A Probabilistic Approach to Automatic Verification of Concurrent Systems[1]

Enrico Tronci [2,4], Giuseppe Della Penna [2,4], Benedetto Intrigila [2,4], Marisa Venturini Zilli [3,5]

Abstract

The main obstruction to automatic verification of concurrent systems *is the huge amount of memory required to complete the verification task (*state explosion*).*

In this paper we present a probabilistic algorithm for automatic verification *via* model checking*. Our algorithm trades space with time. In particular, when our memory is over because of state explosion our algorithm does not give up verification. Instead it just proceeds at a* lower speed *and its results will only hold with some* arbitrarily small *error probability.*

Our preliminary experimental results show that using our probabilistic algorithm we can typically save more than 30% of RAM with an average time penalty of about 100% w.r.t. a deterministic state space exploration with enough memory to complete the verification task. This is better than having to give up the verification task because of lack of memory.

Key Words Concurrent Systems, Distributed Systems, Reactive Systems, Embedded Systems, Formal Methods, Automatic Verification and Validation, Model Checking, Probabilistic Verification

1 Introduction

The introduction of *Formal Methods* in the design process aims at decreasing time-to-market as well as at improving design quality. In this respect *Model Checking* [2, 3, 17] has been very effective in *Automatic Verification* of *Finite State Systems* (FSSs).

Unfortunately, in general software cannot be conveniently modeled as a FSS. However this is often possible (possibly via *abstraction*) for software defining e.g. *concurrent systems*, *distributed systems*, *reactive systems*, *embedded systems*, *protocols*. In such cases model checking can be a very effective way to detect errors in the earlier phases of the design cycle. Thus meeting formal methods goals of reducing time-to-market and increasing design quality. E.g. see [37, 36, 1, 16, 5, 4, 8, 31, 11, 35].

For concurrent systems the properties to be verified are typically stated as *state invariants* or as *safety properties* (e.g. [1, 4]). Here we focus on such a kind of properties. Checking validity of a property φ for a concurrent system S in our context comes down to *Reachability Analysis* (*State Space Exploration*). That is, to the computation (visit) of the set of all states (*reachable states*) that S can reach starting from its initial state.

For the above reasons state space exploration is at the very heart of all automatic verification tools for concurrent systems (e.g. [9, 7, 17]) . Unfortunately, state space exploration is plagued by the well known *state explosion* problem. That is, a *huge* amount of memory may be required to complete state space exploration. This is indeed one of the main obstructions to automatic verification of concurrent systems.

Two approaches to state space exploration are known: *explicit* state space exploration (e.g. SPIN [9, 24], Murφ [7, 20]) and *symbolic* (i.e. OBDD based [2, 3, 17]) state space exploration (e.g. SMV [23], BSP [32]).

Experimentally it is known [13] that typically explicit state space exploration works better for protocols and software-like systems, whereas symbolic state space exploration works better for hardware like systems. Since here we target protocols and software-like systems, we focus on explicit state space exploration.

When applying model checking to software systems the system to be verified is modeled as the interleaving parellelo (product) of finite state machines (*processes*) (e.g. see [9, 7, 4]) . Thus, from this point of view, a software-like (finite state) concurrent system looks like a (finite state) protocol which, in fact, is typically defined as the interleaving parallelo of finite state machines. For this reason, using a Murφ *parlance*, in the following we use the term *protocol* also for (finite state) *software-like* concurrent system.

In our context, roughly speaking, two kinds of approaches have been studied to counteract (i.e. delay) state explosion: *memory saving* and *auxiliary storage*.

In a memory saving approach, one essentially tries to reduce the amount of memory needed to represent the set of visited states. Examples of the memory saving approach are, e.g., in [6, 38, 14, 15, 27, 28, 10].

In an auxiliary storage approach, one tries to exploit disk storage as well as distributed processors (network storage) to enlarge the available memory (and CPU). Examples of this approach are, e.g., in [25, 26, 21, 30, 22].

In this paper we explore the possibility of trading space with time in order to reduce the amount of memory needed to complete a given verification task.

[1] This research has been partially supported by MURST project TOSCA

[2] Area Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy

[3] Dip. di Scienze dell'Informazione, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy

[4] {tronci,gdellape,intrigil}@univaq.it http://univaq.it/~tronci

[5] zilli@dsi.uniroma1.it

More specifically, our goal here is to devise a *Breadth First* State Space Exploration (BFS) algorithm with graceful degradation. In particular, we want a BFS algorithm *rbfs* that uses only a given fixed amount of memory M. When M is large enough *rbfs* should behave as a standard BFS algorithm (call it *bfs*). When M is too small w.r.t. the problem at hand (i.e. *bfs* runs out of memory) *rbfs* should not give up search, instead it should be able to run with the available memory and *eventually* produce *sensible* results.

We believe that *waiting longer* for a *reasonable answer* is better than being left with an `out of memory` messagge and no answer at all. Therefore we are willing to trade space with computation time to make up for the missing space when M is too small. This is the meaning of the *eventually* above. Moreover we accept a certain error probability in the answer of our algorithm. That is we accept that, hopefully with small probability, our algorithm may miss a reachable state. This is the meaning of "*sensible* results" above.

Note that *probabilistic state space exploration* has already been used in formal verification tools. For example the *hash compaction* technique used in Murφ [27, 28] as well as in SPIN [12] leads to a probabilistic state space exploration. Namely, upon termination there is a small probability (*omission probability*) that a nonvisited reachable state does exist. Hash compaction can considerably reduce the amount of memory needed to carry out the verification task. However it does not trade space with time. In fact, when the memory is over hash compaction abondons the verification effort.

Existence of an algorithm meeting our requirements, at least in a useful sense, is not obvious. If we were not using memory at all, we could try to use results from random walks (e.g. see [19]). However they are too time consuming to be of any use for us. We need an algorithm that can use the *whole* available memory and only at that point starts *trading* space with time (i.e. making random exploration).

Moreover, we need a reasonable tight estimation on the probabilistic behaviour of such an algorithm. Since the transition graph of a concurrent system is a directed graph, we have that a random walk exploring (with high probability) the whole transition graph has an expected running time exponential in the number of reachable states (e.g. [19]). This is indeed the case for the worst case performance.

Fortunately, our experimental results (Section 4) show that concurrent processes do not fall in such a worst case.

This suggests us to look for an algorithm that at run time estimates the probability of having omitted even one state in the state space exploration. For any given ϵ our algorithm will keep running until such an estimated omission probability is less than ϵ.

We base our estimation of the error probability on *run time measures*. Reliability of our estimation rests on statistical properties of protocols that we will establish experimentally.

It is worth noting that our *random walk like* state space exploration algorithm can be easily parallelized. In fact all we have to do is to run our algorithm with the same input (system to be verified) on different machines. Moreover note that our probabilistic algorithm can also be used together with other memory saving techniques (e.g. that in [33]).

Our results can be summarized as follows.

- We present a probabilistic algorithm for Explicit State Space Exploration (Section 3). Essentially our algorithm replaces the hash table used in a BFS algorithm with a cache memory (i.e. no collision detection is done) and instead of the unbounded queue used in a BFS algorithm it uses a fixed size queue. As a result we never run out of memory. However we may forget visited states (because of the cache) and (with arbitrarily small probability) we may fail to visit a reachable state because of our fixed size queue.
  Shortly, our algorithm can be seen as an array of *Random Walks with Memory*. To the best of our knowledge this is the first time that such an approach to state space exploration is presented. Note that our approach differs from that in [18] since we do not make any hypothesis on the structure of the system to be verified.
- We estimate the error probability (i.e. the probability of missing even one reachable state from our visit) at run time. Reliability of our estimation rests on statistical properties of protocols. This is a key feature of our algorithm. Measuring omission probability at run time rather than using an a priori bound avoids us the exponential worst case which exists, but does not seem to occur in practice for software-like systems.
  In other words, our experimental results show that *real world* software-like systems are *well behaved* from a statistical point of view. It is exactly this fact that makes our probabilistic approach viable in the face of the theoretical exponential run time [19] for the worst case scenario.
  We implemented our algorithm within the Murφ verifier and run extensive experiments (Section 4) with all benchmark protocols available with the Murφ distribution.
  Note that our approach applies to the SPIN verifier as well.
- Our algorithm is compatible with all well-known state compression techniques (e.g. as those in [28, 10]). In particular, it is compatible with all state reduction techniques present in the Murφ verifier. We show experiments using our algorithm together with the bit compression [20] and hash compaction [27, 28] features of the Murφ verifier (Section 4).
  Such compatibility is important. In fact, since our algorithm trades space with time, it is not an alternative to other memory saving approaches (e.g. the reduction techniques in the Murφ verifier). On the contrary our probabilistic algorithm should be used *together* with the other available memory saving techniques and only when such techniques used alone cannot complete the verification task at hand because of lack of memory.
- Our preliminary experimental results (Section 4) show that we can typically save about 30% of RAM paying that with about a 100% increase in verification time w.r.t. a deterministic state space exploration with enough memory to complete the verification task. To the best of our knowledge the alternative to our time penalty is just giving up verification beacuse of lack of memory.

Shortly, our probabilistic approach allows verification of systems more than 30% larger than those that can be handled using a deterministic state space exploration.

2 One Step Omission Probability

We define a *concurrent system* $\mathcal{S}$ as a triple (X, I, Next), where X is a (finite) set of states, I is the set of initial states, Next is a function returning the set of successors of a state.

From now on we assume that a concurrent system $\mathcal{S} = (X, I, \text{Next})$ is given once and for all. Thus, e.g., usually we will not show explicitly the dependency of all our functions from $\mathcal{S}$.

We denote with **Reach** (rather than **Reach**(S)) the set of states in X reachable from the states in I. That is **Reach** is the least Z s.t. $I \subseteq Z$ and $(Z \cup \text{Next}(Z)) \subseteq Z$.

Note that **Reach** is *not* known in advance. Indeed **Reach** is exactly what we want to compute. The computation of **Reach** (*State Space Exploration*) can be implemented, e.g., by a *Breadth First* (BF) visit (e.g. as in Murφ [7]) or by a *Depth First* (DF) visit (e.g. as in SPIN [9]).

Our aim is to decide when all reachable states have been visited. We want to take such a decision by looking only at the external behaviour of our search process. There are two reasons for this *observational* approach. First, we do not want to depend too much on implementation details of the state space exploration (e.g. hash table implementation, etc). Second, to estimate omission probabilities starting from implementation details appears to be a daunting task.

We focus on *Explicit Breadth First State Space Exploration*, BFS for short. BFS uses a hash table (to mark visited states) and a queue to hold states whose successors have to be examined. State explosion in a BFS occurs because the hash table eventually contains all reachable states.

We use a (fixed size) cache rather than a hash table and a queue of a given fixed size. Thus our visit will only take up a given fixed amount of memory. We will pay for this with computation time, somehow.

At each step of our search process there will be a set of states Y that may be visited. Note that since we are using a cache, such states need not be new (unvisited). In fact, because of collisions we may *forget* visited states.

Having our queue a fixed size, only a subset Z, of size $1 \leq |Z| \leq |Y|$, of the states in Y can actually be visited. Thus a random selection is done here. We call *search run* the sequence $\xi = [(Y_0, Z_0), \ldots (Y_m, Z_m)]$ of the above mentioned *random trials*. To a run ξ as above we associate the sequence $[(Y_0, |Z_0|), \ldots (Y_m, |Z_m|)]$ that we call *search trace* or the *trace of run* ξ.

Of course to each run corresponds exactly one search trace. However, there can be many runs yielding the same search trace.

In the following we assume that search traces are observable. That is, for each random trial, we can observe the set of states Y and the number k of them that will actually be visited. For our purposes we do not need to be able to observe the outcomes (visited states) of such trials.

Given a search trace Γ and a state x the *one step omission probability* $P_{om1}(\Gamma, x)$ is the probability that x is not visited by a randomly chosen search run ξ having Γ as search trace.

The *domain* $\text{Dom}(\Gamma)$ of a search trace is the set of states that have a chance of being visited from a run for Γ. Let $\Gamma = [(Y_0, k_0), \ldots (Y_m, k_m)]$. Then $\text{Dom}(\Gamma) = \cup_{i=0}^{i=m} Y_i$.

We denote with Γ^x the *projection* of Γ on x. That is, $\Gamma^x = \{(Y, k) \mid (Y, k) \in \Gamma \text{ and } x \in Y\}$.

It can be shown that the one step omission probability can be computed at run time.

Proposition 1. *Let* $x \in \text{Dom}(\Gamma)$. $P_{om1}(\Gamma, x) = \prod_{(Y,k) \in \Gamma^x} (1 - \min\{1, \frac{k}{|Y|}\})$.

Altough we can observe search traces, we will only store information about one step omission probabilities. That is we consider as equivalent traces yielding the same omission probabilities. This motivates the following definition.

Let Γ and Ξ be search traces. We say that Γ *is equivalent to* Ξ (notation $\Gamma \sim \Xi$) iff the following conditions are satisfied:
1. $\text{Dom}(\Gamma) = \text{Dom}(\Xi)$
2. For all $x \in \text{Dom}(\Gamma)$, $P_{om1}(\Xi, x) = P_{om1}(\Gamma, x)$.

To use $P_{om1}(\Gamma, x)$ we need to link it to the probability of omitting x. Let Γ be a search trace and $x \in$ **Reach**. $P_{om}(\Gamma, x)$ denotes the probability that state x is not visited by a search run with a search trace having the same one step omission probabilities as those of Γ. That is, $P_{om}(\Gamma, x) = Pr(x$ is not in run $\xi \mid \xi$ is a run with trace Ξ s.t. $\Xi \sim \Gamma)$.

The probability of omitting even one reachable state (i.e., as in [27, 28], the probability that there exists a nonvisited reachable state) when the observed one step omission probabilities are those of Γ is: $P_{om}(\Gamma) = \max\{P_{om}(\Gamma, x) | x \in \textbf{Reach}\}$.

As in [27, 28], our goal is to bound $P_{om}(\Gamma)$, the probability of missing even one state in our visit. Note that, while $P_{om1}(\Gamma, x)$ can be measured at run time this is not the case neither for $P_{om}(\Gamma, x)$ nor for $P_{om}(\Gamma)$. In the next sections we will show how to estimate $P_{om}(\Gamma)$.

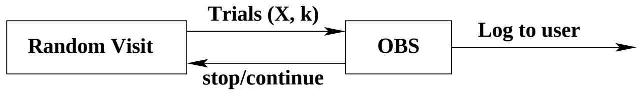

Fig. 1. RBFS and Observer

3 Randomized BFS

In this section we present our *Randomized Breadth First State Space Exploration* algorithm, RBFS for short.

Our idea, shown in Fig. 1, is that of adding to a (randomized) BF visit an observer that by looking at the search trace (Section 2) decides when to stop the search. Moreover, the observer reports to the user the status of the search process.

Figs. 2, 3 show some more details using C-like pseudo-code for, respectively, the RBFS algorithm and the observer. Note that our queue Q, cache T and sample table Y all have a fixed size (chosen by the user). The code in Figs. 2, 3 is to be considered as a conceptual schema. Its actual implementation within the Murφ verifier requires many adjustments and will not be discussed in this paper.

The role of the observer in Fig. 3 is that of estimating $P_{om}(\Gamma)$. To this end our observer tries to estimate $P_{om1}(\Gamma) = \max\{P_{om1}(\Gamma, x) | x \in \mathbf{Reach}\}$.

This is done by computing at run time $P_{om1}(\Gamma, x_1)$, $\ldots P_{om1}(\Gamma, x_k)$ for k states $x_1, \ldots x_k$ (samples) chosen at random in **Reach**. The observer then computes $\hat{P}_{om1}(\Gamma) = \max\ \{P_{om1}(\Gamma, x_i) \mid i = 1, \ldots k\}$. When $\hat{P}_{om1}(\Gamma)$ is less than MAXPOM (Fig. 2) the observer stops the state space exploration.

```
#define W 1009  /* queue size */
#define H 10007  /* cache size */
#define K 103  /* sample size */
#define MAXPOM 0.01 /* threshold for max omission prob */
/* pom1 = one step omission probability */

Q is a queue of size W;
T is an cache of size H; /* storing visited states */
/* Y stores pom1 for K sampled states */
Y is an hash table of size K;
Init_states is the set of initial states;
Next is the system transition function;

unsigned int L; /* Length of breadth first search */
unsigned int size_old = 0; /* num old states in Q */
unsigned int size_new = 0; /* num new states in Q */
double prob_ins = 0.0;

rbfs() {
 while (obs_not_sure_yet())  { /* begin visit */
  L = obs_pick_length();
  size_old = initialize_queue();
  for (level = 1; level <= L && (size_old > 0) ; level++) {
   while (size_old > 0)  {
    pick at random an old state s from Q;
    remove s from Q; size_old--;
    B = array of states s' s.t.
        (s' is in Next(s)) and (s' is not in T);
    empty_slots = (W - (size_old + size_new));
    b = size of B;
    prob_ins = min(1.0, empty_slots/b);
    for (j = 0; j < b; j++)
         obs_update_pom_table(B[j], prob_ins);
    for (j = 0; (size_old + size_new <= W) && (j < b); j++) {
       pick at random a state s' in B;
       remove s' from B;
       if (s' is not in T) {
          insert(T, s');
          insert s' in Q as new state;
          size_new++; } } /* for j */ } /* while size_old */
  /* size_old = 0 */
  relabel all new states in Q as old;
  size_old = size_new; size_new = 0; }  /* for level */
  } /* end visit */ } /* random_bbfs_visit()  */

initialize_queue() {
choose at random at most W states (if any) from the states that
are in Init_states and insert them in Q as old states;
return the number of old states in Q;
/* Q need not be empty.
   It may contain states from previous visit. */ }
```

Fig. 2. Randomized Breadth First Visit

Note that, altough theoretically speaking we could compute $P_{om1}(\Gamma, x)$ for all states in **Reach**, this would vanify our whole approach. In fact this would be equivalent to storing all reachable states that is exactly what we want to avoid. Thus the observer has to do its work using $\hat{P}_{om1}(\Gamma)$.

```
N is the number of bits used to represent
state values;

int obs_pick_length() {
return an int at random between N and 3*N;}

obs_update_pom_table(s, prob_ins) {
if (s is already in table Y)
  write(Y, s, read(Y,s)*(1 - prob_ins));
  return;
/* s new */
if (table Y is not full and level > 2*L/3)
  sample = pick at random a value in {0, 1};
  if (sample) write(Y, s, (1 - prob_ins)); }

obs_not_sure_yet() {
 maxpom1 = max of all pom1 of samples in Y;
 if (maxpom1 < MAXPOM)
      return (0); /* stop */
 else return (1); /* continue */
}
```

Fig. 3. Observer for RBFS

The sampling procedure by itself is not obvious. In fact, during a search run we are required to take k samples at random from the set of states **Reach** which is not known in advance.

Our sampling strategy is the following. During state space exploration we decide, at random, when a visited state is to be included in the set of sampled states (in table Y). We only consider as *candidates* for sampling those visited states that have a nonzero omission probability.

The observer makes the following assumptions.

Hypothesis 1. *When* $\hat{P}_{om1}(\Gamma)$ *is* small*, the following formulas hold.*
A. $P_{om1}(\Gamma) \approx \hat{P}_{om1}(\Gamma)$
B. $P_{om}(\Gamma) \approx P_{om1}(\Gamma)$.
Thus, when $\hat{P}_{om1}(\Gamma)$ *is small,* $P_{om}(\Gamma) \approx \hat{P}_{om1}(\Gamma)$.

According to the hypotheses in Hyp. 1 the observer assumes that $P_{om}(\Gamma) = \hat{P}_{om1}(\Gamma)$. This is how the observer estimates $P_{om}(\Gamma)$.

The validity of the hypotheses in Hyp. 1 rests on statistical properties of protocols as well as on the meaning of *small*. For our observer (3) the meaning of *small* is defined by MAXPOM (Fig. 2). These issues will be discussed in Section 4.

4 Experimental Results

A run of our RBFS algorithm consists of many (randomized) BF visits. As shown in Fig. 1, upon termination and actually after each visit, our RBFS algorithm reports the estimated one step omission probability so far. When such a value is below the threshold MAXPOM (Fig. 2) of the observer (Fig. 3) we stop our algorithm and accept the run as a full visit of the state space. For us, typically, MAXPOM = 0.01. However, as we will see, its value is not too critical.

We can make two kinds of errors.
1. We can accept as good a run that is not a full visit (error of *type 1*, *false positive*).
2. We can refuse a run that actually achieves a full visit

(*error of type 2*, *false negative*).

The experiments presented in this section aim to study the above kinds of errors for our RBFS algorithm as well as the overall effectiveness of our approach.

Note that studying the occurrence frequency of errors of type 1 amounts to validate (or to confute) the assumptions made by the observer (Hyp. 1).

Errors of type 2 are of course linked to the effectiveness of our approach.

We implemented our probabilistic algorithm within the Murφ verifier. This allows us to use as benchmark protocols all those available in the Murφ verifier distribution [20] plus the Kerberos protocol from [29]. This gives us a fairly representative benchmark set.

Space constraints do not allow us to report in full details all of our experiments. More experimental results are available in [34]. All protocols we studied however, have a *statistical behaviour* similar to that of the protocols presented here.

We want to measure how much (RAM) memory we can save by using our probabilistic approach and how reliable are the observer estimates. To make the results from different protocols comparable, we proceed as follows.

First, for each protocol we determine the minimum amount of memory M needed to complete verification using the Murφ verifier (namely Murφ version 3.1 from [20]). Let g (in $[0,1]$) be the fraction of M used for the queue (i.e. g is `gPercentActive` using a Murφ parlance). We say that the pair (M, g) is *suitable* for protocol p iff the verification of p can be completed with memory M and queue gM. For each protocol p we determine the least M s.t. for some g, (M, g) is suitable for p. In the following we denote with $M(p)$ such an M.

Of course $M(p)$ depends on the compression options one uses. Murφ offers *bit compression* (`-b`) and *hash compaction* (`-c`). We are interested in the case in which both bit compression and hash compaction are used (`-b -c`). The results are in Fig. 4.

The meaning of the columns in Fig. 4 is as follows.

Column *Bytes* gives the number of bytes needed to represent a state when bit compression is used. When hash compaction is used hash-compressed states are represented with 40 bits (default).

Column *Reach* gives the number of reachable states for the protocol. For protocol p, in the following we denote such a number with $|\mathrm{Reach}(p)|$.

Column *Rules* gives the number of rules fired during state space exploration. This number depends only on the transition graph and on the visit strategy. For protocol p, in the following we denote such a number with $\mathrm{RulesFired}(p)$.

Column *Max Q* gives the maximum queue size in a BF visit. This number depends only on the transition graph.

Column *Diam* gives the diameter of the transition graph. This number depends only on the transition graph.

Column M gives the minimum amount of memory (in kilobytes) needed to complete state space exploration.

Column g gives the fraction of memory M used for the queue.

Column T gives the time (in seconds) to complete state space exploration when using memory M and queue gM. For protocol p, in the following we denote such a number with $T_{bc}(p)$ (subscript *bc* reminds us that we are using bit compression `-b` and hash compaction `-c`).

Our next step is to run each protocol p with less and less memory using our modified version of the Murφ verifier. That is we run p with memory limits $M(p)$, $0.9M(p)$, ..., $0.1M(p)$.

This approach allows us to easily compare the experimental results obtained from different protocols.

Of course performances also depend on g (the amount of RAM used for the queue) and on the number k of samples taken. We only present results for typical values of such parameters: $g = 0.8$, $k = |\mathrm{Reach}(p)|/100$, $|\mathrm{Reach}(p)|/1000$. Note that when using Murφ typical values for g are around 0.1. However our experiments show that for our randomized approach a value of g around 0.8 typically performs better. Intuitively this means that it is better to explore the state space using many *short memory* random walks rather than just a few *long memory* random walks.

Figs. 5, 6 show our experimental results for the protocols in Fig. 4.

We ran our RFBS algorithm 10 times for each protocol. Figs. 5, 6 report, for each parameter, the average value on such 10 runs. The results obtained on different runs are however very similar. Thus such average values also faithfully represent the values we may expect on a single run. In other words, there is a kind of *ergodicity* which is somehow to be expected.

Figs. 5, 6 all have the same structure. So we just describe the meaning of rows and columns for Fig. 5.

Row *Mem* (with *Mem* = 1, 0.9, ..., 0.1) gives information about the run of protocol p with memory (*Mem**$M(p)$) (p = `sci.m` in Fig. 5).

Row *Pom1s* gives the value of the estimated (using the taken samples) max one step omission probability. That is $\hat{P}_{om1}(\Gamma) = \max\ \{P_{om1}(\Gamma, x_i) \mid i = 1, \ldots, k\}$, where $x_1, \ldots, x_k$ are the samples (if any) taken so far (in the state space exploration) and Γ is the observed search trace (Section 2).

Row *Pom1* gives the value of the exact max one step omission probability. That is $P_{om1}(\Gamma) = \max\ \{P_{om1}(\Gamma, x) \mid x \text{ is one of the states reached so far}\}$.

Row *Pom* gives the value of the omission probability. That is the probability that even one state is omitted in the search. To estimate such a value we run our search process r times. Some of such r runs, say a, will achieve a full visit (i.e. no reachable state has been omitted) and some, say b, will not (i.e. there exists a reachable state that has not been visited). Of course $a + b = r$. We compute *Pom* as follows: $Pom = b/r$. In our case $r = 10$.

Row *Overhead* gives the state overhead. That is *Overhead* = <visited states>/<fresh visited states>. Since we use a cache rather than a hash table we may revisit visited states. Thus *Overhead* can be greater than 1. The state overhead measures how many more states we visit w.r.t. a standard BFS algorithm running with enough memory to complete the verification task.

Protocol	Bytes	Reach	Rules	Max Q	Diam	M	g	T
`kerb.m`	80	109282	172111	20523	19	615	0.190625	301.86
`sci.m`	56	18193	60455	1175	62	94	0.06875	28.17

Fig. 4. Results on a SUN Sparc machine with 512M RAM and SunOS 5.8. Murφ options: `-b` (bit compression) and `-c` (40 bit hash compaction). All experiments have been carried out with option `-ndl` (no deadlock detection). Column M gives memory used (in kilobytes) and column T gives CPU time (in seconds).

Indeed the state overhead also measures the time overhead of our RBFS algorithm w.r.t. a BFS algorithm running with enough memory to complete the verification task. To count fresh visited states as well as other performance parameters we instrumented our randomized Murφ as needed.

Row *Coverage* gives the ratio between the fresh visited states and the number of reachable states (known from Fig. 4). That is, *coverage* = <fresh visited states>/<reachable states>.

Row *Samples* gives the number of samples taken so far. Note that no sample is taken as long as no state has had a chance of being missed. That is only states with one step omission probability greater than 0 are considered for sampling.

Row *Clrate* gives the collision rate so far. That is *Clrate* = <number of collisions so far>/<number of insertions in cache so far>.

Column *Visit i* gives the values of the above mentioned parameters at the end of visit i. A run of our RBFS algorithm may consist of one ore more random BF visits. Thus column *Visit i* reports information on the status of the computation at the end of the i-th visit of the run. The observer stops the exploration when *Pom1s* <= MAXPOM.

If we let our RBFS run *long enough* the omission probability can be made arbitrarily small (we shall not address such completeness issue here). However when *long enough* becomes *too long*, our approach loses its appeal. For this reason the experiments in Figs. 5, 6 are stopped when the state overhead becomes greater than 10 or when the collision rate gets too close to 1. In other words, we do not accept a time penalty greater than 10 (1000%). Of course for a prematurily stopped visit *Pom1s* will be grater than the chosen MAXPOM.

From the results in Figs. 5, 6 we can draw the following conclusions.

The estimated one step omission probability *Pom1s* gives us a reliable upper bound to the omission probability *Pom*. Namely, accepting as full visit a run with *Pom1s* $\leq$ MAXPOM = 0.01 we *never* commit an error of type 1 (false positive). One of the worst cases in this respect is Kerberos with *Mem* = 0.3 (Fig. 6). In fact from Fig. 6 (*Mem* = 0.3, *Visit* = 10) we have that *Pom1s* = 0.064 and *Pom* = 0.250. That is we are very close to accept as full visit (*Pom1s* $\leq$ MAXPOM = 0.01) one with a high omission probability (*Pom* = 0.250) (i.e. an error of type 1, *false positive*).

The number of taken samples is not very critical. A number of samples which is about 1/1000 of the number of reachable states does the job.

The value of MAXPOM is not very critical. E.g. 0.05 would also work fine. Indeed a value of 0.1 would also work fine in all cases but the one in Fig. 6. Making MAXPOM too small is pointless because of time penalty.

We stop our RBFS algorithm when the time penalty (measured by the state overhead *Overhead*) is greater than 10 or when the collision rate *Clrate* gets too close to 1. This typically forces *Pom1s* to be very small (almost 0) or quite large (say > 0.3). In fact, if *Pom1s* grows too large it does not have time to decrease below MAXPOM within our time horizon. This is a source of an (unavoidable) error of type 2 (*false negative*).

E.g. looking at Fig. 5 (samples = 181) we have the following situation.
Row *Mem* = 0.7, *Pom1s* = 0, *Pom* = 0.
Row *Mem* = 0.6, *Pom1s* = 0.7, *Pom* = 0.
Row *Mem* = 0.5, *Pom1s* = 0.6, *Pom* = 1.

The decision of refusing a run with *Mem* = 0.6 is an error of type 2. We could have saved 10% more RAM if we could detect that the runs in row *Mem* = 0.6 are all full visits. Of course making MAXPOM = 0.7 is out of question (because of errors of type 1).

This is a typical situation. That is between the last rightly accepted run (e.g. as *Mem* = 0.7) and the first rightly refused run (e.g. as *Mem* = 0.5) there is typically an *unfairly* refused run (e.g. as *Mem* = 0.6).

We can summarize our results by saying that using memory M, our approach typically allows verification of systems 30% larger than those that can be verified with a deterministic state space exploration using memory M. Our typical time penalty is about 100%. That is we take twice as much time as a deterministic state space exploration with enough memory to complete the verification task.

5 Conclusions

We presented a probabilistic approach to automatic verification. Using our approach we are able to trade space with time. In particular when our memory is over we do not have to stop our verification, it just proceeds at a *lower speed* and its results will only hold with some *arbitrarily small* error probability.

Our experimental results show that typically our probabilistic algorithm allows automatic verification of systems more than 30% larger than those that can be handled using a deterministic state space exploration. Our typical time penalty is about 100%.

Many improvements are possible to the proposed approach. For example: improve efficiency at the program level, find a tighter bound on the omission probability, parallelization.

A probabilistic version of a symbolic search also appears to be an interesting topic for further investigation.

		Samples = 181					samples = 17		
Mem	Visit	1	2	3	4	5	1	2	3
1.0	Pom1s	0					0		
	Pom1	0					0		
	Pom	0					0		
	Overhead	1.066					1.067		
	Coverage	1					1		
	Samples	0					0		
	Clrate	0.394					0.395		
0.9	Pom1s	0					0		
	Pom1	0					0		
	Pom	0					0		
	Overhead	1.134					1.139		
	Coverage	1					1		
	Samples	0					0		
	Clrate	0.491					0.493		
0.8	Pom1s	0					0		
	Pom1	0					0		
	Pom	0					0		
	Overhead	1.313					1.291		
	Coverage	1					1		
	Samples	0					0		
	Clrate	0.607					0.601		
0.7	Pom1s	0					0		
	Pom1	0					0		
	Pom	0					0		
	Overhead	1.900					1.952		
	Coverage	1					1		
	Samples	0					0		
	Clrate	0.764					0.770		
0.6	Pom1s	0.483	0.700				0.450	0.648	
	Pom1	0.483	0.747				0.450	0.742	
	Pom	0.100	0				0	0	
	Overhead	5.859	10.178				5.752	10.151	
	Coverage	1	1				1	1	
	Samples	177.300	181				16.900	17	
	Clrate	0.933	0.962				0.933	0.962	
0.5	Pom1s	0.467	0.675	0.611			0.500	0.633	
	Pom1	0.467	0.760	0.757			0.500	0.755	
	Pom	1	1	1			1	1	
	Overhead	4.113	9.947	10.218			3.974	10.211	
	Coverage	0.980	0.999	0.999			0.979	0.999	
	Samples	180.600	181	181			17	17	
	Clrate	0.919	0.968	0.968			0.917	0.968	
0.4	Pom1s	0.533	0.700	0.608			0.550	0.583	
	Pom1	0.533	0.775	0.775			0.550	0.780	
	Pom	1	1	1			1	1	
	Overhead	3.165	7.804	10.166			2.934	10.211	
	Coverage	0.867	0.975	0.986			0.850	0.984	
	Samples	181	181	181			17	17	
	Clrate	0.907	0.967	0.975			0.898	0.975	
0.3	Pom1s	0.450	0.692	0.675	0.667		0.467	0.567	
	Pom1	0.450	0.770	0.785	0.787		0.467	0.760	
	Pom	1	1	1	1		1	1	
	Overhead	2.677	6.523	9.989	10.136		2.658	10.121	
	Coverage	0.753	0.915	0.948	0.953		0.751	0.948	
	Samples	181	181	181	181		17	17	
	Clrate	0.905	0.968	0.980	0.980		0.904	0.980	
0.2	Pom1s	0.500	0.733	0.717	0.717		0.500	0.608	0.497
	Pom1	0.500	0.765	0.775	0.760		0.500	0.775	0.780
	Pom	1	1	1	1		1	1	1
	Overhead	2.173	5.228	7.953	10.099		2.182	8.231	10.068
	Coverage	0.575	0.783	0.826	0.844		0.574	0.835	0.849
	Samples	181	181	181	181		17	17	17
	Clrate	0.901	0.970	0.981	0.985		0.901	0.982	0.986
0.1	Pom1s	0.500	0.708	0.708	0.708	0.708	0.483	0.597	0.528
	Pom1	0.500	0.770	0.770	0.775	0.775	0.483	0.765	0.775
	Pom	1	1	1	1	1	1	1	1
	Overhead	2.306	4.718	6.982	9.049	10.064	2.171	7.090	10.060
	Coverage	0.336	0.509	0.543	0.563	0.571	0.317	0.549	0.574
	Samples	181	181	181	181	181	17	17	17
	Clrate	0.920	0.974	0.984	0.988	0.989	0.910	0.984	0.989

Fig. 5. Random BF Visit with option `-bc`. Protocol: `sci` (g = 0.8, M = 94, k = 181 / 17).

		Samples = 1091									
Mem	Visit	1	2	3	4	5	6	7	8	9	10
1.0	Pom1s	0									
	Pom1	0									
	Pom	0									
	Overhead	1.008									
	Coverage	1									
	Samples	0									
	Clrate	0.303									
0.9	Pom1s	0									
	Pom1	0									
	Pom	0									
	Overhead	1.013									
	Coverage	1									
	Samples	0									
	Clrate	0.376									
0.8	Pom1s	0									
	Pom1	0									
	Pom	0									
	Overhead	1.021									
	Coverage	1									
	Samples	0									
	Clrate	0.450									
0.7	Pom1s	0									
	Pom1	0									
	Pom	0									
	Overhead	1.033									
	Coverage	1									
	Samples	0									
	Clrate	0.524									
0.6	Pom1s	0									
	Pom1	0									
	Pom	0									
	Overhead	1.053									
	Coverage	1									
	Samples	0									
	Clrate	0.599									
0.5	Pom1s	0									
	Pom1	0									
	Pom	0									
	Overhead	1.084									
	Coverage	1									
	Samples	0									
	Clrate	0.676									
0.4	Pom1s	0.326	0.951	0.925	0.938	0.922	0.841	0.841			
	Pom1	0.326	0.951	0.925	0.938	0.922	0.841	0.841			
	Pom	1	0.750	0.667	0	0	0	0			
	Overhead	1.142	1.708	2.024	2.514	2.840	3.403	3.403			
	Coverage	1	1	1	1	1	1	1			
	Samples	582	582	530	685.500	685.500	619	619			
	Clrate	0.754	0.827	0.861	0.887	0.897	0.917	0.917			
0.3	Pom1s	0.230	0.975	0.917	0.774	0.560	0.375	0.221	0.221	0.064	0.064
	Pom1	0.230	0.979	0.978	0.976	0.967	0.966	0.969	0.969	0.955	0.955
	Pom	1	1	1	1	1	0.750	0.750	0.250	0.250	0.250
	Overhead	1.246	2.394	3.605	4.273	6.084	7.270	7.699	9.635	9.638	9.713
	Coverage	0.987	0.998	0.999	0.999	1	1	1	1	1	1
	Samples	1091	1091	1091	1091	1091	1091	1091	1091	1091	1091
	Clrate	0.829	0.912	0.942	0.951	0.965	0.971	0.973	0.978	0.978	0.978
0.2	Pom1s	0.394	0.976	0.934	0.813	0.813	0.444	0.343	0.277		
	Pom1	0.394	0.979	0.979	0.978	0.978	0.966	0.960	0.963		
	Pom	1	1	1	1	1	1	1	1		
	Overhead	1.404	2.723	4.035	4.182	4.809	7.881	8.417	10.026		
	Coverage	0.939	0.975	0.987	0.987	0.989	0.997	0.997	0.998		
	Samples	1091	1091	1091	1091	1091	1091	1091	1091		
	Clrate	0.893	0.947	0.965	0.966	0.969	0.982	0.983	0.986		
0.1	Pom1s	0.373	0.977	0.920	0.845	0.845	0.843	0.598	0.492	0.492	0.396
	Pom1	0.373	0.979	0.979	0.979	0.979	0.977	0.979	0.979	0.979	0.979
	Pom	1	1	1	1	1	1	1	1	1	1
	Overhead	1.512	2.652	3.755	3.780	4.043	5.943	6.742	7.154	9.022	10.040
	Coverage	0.732	0.837	0.887	0.887	0.895	0.938	0.940	0.949	0.969	0.973
	Samples	1091	1091	1091	1091	1091	1091	1091	1091	1091	1091
	Clrate	0.937	0.969	0.979	0.979	0.980	0.987	0.989	0.990	0.992	0.993

Fig. 6. Random BF Visit with option `-bc`. Protocol: `kerb` (g = 0.8, M = 615, k = 1091).

References

1. J.M. Atlee and J.D. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. on Software Engineering*, 19(1):24–40, January 1993.
2. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8), Aug 1986.
3. 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.
4. W. Chan, R. J. Anderson, S.Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Trans. on Software Engineering*, 24(7):498–529, July 1998.
5. E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.
6. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, (1):275–288, 1992.
7. 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.
8. C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Software Engineering*, 24(11), Nov. 1998.
9. G. J. Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
10. G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in Systems Design*, 1998.
11. Gerard Holzmann and Margaret Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, April–June 2000.
12. G.J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In R. Langerak, editor, *Proc. Third SPIN WorkShop*, Twente Univ., The Netherlands, April 1997.
13. A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitily conjoined bdds. In *31st IEEE Design Automation Conference*, pages 276–282, 1994.
14. C. N. Ip and D. L. Dill. Better verification through symmetry. In *11th International Conference on: Computer Hardware Description Languages and their Applications*, pages 97–111, 1993.
15. C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, 1993.
16. D. Jackson. Elements of style: Analyzing a software design feature. *IEEE Trans. on Software Engineering*, 22(7):484–495, July 1996.
17. K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.
18. Milena Mihail and Christos H. Papadimitriou. On the random walk method for protocol testing. In Springer LNCS, editor, *Proc. of: Computer Aided Verification (CAV)*, pages 132–141, 1994.
19. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
20. `http://sprout.stanford.edu/dill/murphi.html`.
21. R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. In *IEEE International Conference on Computer Design*, pages 358–364, 1996.
22. J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance bdd package by exploiting memory hierarchy. In *33rd IEEE Design Automation Conference*, 1996.
23. `http://www.cs.cmu.edu/~modelcheck/`.
24. `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.
25. U. Stern and D. Dill. Parallelizing the murφ verifier. In *Proc. 9th Int. Conference on: Computer Aided Verification*, volume 1254, pages 256–267, Haifa, Israel, 1997. LNCS, Springer.
26. U. Stern and D. Dill. Using magnetic disk instead of main memory in the murφ verifier. In *Proc. 10th Int. Conference on: Computer Aided Verification*, volume 1427, pages 172–183, Vancouver, BC, Canada, 1998. LNCS, Springer.
27. 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.
28. 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.
29. `http://verify.stanford.edu/uli/research.html`.
30. T. Stornetta and F. Brewer. Implementation of an efficient parallel bdd package. In *33rd IEEE Design Automation Conference*, pages 641–644, 1996.
31. P. Morel T. Jèron. Test generation derived from model–checking. In *Proc. of: Intern. Conf. on Computer Aided Verification (CAV'99)*. LNCS 1633, Springer, July 1999.
32. E. Tronci. Hardware verification, boolean logic programming, boolean functional programming. In *Proceedings of IEEE Conference: Logic in Computer Science*, San Diego, CA, USA, 1995. IEEE Computer Society Press.
33. 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.
34. E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. A probabilistic approach to automatic verification of concurrent systems. Technical report, Università di L'Aquila, Area informatica, September 2001.
35. G. Brat W. Visser, K. Havelund and S. Park. Model checking programs. In *Proc. of IEEE International Conference on: Automated Software Engineering (ASE)*. IEEE Computer Society Press, Sept. 2000.
36. F. Wang, P.-A. Hsiung, and Y.-S. Kuo. Verification of concurrent client-server real-time scheduling systems. In *Proc. of: Real-Time Computing Systems and Applications (RTCSA) Workshop*. IEEE CS Press, Dec 1999.
37. F. Wang and C.-T. Lo. Procedure-level verification of real-time concurrent systems. *Journal of Real-Time Systems*, 16(1):81–114, Jan. 1999.
38. Pierre Wolper and Dennis Leroy. Reliable hashing without collision detection. In *Proc. 5th Int. Conference on: Computer Aided Verification*, pages 59–70, Elounda, Greece, 1993.