Proceedings of: 12th Advanced Research Working Conference on *Correct Hardware Design and Verification Methods* (CHARME), 21-24 October 2003, L'Aquila (Italy), LNCS, Springer-Verlag

Co-sponsored by the IFIP TC10/WG10.5 Working Group on: Design and Engineering of Electronic Systems

# Finite Horizon Analysis of Markov Chains with the Mur$\varphi$ 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 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 Mur$\varphi$ verifier. We call the resulting probabilistic model checker FHP-Mur$\varphi$ (*Finite Horizon Probabilistic* Mur$\varphi$).
We present experimental results comparing FHP-Mur$\varphi$ with (a finite horizon subset of) PRISM, a state-of-the-art symbolic model checker for Markov Chains. Our experimental results show that FHP-Mur$\varphi$ can handle 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, 11, 16, 15, 21, 28] 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 behaviour, 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, 12]. 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. [32, 8, 13]). This motivates the development of *Probabilistic Model Checkers* [9, 1, 17], i.e. of model checking algorithms and tools whose goal is to automatically verify (probabilistic) properties

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

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".

Many methods have been proposed for probabilistic model checking, e.g. [10, 3, 8, 13, 14, 19, 24, 27, 32].

To the best of our knowledge, currently, the state-of-the-art probabilistic model checker is PRISM [25, 1, 18]. PRISM overcomes the limitations due to the use of linear algebra packages in Markov Chain analysis by using *Multi Terminal Binary Decision Diagrams* (MTBDDs) [6], a generalization of *Ordered Binary Decision Diagrams* (OBDDs) [4] 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 [7]) as well as a hybrid approach. The user can choose the best approach for the problem at hand.

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.

Indeed our experience shows that *Explicit Model Checking* can outperform *Symbolic Model Checking* in automatic analysis of *Hybrid Control Systems* [22]. 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.

- We present (Sections 3, 4) an *explicit* algorithm for automatic verification of discrete time/finite state Markov Chains. Given a Markov Chain $\mathcal{M}$, our algorithm checks wheter 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.
- We present (Sections 5) an implementation of our algorithm within the Mur$\varphi$ [21] verifier. We call the resulting probabilistic model checker FHP-Mur$\varphi$ (*Finite Horizon Probabilistic* Mur$\varphi$).
- We present (Section 6.1) experimental results comparing FHP-Mur$\varphi$ with PRISM on two suitably modified versions of the dining philosophers protocol included in the PRISM distribution. Our experimental results show that FHP-Mur$\varphi$ 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-Mur$\varphi$ (as to be expected).
Note however that PRISM can handle more general models than FHP-Mur$\varphi$, and can verify more general properties (namely all PCTL [13] properties) than FHP-Mur$\varphi$. In fact, FHP-Mur$\varphi$ can only verify finite horizon safety properties for Markov Chains, a subclass (although an important one) of the verification tasks that PRISM can handle.

– We present (Section 6.2) experimental results on using FHP-Mur$\varphi$ for a probabilistic analysis of a "real world" hybrid system, namely the Turbogas Control System of the Co-generative power plant described in [22]. 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-Mur$\varphi$ can complete (finite horizon) verification within reasonable time.

## 2 Basic Notation

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: $\mathbf{I}(s,j) = \textbf{if } (s = j) \textbf{ then } 1 \textbf{ else } 0$. 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 $\omega$. We denote with $|\pi|$ the length of $\pi$. We* *denote with* $Path(\mathcal{M}, s)$ *the set of infinite paths* $\pi$ *in* $\mathcal{M}$ *s.t.* $\pi(0) = s$. *If* $\mathcal{M} = (S, \mathbf{P}, q)$ *we write also* $Path(\mathcal{M})$ *for* $Path(\mathcal{M}, q)$.

3. *For* $s \in S$ *we denote with* $\sum(s)$ *the smallest* $\sigma$*-algebra on* $Path(\mathcal{M}, s)$ *which, for any finite path* $\rho$ *starting at* $s$, *contains the basic cylinders* $\{ \pi \in Path(\mathcal{M}, s) \mid \rho$ *is a prefix of* $\pi \}$. *The probability measure* $Pr$ *on* $\sum(s)$ *is the unique measure with* $Pr(\{\pi \in Path(\mathcal{M}, s) | \rho \text{ is a prefix of } \pi\}) = Pr(\rho) = \prod_{i=0}^{k-1} \mathbf{P}(\rho(i), \rho(i+1)) = \mathbf{P}(\rho(0), \rho(1))\mathbf{P}(\rho(1), \rho(2)) \cdots \mathbf{P}(\rho(k-1), \rho(k))$, *where* $k = |\rho|$.

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

## 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 $\phi$ (i.e. $\phi(s) = 1$). If $\phi$ models an *error condition* the above computation allows us to compute the probability of reaching an error condition in at most $k$ transitions.

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

**Definition 2.** *Let* $\mathcal{M} = (S, \mathbf{P}, q)$ *be a Markov Chain and let* $\phi$ *be a boolean function on* $S$, *i.e.* $\phi : S \to \mathcal{B}$. *We define Markov Chain* $\mathcal{M}_\phi$ *as follows.*
$\mathcal{M}_\phi = (S, \mathbf{P}_\phi, q)$, *where for all* $s, t \in S$, $\mathbf{P}_\phi(s,t) = \begin{cases} \mathbf{P}(s,t) & \text{if } \neg\phi(s) \\ 1 & \text{if } \phi(s) \wedge (s = t) \\ 0 & \text{if } \phi(s) \wedge (s \neq t) \end{cases}$

In other words, Markov Chain $(S, \mathbf{P}_\phi, q)$ is obtained from $(S, \mathbf{P}, q)$ by removing all outgoing edges from any state $s$ satisfying $\phi$ (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 $\phi$ is exactly the same as the probability of reaching in *at most* $k$ steps a state satisfying $\phi$. Note that according to item 1 of Definition 1 $(S, \mathbf{P}_\phi, q)$ is indeed a Markov Chain.

From the above considerations follow 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. [13, 1]).

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

*Example 1.* Consider Markov Chain $\mathcal{M} = (S, \mathbf{P}, \mathbf{q})$ with $S = \{1, 2\}, \mathbf{P} = \begin{bmatrix} 0.8 & 0.2 \\ 0.7 & 0.3 \end{bmatrix}$ 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.

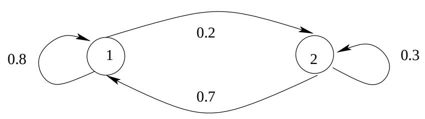


**Fig. 1.** A Markov Chain

Let $\phi$ be defined as follows: $\phi(s) = (s = 2)$, i.e. only state 2 satisfies $\phi$.
Then $\mathbf{P}_\phi = \begin{bmatrix} 0.8 & 0.2 \\ 0.0 & 1.0 \end{bmatrix}$.
From Theor. 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 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 3.** *A* Probabilistic Finite State System *(PFSS) $\mathcal{S}$ is a 3-tuple $(S, q, \texttt{next})$, where: $S$ is a finite set (of states), $q \in S$ and* next *is a function taking a state $s$ as argument and returning a set $\texttt{next}(s)$ of pairs $(t, p)$ s.t. $\sum_{(t,p) \in \texttt{next}(s)} p = 1$.*

To a PFSS we can associate a Markov Chain in a unique way.

**Definition 4.** *1. Let $\mathcal{S} = (S, q, \texttt{next})$ be a PFSS. The Markov Chain $\mathcal{S}^{mc} = (S, \mathbf{P}, q)$ associated to $\mathcal{S}$ is defined as follows: $\mathbf{P}(s, t) = \begin{cases} p & \text{if } (t, p) \in \texttt{next}(s) \\ 0 & \textit{otherwise} \end{cases}$*

*2. Given $k \in \mathbb{N}$ and a boolean function $\phi$ on $S$ we write $P(\mathcal{S}, k, \phi)$ for $P(\mathcal{S}^{mc}, k, \phi)$ as defined in Problem 1. Thus Problem 1 for PFSSs becomes: given a PFSS $\mathcal{S}$ compute $P(\mathcal{S}, k, \phi)$.*

Given a PFSS $\mathcal{S}$ we want to compute $P(\mathcal{S}, k, \phi)$ without generating the transition matrix for Markov Chain $\mathcal{S}^{mc}$. Using Proposition 1 this can be done as shown in Proposition 2.

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

*Proof.* (Sketch). *Let $\mathcal{M} = \mathcal{S}^{mc} = (S, \mathbf{P}, \mathbf{q})$. Consider the following sequence of distributions: $\mathbf{y}^{(0)} = \mathbf{q}$, $\mathbf{y}^{(i+1)} = \mathbf{y}^{(i)}\mathbf{P}_\phi$ for $i = 0, \ldots k$. From Proposition 1 we*

```
//For i = 1,...k+1, Q(i) is a queue of state-probability pairs (s,p);
P((S,q,next),k,φ) { i = 1; r = 0; enqueue(Q(i),(q,1));
forall i = 1...k do { while (Q(i) is nonempty) { (s,p) = dequeue(Q(i));
                  forall (t,a) ∈ next(s) do { if φ(t) { r = r + p * a; }
                        else enqueue(Q(i+1),(t,p * a)); } } } return(r); }
```

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

*have that* $\sum_{s:\phi(s)} \mathbf{y}^{(k+1)}(s) = P(\mathcal{S}^{mc}, k, \phi)$. *Moreover, from Fig. 2 we have that for all* $i = 1, \ldots k+1$, *and for all* $s \in S'$, $\mathbf{y}^{(i)}(s) \neq 0$ *iff* $(s, \mathbf{y}^{(i)}(s)) \in Q(i)$. *Note that states s s.t.* $\phi(s) = 1$ *are not enqueued. In fact, by Definition 2, the only state reachable form such a state s is s itself. Thus, from Definition 2 of* $\mathbf{P}_\phi$, *we have that the value r returned by* $P((S, q, \texttt{next}), k, \phi)$ *in Fig. 2 is exactly* $P(\mathcal{S}^{mc}, k, \phi)$.

*Remark 1.* Given a PFSS $\mathcal{S} = (S, q, \texttt{next})$, $k \in \mathbb{N}$, a boolean function $\phi$ on $S$ and a probability threshold $p$, in Section 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 validity of a *Finite Horizon Probabilistic* (FHP) *Safety Property*. FHP safety properties 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 [13] formula, namely $P_{<p}[\text{true } U^{\leq k} \phi]$. Thus also the probabilistic model checker PRISM [25] can be used to verify FHP safety properties.

Note however that PRISM can handle *all* PCTL formulas, whereas our algorithm can only handle FHP safety properties. In particular PRISM can verify properties like $P_{<p}[\text{true } U \phi]$ (*the probability of reaching a state satisfying* $\phi$ *is less than* $p$). Such *unbounded horizon* properties cannot be handled with our algorithm.

## 5 Analysing Probabilistic Systems with the Mur$\varphi$ 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 Mur$\varphi$ verifier. We call the resulting tool FHP-Mur$\varphi$ (*Finite Horizon Probabilistic* Mur$\varphi$).

### 5.1 Functions and Data Structures

FHP-Mur$\varphi$ input defines a PFSS $\mathcal{S} = (S, q, \texttt{next})$ to which we will refer in the sequel. The FHP-Mur$\varphi$ keyword `startstate` defines $\mathcal{S}$ initial state $q$. Indeed, Mur$\varphi$ can have a set of initial states, however, w.l.o.g. in the following we assume we have just one initial state. FHP-Mur$\varphi$ keyword `invariant` defines the boolean function $\phi$ on $S$ as well as the probability threshold $\beta$ s.t. $P(\mathcal{S}, k, \phi) < \beta$ must hold (Remark 1).

The meaning of the declarations in Fig. 3 is as follows. Constant `k` (implementing $k$) is our verification horizon and is given to FHP-Mur$\varphi$ as a command line parameter. Functions `Phi()` implements $\phi$. Function `next()` is the nextstate function of the PFSS $\mathcal{S}$ defined by FHP-Mur$\varphi$ input. Thus function `next()` takes a state $s$ as argument and returns the set `next`$(s)$ of pairs $(t, p)$ s.t. $s$ goes to $t$ with probability $p$. 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 in Fig. 2. 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 [31]. This allows us to handle fairly large state spaces. The hash table `M` is a cache whose entries are pairs $(s, p)$ as for queues `Q_old`, `Q_new`. Constant `max_prob_Phi` (implementing $\beta$) defines our probability threshold, i.e. the max allowed value for the probability `prob_Phi` of reaching (within the given horizon `k`) an error state (i.e. a state $s$ s.t. `Phi`$(s)$ = `true`).

Note that from the above discussion follows that Mur$\varphi$ *hash compaction* (`-c`) [21] has no effect in FHP-Mur$\varphi$ since no FHP-Mur$\varphi$ data structure uses *state signatures* [29, 30].

```
int k; /* is the horizon, i.e. the max allowed number of steps
          to reach a state violating our invariant */
boolean Phi(state s);      state_probability_pairs next(state s);
Queue Q_old, Q_new;        Cache M;
double prob_Phi; /* incrementally stores the probability of
                    violating the invariant in at most k steps */
double max_prob_Phi; /* is the max allowed value for prob_Phi */
```

**Fig. 3.** Functions and Data Structures

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

Our main function `Search()` is shown in Fig. 4. This function efficiently implements the computation described in Fig. 2.

Function `Insert()` is shown in Fig. 4. This function uses a cache table `M` in RAM to save queue space and thus computation time. `M[h]` returns the pair $(s, p)$ stored in entry `h` of `M`. `M[h].state` denotes $s$ and `M[h].prob` denotes $p$.

Every time it is necessary to enqueue a new pair (state $s$, probability $p$), `Insert(s, p)` is called. If state $s$ is already stored in cache `M`, we simply update the stored probability in `M`, adding $p$ to it. If state $s$ is not stored in `M`, we check if the slot in `M` in which we have to put $s$ is free. If it is free then we insert pair $(s, p)$ in `M`. If it is not free, we call function `Checktable()` to empty `M` and then we insert pair $(s, p)$ in `M`.

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 $q$. Using `M` rather than $w$ copies of $s$ we have just one or slightly more than one (depending on how large is `M`). 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()` is shown in Fig. 4. Function `Insert_in_table()` calculates the hash value `h` of `s`. If `M[h]` is a free slot, `Insert_in_table()` 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()` is shown in Fig. 4. It is the only function that enqueues values in `Q_new`; it simply flushes `M` into `Q_new`.

Function `Checktable()` is used by function `Insert()` to free `M` when a collision occurs. It 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 effectiveness of our approach we run two kind of experiments.

First, in Section 6.1, we compare FHP-Mur$\varphi$ with the probabilistic model checker PRISM [25].

Second, in Section 6.2, we run FHP-Mur$\varphi$ on a quite large probabilistic hybrid systems. Since our main goal is to use FHP-Mur$\varphi$ on hybrid systems, this second kind of evaluation is very interesting for us.

### 6.1 Probabilistic Dining Philosophers

In this Section we give our experimental results on using FHP-Mur$\varphi$ on the probabilistic protocols included in PRISM distribution [25]. We do not consider the protocols that lead to Markov Decision Processes or to Continuous Time Markov Chains, since FHP-Mur$\varphi$ cannot deal with them. Hence we only consider Pnueli-Zuck [23] and Lehmann-Rabin [20, 26] probabilistic dining philosophers protocols. Moreover, we modify PRISM definitions for such protocols in order to have a finite horizon property to verify with FHP-Mur$\varphi$. In fact, FHP-Mur$\varphi$ 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 $P_{<p}[\text{true}\ U^{\leq k}\phi]$.

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

Our modifications to PRISM protocols consist in adding variables to count the number of times that a philosopher fails in getting both forks. 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 verify *quality of*

```
int Search() {
  prob_Phi = 0;
  enqueue(Q_old, (q, 1));  /* enqueue initial state q */
  for (level = 1; level <= k; level++) {
    clear cache table M;
    while (Q_old is not empty) {
      (s, p) = dequeue(Q_old);
      for all (s', a) in next(s) {
        if (Phi(s')) {
          prob_Phi = prob_Phi + p*a;
          if (prob_Phi >= max_prob_Phi)
            return(0); /* property does not hold */
        } else Insert(s', p*a);
      } /* for all */
    } /* 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;
    M[h] = (s, prob); /* new probability of s is prob */
  } else {
    collision = Insert_in_table(s, p);
    if (collision) {
      Checktable(); /* there is space to insert now */
      Insert_in_table(s, p);
    }
  }
} /* Insert() */

boolean 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; /* M is empty now */
}  /* Checktable() */
```

**Fig. 4.** Functions: `Search()`, `Insert()`, `Insert_in_table()`, `Checktable()`

*service* properties, which are very frequent in practice. E.g., in the Pnueli-Zuck protocol, we changed the code fragment in Fig. 5 with the one in Fig. 6.

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 PRISM distribution [25]).

Fig. 7 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 $P(\texttt{MAX\_CONT}, k)$ the value of $p$ does not matter.

In Fig. 8 we have the FHP-Mur$\varphi$ code corresponding to the PRISM code fragment of Fig. 6. Of course FHP-Mur$\varphi$ input language is the same as Mur$\varphi$ one [21], only FHP-Mur$\varphi$ has probabilities rather than booleans on rule guards. FHP-Mur$\varphi$ invariant `invariant p` $\gamma$ requires that with probability at least $p$ "all states reachable in at most $k$ steps from the initial state satisfy $\gamma$" ($k$ is FHP-Mur$\varphi$ horizon). Thus, using the notation in Section 5 we have that: $\phi = \neg\gamma$ and the probability threshold (`max_prob_Phi` in Fig. 3) is $(1 - \texttt{p})$.

Note that in Fig. 8 the probability threshold for FHP-Mur$\varphi$ invariant is 0, so that FHP-Mur$\varphi$ will not stop verification before completing all levels of the BF computation. This forces FHP-Mur$\varphi$ to compute $P(\texttt{MAX\_CONT}, k)$.

To assess FHP-Mur$\varphi$ effectiveness in Figs. 9, 10 we compare the results obtained with FHP-Mur$\varphi$ and with PRISM on, respectively, Pnueli-Zuck and Lehmann-Rabin protocols (modified as described above).

From Fig. 9 we can see that, for Pnueli-Zuck algorithm, when $\texttt{NPHIL} = 5$ (5 philosophers) and `MAX_CONT` is 4, PRISM is unable to complete any verification within 2GB of RAM, independently on which of the 3 PRISM verification algorithms (totally MTBDD based, algebraic and hybrid) is chosen. Similarly, for the Lehmann-Rabin algorithm, in Fig. 10 we see that when `NPHIL` is 4, and `MAX_WAIT` is 3, then PRISM is unable to complete the verification task in the same environment as above.

FHP-Mur$\varphi$ was always able to complete all given verifications tasks. Note however that, as it can be seen from Figs. 9 and 10, for the verifications tasks in which PRISM terminates, PRISM is always faster than FHP-Mur$\varphi$.

Our experimental results show that for probabilistic protocols involving arithmetical computations FHP-Mur$\varphi$ is to be considered among the available (and valuable) tools for automatic finite horizon analysis of safety properties.

As for the numerical quality of FHP-Mur$\varphi$ we have that when both PRISM and FHP-Mur$\varphi$ terminate both give the same value for $P(\texttt{MAX\_CONT}, k)$ (column *Probability* in Figs. 9, 10).

### 6.2 Analysis of a Probabilistic Hybrid Systems with FHP-Mur$\varphi$

In this section we show our experimental results on using FHP-Mur$\varphi$ 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).

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

**Fig. 5.** Pnueli-Zuck algorithm fragment to be modified in PRISM.

```
module phil1
  p1: [0..10] init 0;
  cont1: [0..3] 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. 6.** Pnueli-Zuck algorithm modified fragment in PRISM.

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

**Fig. 7.** PCTL formula 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. 8.** Pnueli-Zuck algorithm in FHP-Mur$\varphi$.

| NPHIL | MAX_WAIT | Probability | Mur$\varphi$ memory | PRISM memory | Mur$\varphi$ 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 |

**Fig. 9.** Results on a machine with 2 processors (both INTEL Pentium III 500Mhz) and 2GB of RAM. Mur$\varphi$ options: `-b` (bit compression), `-m200` (use exactly 200MB of RAM), `-max120` (the finite horizon is 20). The last verification had `-m1000` (use exactly 1GB of RAM). PRISM options: default options. N/A means that PRISM was unable to complete the verification; in this case, also the `-m` and `-s` (totally MTBDD and algrebraic verification algorithm respectively) have been used, with the same result. Memory occupations are in MB, time is in seconds.

| NPHIL | MAX_WAIT | Probability | Mur$\varphi$ memory | PRISM memory | Mur$\varphi$ time | PRISM time |
|---|---|---|---|---|---|---|
| 3 | 3 | 4.8039366e-06 | 800 | 39.0625 | 1040.330 | 84.556 |
| 3 | 4 | 0. | 800 | 70.1483 | 1041.700 | 121.147 |
| 4 | 3 | 5.609882064e-08 | 800 | N/A | 34307.740 | N/A |

**Fig. 10.** W.r.t. Fig. 9, the only change is in the Mur$\varphi$ option `-m800` (use exactly 800MB of RAM).

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 [22] it is shown that by adding finite precision real numbers to Mur$\varphi$, we can use Mur$\varphi$ to automatically verify TCS. In particular in [22] 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.*

In [22] the user demand has been modeled rather roughly, using nondeterministic automata. Here we show that using FHP-Mur$\varphi$ we can define and, more importantly, automatically analyse, 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:

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

where $M$ =`MAX_U` (maximum user demand value) and $\alpha$ =`MAX_D_U`.

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

$$u(t+1) = \begin{cases} min(u(t)+\alpha, M) & \text{with probability } p(u(t), 1) \\ u(t) & \text{with probability } p(u(t), 0) \\ max(u(t)-\alpha, 0) & \text{with probability } p(u(t), -1) \end{cases} \quad (2)$$

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

With FHP-Mur$\varphi$ the definition of Markov Chain (2), starting from the TCS model, is quite simple. This is done in Fig. 11, where `user_demand(u, d_u)` computes $p(u, d\_u)$ (1) and function `main` updates the system state, in particular updates `u` as described in (2).

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

**Fig. 11.** Rulesets with probabilistic *user demand*

In Fig. 12 we report the results of some verification runs done by FHP-Mur$\varphi$ with $\beta = 0.4$.

We are interested in cases where the error probability is greater than 0 (zero). From the results in [22] 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 the transition graph diameter. In our experiments here we choose our horizon as follows. Let $\text{Diam}(n)$ be the diameter of TCS transition graph when `MAX_D_U` $= n$. We set our horizon $k$ to be equal to $\lceil \frac{\text{Diam}(n)}{100} \rceil 100$. In this way we check the error probability in the error neighborhood.

| `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 |

**Fig. 12.** Results on a machine with 2 processors (both INTEL Pentium III 500Mhz) and 2GB of RAM. Mur$\varphi$ options used: `-b` (bit compression), `-m500` (use 500 MB of RAM). Time is given in seconds.

Fig. 12 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. This kind of evaluations are not possible with the nondeterministic verification of TCS carried out in [22].

## 7 Conclusions

We presented (Sections 3, 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 presented (Section 5) an implementation of our algorithm within a suitable extension of the Mur$\varphi$ verifier that we call FHP-Mur$\varphi$ (*Finite Horizon Probabilistic*-Mur$\varphi$).

We presented (Section 6) experimental results comparing FHP-Mur$\varphi$ with (a finite horizon subset of) PRISM, a state-of-the-art symbolic model checker for Markov Chains. Our experimental results show that FHP-Mur$\varphi$ 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] C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. *Automata, Languages and Programming*, pages 430–440, 1997.

[2] E. Behrends. *Introduction to Markov Chains*. Vieweg, 2000.

[3] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.

[4] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8), 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. *Information and Computation*, 98, 1992.

[6] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. *Proc. 30th ACM/IEEE Design Automation Conference*, pages 54–60, 1993.

[7] url: `http://vlsi.colorado.edu/~fabio/`.

[8] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Proc. of FOCS'88*, pages 338–345. IEEE CS Press, 1988.

[9] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4):857–907, 1995.

[10] L. de Alfaro. Formal verification of performance and reliability of real-time systems. Technical report, Stanford University, 1996.

[11] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992.

[12] H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. Elsevier, 1994.

[13] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6:512–535, 1994.

[14] S. Hart and M. Sharir. Probabilistic temporal logic for finite and bounded models. In *Proc. of 16th ACM Symposium on Theory of Computing*, pages 1–13. ACM, 1984.

[15] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, New Jersey, 1991.

[16] G. J. Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[17] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In P. Kemper, editor, *Proc. Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 7–12, September 2001. Available as Technical Report 760/2001, University of Dortmund.

[18] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with prism: A hybrid approach. In *Proc. TACAS'02*, volume 2280. LNCS, Springer Verlag, April 2002.

[19] K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94:1–28, 1991.

[20] D. Lehmann and M. Rabin. On the advantages of free choice: A symmetric fully distributed solution to the dining philosophers problem (extended abstract). *Proc. 8th Symposium on Principles of Programming Languages*, pages 133–138, 1981.

[21] url: `http://sprout.stanford.edu/dill/murphi.html`.

[22] G. Della Penna, B. Intrigila, I. Melatti, M. Minichino, E. Ciancamerla, A. Parisse, E. Tronci, and M. V. Zilli. Automatic verification of a turbogas control system with the murphi verifier. In *Proc. of 6th International Workshop on: Hybrid Systems: Computation and Control (HSCC)*, LNCS, Prague, Czech Republic, April 2003. Springer.

[23] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, 1986.

[24] A. Pnueli and L. Zuck. Probabilistic verification. *Information and Computation*, 103:1–29, 1993.

[25] url: `http://www.cs.bham.ac.uk/~dxp/prism/`.

[26] N. Lynchand I. Saias and R. Segala. Proving time bounds for randomized distributed algorithms. In *Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 314–323, 1994.

[27] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In *Proc. of CONCUR*, number 836 in LNCS, pages 381–496. Springer, 1994.

[28] url: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[29] U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *IFIP WG 10.5 Advanced Research Working Conference on: Correct Hardware Design and Verification Methods (CHARME)*, pages 206–224, 1995.

[30] U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *IFIP TC6/WG6.1 Joint International Conference on: Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, 1996.

[31] E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. Exploiting transition locality in automatic verification. In *IFIP WG 10.5 Advanced Research Working Conference on: Correct Hardware Design and Verification Methods (CHARME)*. LNCS, Springer, Sept 2001.

[32] M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. of FOCS'85*, pages 327–338. IEEE CS Press, 1985.