Randomization Helps in LTL Model Checking*

Luboš Brim, Ivana Černá, and Martin Nečesal

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic
{brim,cerna,xnecesal}@fi.muni.cz

Abstract. We present and analyze a new probabilistic method for automata based LTL model checking of non-probabilistic systems with intention to reduce memory requirements. The main idea of our approach is to use randomness to decide which of the needed information (visited states) should be stored during a computation and which could be omitted. We propose two strategies of probabilistic storing of states. The algorithm never errs, i.e. it always delivers correct results. On the other hand the computation time can increase. The method has been embedded into the SPIN model checker and a series of experiments has been performed. The results confirm that randomization can help to increase the applicability of model checkers in practice.

1 Introduction

Model checking is one of the major recent success stories of theoretical computer science. Model checkers are tools which take a description of a system and a property and automatically check whether the system satisfies the property. There are now many different varieties of model checkers including model checkers for real-time systems and probabilistic systems.

Practical application of model checking in the hardware verification became a routine. Many companies in the hardware industry use model checkers to ensure the quality of their products. With the debugging potential afforded by model checking, design of hardware components can be made much more reliable and moreover model checking is seen to accelerate the design process, significantly decreasing the time to market. However, the situation in software model checking is completely different. Software is much more complicated system due to its size and dynamic nature. To achieve similar benefits as in hardware verification, additional methods and techniques need to be explored.

One of the very successful techniques is randomization. The term "probabilistic model checking" (or "probabilistic verification") refers to a wide range of techniques. There are two ways in which probability features in this area. The first approach concerns applying model checking to systems which inherently include probabilistic information [11, 4, 1, 2]. The second approach concerns systems which are non-probabilistic, but of size which makes exhaustive checking

* This work has been partially supported by the Grant Agency of Czech Republic grants No. 201/00/1023 and 201/00/0400.

impractical or infeasible [9, 5]. The aim is to use randomization to make model checking more efficient, albeit at a cost of establishing satisfaction with high probability, possibly with a one-sided error, rather than certainty, or at a cost of other resources. While the topic of verification of probabilistic systems has been intensively studied, there are only a few attempts to use randomization in verification of non-probabilistic systems.

In the paper we focus on automata based LTL model checking of non-probabilistic systems. Our aim is to attack the *state-explosion* problem (the number of reachable states grows exponentially in the number of concurrent components and is the main limitation in practical applications of model checkers). Various techniques and heuristics reducing the random access memory required have been proposed. One possible solution (called *on-the-fly* model checking) is to generate only the part of the state graph required to validate or disprove the given property. On-the-fly algorithms generate the state space in a depth-first manner and keep only track of reached states to avoid doing unnecessary work. Another solution makes use of the fact that one of the reasons of the state explosion problem is the generation of all interleavings of independent transitions in different concurrent components. *Partial order reduction* techniques were introduced to ensure that many of these unnecessary interleavings are not explored during state generation.

If we have some knowledge about the structure of the state graph in advance (before starting the actual verification), we can apply even more efficient heuristics. As in general it is not the case we suggest to use a probabilistic method which can be viewed as a probability distribution on a set of deterministic techniques. We explore two probabilistic approaches to achieve significant space reduction in the depth first search based model checking of non-probabilistic systems.

The core of the first approach is to use randomness to decide which of the needed information (visited states) should be stored during a computation and which could be omitted. Consequently, the time complexity of the computation can increase. The second method simply implements the idea of randomizing the branching structure. Both methods are of Las Vegas type, i.e. they always deliver the correct answer. In the paper we focus on the first method and report on the second one briefly. We stress that both methods are compatible (can be used simultaneously) with on-the-fly and partial order reduction techniques. We have implemented both methods and the experiments gave surprisingly very good results in competition with non-probabilistic approaches.

The paper is organized as follows. We first review some background on model checking using automata, define the corresponding graph-theoretic problem, and briefly discuss possible sources for applying randomization. Then we propose the probabilistic reduction algorithm and report experimental results achieved. We conclude with the description of the second method and with some final remarks.

2 Problem Setting

We consider the following verification problem. A finite state transition graph (also called a Kripke structure) is used to represent the behavior of a given system and a linear temporal logic (LTL) formula is used to express the desired property of the system. The basic idea of automata-based LTL model checking is to associate with each LTL formula a Büchi automaton that accepts exactly all the computations that satisfy the formula. If we consider a Kripke structure to be a Büchi automaton as well, then the model checking can be described as a language containment problem and consequently as a non-emptiness problem of (intersecting) Büchi automata. A Büchi automaton accepts some word iff there exists an accepting state reachable from the initial state and from itself. Hence, we can sum up the model checking problem we consider as the following graph-theoretic problem.

Non-emptiness problem of Büchi automata.
Given a directed graph $G = (V, E)$, start state (vertex) $s \in V$, a set of accepting states $F \subseteq V$, determine whether there is a member of F which is reachable from s and belongs to a nontrivial strongly connected component of G.

The direct approach to solve the problem is to decompose the graph into nontrivial strongly connected components (SCCs), which can be done in time linear in the size of the graph using the Tarjan's algorithm [10]. However, constructing SCCs is not memory efficient since the states in the SCCs must be explicitly stored during the procedure. Courcoubetis et al. [3] have proposed an elegant way to avoid the explicit computation of SCCs. The idea is to use a *nested depth-first search* to find accepting states that are reachable from themselves (to compute *accepting path*). The pseudo-code of the NestedDFS algorithm is given in Fig. 1. Only two bits need to be added to each state to separate the states stored in *VisitedStates* during the first and the second (nested) DFS. The extreme space efficiency of the NestedDFS algorithm is achieved to the detriment of time. The time might double when all the states are reachable in both searches and there are no accepting cycles. However, in applications to real systems the space is actually more critical resource. This makes the nested depth-first search the main algorithm used in many verification tools which support the automata based approach to model checking of LTL formulas (e.g. SPIN).

The space requirements of the NestedDFS algorithm are determined by the necessity of storing *VisitedStates* in randomly accessed memory. Several implementations of NestedDFS use different data structures to represent the set *VisitedStates*. The basic one is a *hash table* [6]. Another implementation [12] makes use of symbolic representation of *VisitedStates* via *Ordered Binary Decision Diagrams* (OBDD).

Hash compaction is used in [14], where the possible hash-collisions are not re-solved. The algorithm can thus detect a state as visited even if it is not. Consequently, not all reachable states are explored during the search, and an error might go undetected.

```
proc DFS(s)
  add {s, 0} to VisitedStates;
  foreach successor t of s do
    if {t, 0} not in VisitedStates then DFS(t) fi
  od;
  if accepting(s) then seed := s; NDFS(s) fi
end


proc NDFS(s)
  add {s, 1} to VisitedStates;
  foreach successor t of s do
    if {t, 1} not in VisitedStates
      then NDFS(t)
       else if t = seed then "report cycle" fi fi
  od
end
```

Fig. 1. Algorithm NestedDFS

Another technique which has been investigated to reduce the amount of randomly accessed memory is *state-space caching* [5]. The idea is based on the observation that when doing a depth-first search of a graph, storing only the states that are on the search stack is sufficient to guarantee that the search terminates. While this can produce a very substantial saving in the use of randomly accessed memory, it usually has a disastrous impact on the run time of the search. Indeed, each state will be visited as many times as there are simple paths reaching it. An improvement on this idea is to store not only the states that are on the search stack, but also a bounded number of other states (as many as will fit into the chosen "state-space cache"). If the state-space cache is full when a new state needs to be stored, random replacement of a state that is not currently on the search stack is used.

The advantage of state-space caching is that the amount of memory that is used can be reduced with a limited impact on the time required for the search. Indeed, if the cache is large enough to contain the whole state space, there is no change in the required time. If the size of the cache is reduced below this limit, the time required for the search will only increase gradually. Experimental results, however, show that below a threshold that is usually between 1/2 and 1/3 of the size of the state space, the run time explodes, unless additional techniques are used to restrict the number of distinct paths that can reach a given state [5].

The behavior of state-space caching is quite the opposite of that of the hashing technique. Indeed, state-space caching guarantees a correct result, but at the cost of a potentially large increase in the time needed for the state-space search. On the other hand, hashing never increases the required run time, but can fail to

explore the whole state space. A combination of state space caching and hashing has been proposed and investigated in [9].

In this paper we propose a new technique to attack the state-explosion problem using a simple *probabilistic method*. Actually, the technique has been strongly motivated by our intention to improve the performance of the model checker SPIN, and the technique has been embedded into SPIN for testing purposes.

The proposed method allows to solve the emptiness problem of Büchi automata (i.e. complete LTL model checking and not only reachability) and it never errs. It can be briefly described in the following way. The algorithm is based on the nested depth-first search as described in Fig. 1. Each time the algorithm backtracks through a state it employs a proper reduction strategy to decide whether the state will be kept in the *VisitedStates* table or whether it will be removed. We propose two reduction strategies, the dynamic and the static one. While the first one takes on the frequency of visiting the state, the second one allows to eliminate delayed storing of the state and thus decreases the number of visits of individual states. We specify properties of systems determining which strategy suits better for a given verification problem.

3 Algorithm with Probabilistic Reduction Strategy

The reason to store the states in the table of visited states during nested depth-first search is to speed up the verification by preventing the multiplication of work when states are re-visited. A state that is visited only once need not be stored at all, while storing a state which will be visited many times can result in a significant speed-up. The standard nested depth-first search algorithm stores *all* visited states. On the other side, the optimal strategy for storing states would take into account the number of times a state will be eventually visited – a *visitation factor*. As it is generally impossible to compute this parameter in advance, we will use probabilistic method to solve the problem.

The pseudo-code of the modified nested depth-first-search algorithm with reduction strategy, NestedDFSReSt, is given in Fig. 2. Whenever the *DFS* procedure explores a new state, the state is temporally saved in the *VisitedStates* table (with parameter 0). Whenever *DFS* backtracks trough a state, a test *ReductionStrategy* is performed and if the test evaluates to *true* the state is *removed* from the *VisitedStates* table. We will consider two basic probabilistic strategies of removing states. The first one *dynamically* decides on removing a state each time the state is backtracked through, while the second heuristic decides randomly in advance (before the verification is started) which states will be stored permanently.

As in the case of *DFS*, the *NDFS* procedure also needs the list of states it has visited to be efficient. Therefore every exploring of a new state results in its saving to the *VisitedStates* table (with parameter 1). Whenever *NDFS* backtracks trough a state it respects the *ReductionStrategy* test performed on this state by the *DFS* procedure and if necessary removes the state from the table.

```
proc DFS(s)
  add {s,0} to VisitedStates;
  foreach successor t of s do
    if {t,0} not in VisitedStates then DFS(t) fi
  od;
  if accepting(s) then seed := s; NDFS(s) fi;
  if ReductionStrategy(s) then delete {s,*} from VisitedStates fi
end

proc NDFS(s)
  if accepting(s) and {s,0} not in VisitedStates then exit fi;          (×)
  add {s,1} to VisitedStates;
  foreach successor t of s do
    if {t,1} not in VisitedStates
      then NDFS(t)
       else if t = seed then "report cycle" fi fi
  od;
  if {s,0} not in VisitedStates then delete {s,1} from VisitedStates fi
end
```

Fig. 2. Algorithm NestedDFSReSt

Removing states from the *VisitedStates* table has direct impact on the time complexity of the algorithm as re-visiting a state removed from the table invokes a new search from this state.

The correctness of the NestedDFSReSt algorithm follows from the correctness of the NestedDFS algorithm [3]. The additional key arguments it depends on are summarized in the following two lemmas.

Lemma 1. *During the whole computation the sequence of states with which the DFS procedure is called (DFSstack) forms a path in the graph G. The same is true for the NDFS procedure and NDFSstack.*

Proof: The *(N)DFS* procedure is always called with the argument t which is a successor of the current state s.

Lemma 2. *Suppose that during the whole computation both the DFSstack and the NDFSstack are subsets of* VisitedStates*, then the* NestedDFSReSt *algorithm terminates.*

Proof: From the inclusion follows that the *(N)DFSstack* always forms a simple path. The number of simple paths in G is finite and each one is explored at most once.

Theorem 1. *The algorithm* NestedDFSReSt *is correct.*

Proof: Whenever the *(N)DFS* procedure explores a new state, the state is temporally saved in the *VisitedStates* table. Therefore (N)DFSstack $\subseteq$ *VisitedStates* is invariantly true and NestedDFSReSt always terminates due to the Lemma 2. If NestedDFSReSt reports "cycle" then due to the Lemma 1 there is a reachable cycle containing an accepting state. Conversely, suppose there is a reachable cycle containing an accepting state in G. Deleting states from *VisitedStates* table cannot cause leaving out any call of *(N)DFS(t)* which would have been performed by NestedDFS algorithm. Moreover, the situation in which the condition of the **if** test on the very first line (denoted by $\times$) in *NDFS* is true is equivalent to the situation when $\{s, 1\}$ is in *VisitedStates* in NestedDFS algorithm. Therefore NestedDFSReSt searches trough all the paths NestedDFS does and thus reports "cycle" when NestedDFS does. ■

Notice that the test on the first line ($\times$) of *NDFS* prevents re-searching of an accepting state and thus speeds-up significantly the overall time complexity. This fact was confirmed also by experimental results.

The proof of the Theorem 1 is based on the fact that the NestedDFSReSt algorithm searches through all the paths the NestedDFS one does. Due to this fact our algorithm is compatible with additional techniques used for state space reductions, especially with partial order reduction techniques used in SPIN.

3.1 Dynamic Reduction Strategy

The pseudo-code implementing the dynamic reduction strategy is as follows:

```
funct ReductionStrategy-Dynamic(s) : boolean
  p := random[0,1];
  if p ≤ P_del
    then ReductionStrategy-Dynamic := true
     else ReductionStrategy-Dynamic := false fi
end
```

P_{del} is a fixed parameter determining the probability of deleting a state from *VisitedStates* table. Each time the *DFS* backtracks through a state s the state is deleted with the probability P_{del} and is kept stored with the probability $P_{sto} = 1 - P_{del}$. Once a state is kept stored in the table by the *DFS* procedure, it is never removed. The probability that a state will be eventually stored thus depends on the number k of its visits during the computation and is equal to $Prob(s \text{ is eventually stored}) = 1 - P_{del}^k$. This means that a state with higher visitation factor k has also higher probability to be stored permanently. The probability that the state s will be re-visited more than i times is equal to $Prob(s \text{ is } i \text{ times deleted}) = P_{del}^i$.

The dynamic reduction strategy would lead to a non-trivial reduction of randomly accessed memory if there is a non-trivial subset of the state space that will never be permanently stored. The expected memory reduction can be expressed as $P \times (\textit{size of the state space})$, where P is the probability that a state

will never be permanently stored. If k is the average visitation factor then P can be estimated as P_{del}^k. Therefore, we would like to have the highest possible value of the probability that a state will never be permanently stored.

On the other hand, not saving a frequently visited state increases the time complexity of the whole computation. Therefore, we are interested in the expected number of visits after which the state is stored permanently. Consider an elementary event $\{s$ is permanently stored during its i-th visit$\}$. Then

$$Prob(\{s \text{ is permanently stored during its } i\text{-th visit}\}) = P_{del}^{i-1} P_{sto}.$$

Let H be a random variable over the above mentioned elementary events defined as

$$H(\{s \text{ is permanently stored during its } i\text{-th visit}\}) = i.$$

We have that the expected value of H is

$$\begin{aligned} E(H) &= \sum_{i=1}^{\infty} i P_{del}^{i-1} P_{sto} = P_{sto} \sum_{i=1}^{\infty} i P_{del}^{i-1} = P_{sto} \sum_{j=1}^{\infty} \sum_{i=j}^{\infty} P_{del}^{i-1} = \\ &= P_{sto} \sum_{j=1}^{\infty} \frac{P_{del}^{j-1}}{1 - P_{del}} = \frac{P_{sto}}{1 - P_{del}} \sum_{j=0}^{\infty} P_{del}^{j} = \frac{P_{sto}}{1 - P_{del}} \frac{P_{del}^0}{1 - P_{del}} = \\ &= \frac{P_{sto}}{(1 - P_{del})^2} = \frac{P_{sto}}{P_{sto}^2} = \frac{1}{P_{sto}} \end{aligned}$$

It can be seen that the expected value of the random variable H depends on the probability P_{sto} and indicates that value P_{sto} should be high.

We can conclude that in systems with a high visitation factor we cannot expect reasonable space savings without enormous increase of the time complexity.

3.2 Static Reduction Strategy

The second strategy tries to eliminate the main disadvantage of the dynamic reduction strategy, namely the delayed storage of a state. If a state will eventually be permanently stored, why not to store it immediately during the first visit. When deciding which states are to be stored we should prefer states with high visitation factor. As we cannot compute this factor in advance we use probabilistic decision. All states are *in advance* and *randomly* divided into two groups: states which will be stored and those which will never be stored (represented as R). Hence, each state is permanently stored during its first visit or never. The ratio between stored and non-stored states is selected with the intention to achieve as highest reduction in state space as possible.

The pseudo-code implementing the static reduction strategy is as follows:

```
funct ReductionStrategy-Static(s) : boolean
  if s ∈ R
    then ReductionStrategy-Static := true
     else ReductionStrategy-Static := false fi
end
```

The disadvantage of the static reduction strategy is its insensibility to the visitation factor.

4 Experiments

To be able to compare experimentally our probabilistic algorithm with the non-probabilistic one, we have embedded the algorithm into SPIN model checker.

We have performed a series of tests on several types of standard parametrized (scalable) verification problems. Here we report on two of them only:

Peterson Peterson's algorithm solves the mutual exclusion problem. We have considered the algorithm for parameter $N = 3$ determining the number of processes. The property to be verified was $\Box(ncrit < 2)$ (no more than one process is in critical section).

Philosophers Dining Philosophers is a model of a problem of sharing of resources by several processes. We have considered the algorithm for $N = 4$ and $N = 6$. The property to be verified was $\Box\Diamond(EatingAny = 1)$ (absence of deadlock).

The other problems we have considered were e.g. the Leader Election problem, Mobile processes. In all these experiments we have obtained similar results.

As our algorithm is compatible with partial order reduction techniques used in SPIN we have compiled all problems with partial order reductions.

For each verification problem we first give two most important characteristics of the computation performed by SPIN checker: *States* (the number of states saved in the *VisitedStates* table) and *Transitions* (the number of performed transitions). The number of transition is proportional to the overall time complexity of the computation. The size of the *VisitedStates* table in SPIN's computation is nondecreasing. Once a state is stored in the table it is never removed. On the other hand in the NestedDFSReSt algorithm every visited state is temporally stored in the table and only when it is backtracked through the (random) decision about its permanent storing is made. Therefore for our algorithm we need another characteristic, namely the highest size of the *Visited States* table, *Peak States*. The parameter *States* declares the number of states stored in the table at the end of computation. The remaining two parameters, *State Saving* and *Transition Overhead*, compare performance of the deterministic algorithm and the probabilistic ones. Computations of probabilistic algorithms were repeated 10 times, presented values are the average ones.

Peterson's Algorithm

Results of experiments are summarized in the Table 1. The best results with Dynamic Strategy were achieved for storing probability 0.5 where saving in the size of stored state space was 33% while increase in the time was negligible, and for probability 0.1 with 52% space saving and multiplication factor 4 of time. To get deeper inside we mention that the computation without the reduction

	States	Peak	Saving	Transitions	Overhead
SPIN	17068	17068	0%	32077	1.00
Dynamic Strategy					
$P_{sto} = 0.50$	10998	11421	33%	46074	1.44
$P_{sto} = 0.10$	6724	8263	52%	136344	4.25
$P_{sto} = 0.01$	5559	7407	57%	1110526	34.62
Static Strategy					
$P_{sto} = 0.75$	12807	12812	25%	38761	1.21
$P_{sto} = 0.50$	8568	9661	43%	63662	1.98
$P_{sto} = 0.40$	6852	8417	51%	390737	12.18

Table 1. Summary of Experimental Results for **Peterson**

strategy took about 1.5 second in this case. Yet another increase in the deleting probability results in substantial grow of time but does not improve space saving factor significantly.

	States	Peak	Saving	Transitions	Overhead
SPIN	3727	3727	0%	18286	1.00
Dynamic Strategy					
$P_{sto} = 0.50$	3047	3178	15%	33475	1.83
$P_{sto} = 0.10$	2482	2686	28%	139263	7.62
$P_{sto} = 0.01$	2316	2531	32%	1287156	70.39
Static Strategy					
$P_{sto} = 0.75$	2788	2961	21%	49112	2.69
$P_{sto} = 0.60$	2221	2577	31%	232973	12.74
$P_{sto} = 0.50$	1875	2340	37%	3285607	179.68

Table 2. Summary of Experimental Results for **Philosophers** with $N = 4$

Experiments with Static Strategy reveal that we can achieve 43% space saving for the price of double time complexity. 51% space saving is attained with worse time multiplication factor (12 in comparison to 4) than in the case of Dynamic Strategy. The difference between storing probability and real space savings (i.e. for storing probability 0.4 we would expect 60% saving instead of measured 51%) has two reasons. Firstly, as we do not know which states of the state space are actually reachable in the verified system we have to divide the whole state space in advance. Secondly, the division determines states which are permanently saved but the *VisitedStates* table contains also temporally saved states and its size can be temporally greater (parameter *Peak States*). State space saving is computed via comparing the number of saved states by non-probabilistic computation and the peak value of probabilistic computation.

Dining Philosophers

Results of experiments are summarized in the Table 2 for $N = 4$ and in the Table 3 for $N = 6$. In both cases the results are comparable. Dynamic Strategy again gives the best results for storing probability between 0.5 and 0.1 . Any further decrease in the storing probability below 0.1 results in significant increase of time complexity. In the case of Static Strategy reasonable results were obtained for storing probability 0.75 and further decreasing of probability leads to unreasonable time overhead and thus prevents from higher space savings.

	States	Peak	Saving	Transitions	Overhead
SPIN	191641	191641	0%	1144950	1.00
Dynamic Strategy					
$P_{sto} = 0.50$	160426	165461	14%	2152384	1.88
$P_{sto} = 0.10$	136081	145214	24%	9400300	8.21
$P_{sto} = 0.01$	131306	140758	27%	91533400	79.90
Static Strategy					
$P_{sto} = 0.75$	143661	155920	19%	6702840	5.85
$P_{sto} = 0.65$	124377	143691	25%	116103466	101.40

Table 3. Summary of Experimental Results for **Philosophers** with $N = 6$

Generally, the results for Philosophers are worse than those for Peterson's algorithm and are remarkably influenced by the visitation factor. While in the Peterson's algorithm the average number of state visits in SPIN's computation is $32077/17068 = 1.8$, in Philosophers it is 4.9 ($N = 4$) and 6 ($N = 6$). Experimental observations are thus in accordance with deduced theoretical results.

5 Random Nested DFS

Besides the algorithm with probabilistic reduction strategy we have also explored the potential of randomizing the branching points in nested depth first search. Verification tools typically build the state space from the syntactical description of the problem. E.g. in SPIN the **foreach** *successor t of s* **do** in the depth first search is implemented as **for** $i = 1$ **to** n cycle. This means that the search order is fixed by the input PROMELA program describing the system. If the verification fails due to space limitations it is recommended to re-write the program to re-order the guarded commands in conditionals and loops. However, the user typically has no information on what would be a good re-arrangement. Hence, the situation is very suitable for a randomized approach.

We have implemented the **foreach** *successor t of s* **do** in the depth first search as a *random selection* of the successors ordering and performed a series of comparisons with the standard SPIN tool on similar set of problems as we did before. Even though the method is trivial, the results we obtained were quite

surprising. For instance for the **Philosophers** (with an error) the results are partially summarized in the Table 4.

	SPIN			Random NDFS				
N	States	Trans	Memory	Runs	Success	States	Trans	Memory
11	288922	1449200	56.9 MB	10	10	100421	505150	26.4
12			205.0 MB	10	3	68355	346824	19.9
14			2.8 GB	50	5	46128	250266	16.2
16			38.5 GB	50	5	46288	245406	17.8
20			6.7 TB	50	2	38282	213639	18.2

Table 4. Summary of Experimental Results for **Random Nested DFS**

For the value of the parameter N greater than 11 the SPIN model checker was not able to complete the computation. We therefore give *estimated* values for the memory requirements obtained by extrapolation from finished computations. The randomized algorithm was repeatedly performed (*Runs*) and the number of successful runs (discovering the error before memory overflow) is reported (*Success*). The experiments indicate that even a small number of repetitions can dramatically increase the power of the tool.

We have also considered some artificial verification examples, which demonstrate the potential of the method in some extreme cases. Consider the following verification problem defined by the program

```
1 proc ExIF
2   MainCounter := 0; StepCounter := 0;
3   while StepCounter < 1000 do
4         if
5           true → MainCounter := MainCounter + 1
6           true → MainCounter := MainCounter + 2
7         fi;
8         StepCounter := StepCounter + 1 od
9 end
```

and the LTL formula

$$\Box(MainCounter < 2000 - Diff)$$

The parameter *Diff* determines the ratio of runs of the program that satisfy and violate the formula. More precisely, the probability that $MainCounter = 2000 - Diff$ is

$$Prob(MainCounter = 2000 - Diff) = \binom{1000}{Diff} \left(\frac{1}{2}\right)^{1000}$$

We have performed experiments for various values of *Diff*. The results are summarized in the Table 5. The experiments have confirmed that the actual memory

savings strictly depend on the value of the parameter *Diff*, that is on the probability of a faulty run, and have ranged from 20% up to 90%. We stress that after re-ordering of guarded commands in the *ExIF* program (swapping lines 5 and 6) SPIN finds the counterexample immediately. Re-writing the program helps in this case. The next example shows that in some situations even re-writing the program does not help.

Diff	ViolProb	Algorithm	States	Transitions	%
400	$1.3642.10^{-10}$	RandNestedDFS	37999	55810	10.4%
		SPIN	363202	543502	100.0%
200	$8.2249.10^{-96}$	RandNestedDFS	368766	551537	57.3%
		SPIN	643602	964002	100.0%
100	$6.7017.10^{-162}$	RandNestedDFS	647855	969977	79.6%
		SPIN	813802	1219250	100.0%

Table 5. Summary of Experimental Results for **ExIF**

Let us consider the LTL formula

$$\Box((StepCounter < 1000) \vee (MainCounter \neq 1500)).$$

The formula expresses the property that at the end of every computation (i.e. when $StepCounter = 1000$) the value of *MainCounter* is not 1500. It is easy to see that the *ExIF* program does not fulfil this property. The erroneous computations are those where both guards are selected equally. For every re-ordering of the guards SPIN has to search the significant part of the state space to discover a counterexample. On the other hand, the *RandNestedDFS* algorithm successes very quickly as it selects both guards with the same probability. The same effect has been observed in other tests as well. E.g. in the Leader election problem, for every permutation of all guards SPIN has searched approximately the same number of states while *RandNestedDFS* has needed to search through significantly smaller part of the state space.

6 Conclusions

While verification of probabilistic systems seems to be ready to move to the industrial practice, the use of probabilistic methods in model checking of non-probabilistic systems is at its beginning. The use of probabilistic methods in the explicit state enumeration techniques to reduce the memory required by hashing is an excellent example of the potential of probabilistic methods. Our intention was to investigate other possibilities how randomization could help in model checking.

We have proposed a new probabilistic verification method which could reduce the amount of random access memory necessary to store the information about

the system. The reduction rate depends on the verified system, namely on the average number of state visits. Our experiments have confirmed that the method could achieve a non-trivial reduction within reasonable time overhead.

Another important issue for further study is to examine possibilities of combining our probabilistic reduction strategy algorithm with other techniques to reduce memory usage. We also plan to perform additional experiments to give a more comprehensive view of the performance of our technique and of its scalability.

7 Acknowledgement

We would like to thank Jiří Barnat for introducing us to the mysteries of the SPIN model checker and for his advice and efficient help with incorporating our algorithms into SPIN.

References

1. C. Baier and M. Kwiatkowska. Model Checking for a Probabilistic Branching Time Logic with Fairness. *DISTCOMP: Distributed Computing*, 11, 1998.
2. A. Bianco and L. De Alfaro. Model Checking of Probabilistic and Nondeterministic Systems. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 1026 of *LNCS*, pages 499–513. Springer-Verlag, 1995.
3. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.
4. C. Courcoubetis and M. Yannakakis. The Complexity of Probabilistic Verification. *Journal of the ACM*, 42(4):857–907, July 1995.
5. P. Godefroid, G. J. Holzmann, and D. Pirottin. State-Space Caching Revisited. *Formal Methods in System Design: An International Journal*, 7(3):227–241, November 1995.
6. G. J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design: An International Journal*, 13(3):289–307, Nov. 1998.
7. G.J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In *The SPIN Verification System*, pages 23–32. American Mathematical Society, 1996. Proc. of the Second SPIN Workshop.
8. M. Narasimha, R. Cleaveland, and P. Iyer. Probabilistic Temporal Logics via the Modal Mu-Calculus. In W. Thomas, editor, *Proceedings of the Second International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1578 of *LNCS*, pages 288–305. Springer-Verlag, 1999.
9. U. Stern and D.L. Dill. Combining State Space Caching and Hash Compaction. In B. Straube and J. Schoenherr, editors, *4. GI/ITG/GME Workshop zur Methoden des Entwurfs und der Verifikation Digitaler Systeme*, pages 81–90. Shaker Verlag, Aachen, 1996.
10. R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM journal on computing*, pages 146–160, Januar 1972.

11. M. Vardi. Probabilistic Linear-Time Model Checking: An Overview of the Automata-Theoretic Approach. In J.-P. Katoen, editor, *International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS)*, volume 1601 of *LNCS*, pages 265–276. Springer-Verlag, 1999.
12. W. Visser. Using OBDD Encodings for Space Efficient State Storage during On-the-fly Model Checking. Proceedings of the 1st SPIN Workshop, Montreal, Canada, 1995.
13. A. K. Wisspeintner, F. Huber, and J. Philipps. Model Checking and Random Competition – A Study Using the Model Checking Framework MIC. 10. GI/ITG Fachgespräch "Formale Beschreibungstechniken für verteilte Systeme", pages 91–100, June 2000.
14. P. Wolper and D. Leroy. Reliable Hashing Without Collision Detection. In C. Courcoubetis, editor, *Proc. 5th International Computer Aided Verification Conference (CAV'93)*, volume 697 of *LNCS*, pages 59–70. Springer-Verlag, 1993.