_1}\ (\phi_{und} \wedge \neg[tt\ \mathbf{U}^{\leq k_2} \neg\phi_{err}]_{\geq 1})]_{\leq 0}$. We give this BPCTL formula both to PRISM and FHP-Murφ.

Our results are in Figures 8 and 9 (with $k_1 = 20$). We can observe that, requiring these protocols some mathematical operations, there are cases (i.e., the last rows in Figures 8 and 9) in which no PRISM strategy (i.e., MTBDD based, sparse matrix based, hybrid approach) is able to complete the verification task, while FHP-Murφ does.

4.2 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 2MW *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 the most critical subsystem in ICARO. Unfortunately TCS is also the largest ICARO subsystem, thus making the use of model checking for such hybrid system a challenge.

In [26] it is shown that by adding finite precision real numbers to Murφ, we can use Murφ to automatically verify TCS. In particular in [26] it has been shown the following. If the 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 ICARO parameters within the required safety ranges. A TCS state in which one of ICARO parameters is outside its given safety range is of course considered an *error state*. On the other hand, a state is considered a *undesired state* when it is outside a larger safety range (the system will crash if it stays in such a state for a too long time).

In [27] FHP-Murφ has been used to verify finite horizon probabilistic safety properties of TCS.

Here we show that by using BPCTL enhanced FHP-Murφ we can verify robustness properties of TCS. Here is an example: *"if the system reaches an undesired state, it is able to return to a non-undesired state with a high probability"*. Our robustness property is so equivalent to say that there is a low probability of reaching an undesired state s, such that there is not an high probability of reaching a non-undesired state from s. The relative BPCTL formula is $[tt\ \mathbf{U}^{\leq k_1}\ (\neg\phi \wedge \neg[tt\ \mathbf{U}^{\leq k_2}\phi]_{\geq 1})]_{\leq 0}$, where ϕ defines the undesired states. The two constants k_1 and k_2 are chosen in such a way that k_1 is sufficient to reach an undesired state (if the first undesired state is reached in d steps, then $k_1 = \lceil \frac{d}{100} \rceil 100$), and k_2 is not too high (in our experiments, we took $k_2 = \frac{k_1}{100}$).

Our results are in Figure 10, where we show, in the field "Probability" the value $P_q[tt\ \mathbf{U}^{\leq k_1}\ (\neg\phi \wedge \neg[tt\ \mathbf{U}^{\leq k_2}\phi]_{\geq 1})]$, being q the system initial state.

`MAX_D_U`	Visited States	Rules Fired	k_1	CPU Time (s)	Probability
35	1.159160e+05	3.477480e+05	800	3.702400e+03	4.104681e-03
45	4.098000e+04	1.229400e+05	700	1.313900e+03	1.792883e-02
50	4.067700e+04	1.220310e+05	700	1.307850e+03	3.825000e-02

Fig. 10. Results on a machine with 2 processors (both INTEL Pentium III 500Mhz) and 2GB of RAM. Murφ options used: `-m500` (use 500 MB of RAM).

5 Conclusions

We presented (Section 3) an *explicit* verification algorithm for Probabilistic Systems defining *discrete time/finite state* Markov Chains. Given a Markov Chain $\mathcal{M}$ and Bounded PCTL formula Φ our algorithm checks if $\mathcal{M} \models \Phi$.

We presented (Section 3.2) an implementation of our algorithm within a suitable extension of the Murφ verifier that we call FHP-Murφ (*Finite Horizon Probabilistic*-Murφ).

We presented (Section 4) 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).

PRISM handles Continuous Time Markov Chains (CTMC) using a symbolic approach. This works well as long as the system dynamics does not involve heavy arithmetical computations. To enlarge the class of automatically verifiable probabilistic systems, future work includes extending our explicit approach to CTMCs. Another possible research direction is to extend Murφ so as to handle *unbounded until* PCTL formulas.

References

1. C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 430–440. Springer, 1997.
2. E. Behrends. *Introduction to Markov Chains.* Vieweg, 2000.
3. Bianco and de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science, 15th Conference, Bangalore, India, December 18-20, 1995, Proceedings*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer, 1995.
4. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug 1986.
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. *Inf. Comput.*, 98(2):142–170, 1992.
6. Cached murphi web page: http://www.dsi.uniroma1.it/~tronci/cached.murphi.html.
7. E. M. Clarke, K. L. McMillan, X Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Proceedings of the 30th international on Design automation conference*, pages 54–60. ACM Press, 1993.
8. C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Proceedings of the IEEE Conference on Decision and Control*, pages 338–345, Piscataway, NJ, 1988. IEEE Press.
9. Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4):857–907, 1995.
10. Cudd web page: http://vlsi.colorado.edu/~fabio/.
11. L. de Alfaro. Formal verification of performance and reliability of real-time systems. Technical Report STAN-CS-TR-96-1571, Stanford University, 1996.
12. David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525. IEEE Computer Society, 1992.
13. Online appendices: http://www.di.univaq.it/melatti/FMCAD04/.
14. B. Jonsson H. Hansson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
15. H. Hansson. *Time and Probability in Formal Design of Distributed Systems.* Elsevier, 1994.

16. Sergiu Hart and Micha Sharir. Probabilistic temporal logics for finite and bounded models. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 1–13. ACM Press, 1984.
17. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A tool for model-checking markov chains. *Software Tools for Technology Transfer*, 4(2):153–172, Feb 2003.
18. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, New Jersey, 1991.
19. G. J. Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
20. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In Tony Field, Peter G. Harrison, Jeremy T. Bradley, and Uli Harder, editors, *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, London, UK, April 14-17, 2002, Proceedings*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204. Springer, 2002.
21. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In Joost-Pieter Katoen and Perdita Stevens, editors, *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*, volume 2280 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2002.
22. Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Inf. Comput.*, 94(1):1–28, 1991.
23. D. Lehmann and M. Rabin. On the advantages of free choice: A symmetric fully distributed solution to the dining philosophers problem (extended abstract). In *Proc. 8th Symposium on Principles of Programming Languages*, pages 133–138, 1981.
24. N. Lynch, I. Saias, and R. Segala. Proving time bounds for randomized distributed algorithms. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 314–323. ACM Press, 1994.
25. Murphi web page: http://sprout.stanford.edu/dill/murphi.html.
26. 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 murφ verifier. In Oded Maler and Amir Pnueli, editors, *Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003 Prague, Czech Republic, April 3-5, 2003, Proceedings*, volume 2623 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2003.
27. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Finite horizon analysis of markov chains with the murφ verifier. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volume 2860 of *Lecture Notes in Computer Science*, pages 394–409. Springer, 2003.
28. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Finite horizon analysis of stochastic systems with the murφ verifier. In Carlo Blundo and Cosimo Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS 2003, Bertinoro, Italy, October 13-15, 2003, Proceedings*, volume 2841 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2003.

29. A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distrib. Comput.*, 1(1):53–72, 1986.
30. Prism web page: http://www.cs.bham.ac.uk/∼dxp/prism/.
31. R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings*, volume 836 of *Lecture Notes in Computer Science*, pages 481–496. Springer, 1994.
32. Spin web page: http://spinroot.com.
33. M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *26th Annual Symposium on Foundations of Computer Science*, pages 327–338, Portland, Oregon, Oct 1985. IEEE CS Press.
34. Edmund M. Clarke Vicky Hartonas-Garmhausen, Sérgio Vale Aguiar Campos. Probverus: Probabilistic symbolic model checking. In Joost-Pieter Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99, Bamberg, Germany, May 26-28, 1999. Proceedings*, volume 1601 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 1999.