Resource-Aware Verification using Randomized Exploration of Large State Spaces

Nazha Abed[1], Stavros Tripakis[2], and Jean-Marc Vincent[1] *

[1] LIG, 51, avenue Jean Kuntzmann, 38330 Montbonnot Saint-Martin, France
[2] Cadence Laboratories, 2150 Shattuck Avenue, Berkeley, CA 94704

Abstract. Exhaustive verification often suffers from the state-explosion problem, where the reachable state space is too large to fit in main memory. For this reason, and because of disk swapping, once the main memory is full very little progress is made, and the process is not scalable. To alleviate this, partial verification methods have been proposed, some based on randomized exploration, mostly in the form of random walks. In this paper, we enhance partial, randomized state-space exploration methods with the concept of *resource-awareness*: the exploration algorithm is made aware of the limits on resources, in particular memory and time. We present a memory-aware algorithm that by design never stores more states than those that fit in main memory. We also propose criteria to compare this algorithm with similar other algorithms. We study properties of such algorithms both theoretically on simple classes of state spaces and experimentally on some preliminary case studies.

1 Introduction

To verify system correctness, one can proceed by exhaustive verification (e.g. model checking) or testing. Model checking [1,2,3] has gained wide acceptance within the hardware and protocol verification communities, and is witnessing increasing application in the domain of software verification. When the state space of the system under investigation is finite, model checking may proceed in a fully automatic, push-button fashion. Moreover, should the system fail to satisfy the formula, a counter example trace to the error state is produced. Model checking however is not without its drawbacks, the most prominent of which is state explosion: the phenomenon where the size of a system's state space grows exponentially in the size of its specification. State explosion can render the model-checking problem intractable for many applications of practical interest.

Testing, on the other hand, is typically performed directly on the implemented system. This has the advantage of checking the "real" system instead of a model of it. The disadvantage is that anomalies are detected often too late, resulting in high costs to correct them. Testing is inherently incomplete, as there is no guarantee of covering the state space even after several experiments.

* This work is partially supported by the ANR SETIN Check-Bound and the Region Rhône-Alpes, France.

Researchers have developed a plethora of techniques aimed at curtailing state explosion, by reducing the amount of memory necessary for states storage or reducing the state space to explore. Examples of the approaches made to reach the first goal are hash compaction [25] and bi-state hashing [22] which consists of encoding the graph states by the memory bits via a hash function. The methods that aim to reduce the state space include partial-order reduction methods [26]; which are based on the observation that executing two independent events in either order results in the same global state and symmetry reduction [27]; which uses the existence of nontrivial permutation group that preserves the state transition graph. There is also *symbolic* model checking techniques that operate on sets of states rather than individual states, and represent such sets symbolically, for instance, using binary decision diagrams (BDDs) [6]. In this paper we focus on explicit enumerative state space exploration methods.

Other techniques aim at a partial, i.e., incomplete, exploration of the state space, in particular, using randomized algorithms, which make decisions based on outcomes of random experiments such as tossing a fair coin or generating a random number. Randomized algorithms are extensively used, basically for two reasons: simplicity and speed [4]. A consequence of using randomization is that correctness or termination can often be asserted only with some controlled probability.

The randomized algorithms proposed in the literature are mostly based on random walks. A random walk on a graph starts from the initial state, and at each step, chooses with uniform probability a successor of the current state and visits it. This choice is independent from the traversal history, which is characteristic of a Markov chain. When the random walk encounters a deadlock point, it restarts from the initial state. The algorithm terminates when a target state is reached or when the expected number of the visited states reaches a certain limit. This method stores only an actual state and does not keep any information about previously visited states, thus it has very little memory requirements.

This simple form of random walk was applied first to model-checking by West [8] and more recently in [9,12,10]. Because it is completely memory free, the random walk method cannot distinguish between visited and not visited states, and so it may spend a large amount of time repeatedly visiting the same states: we call this *redundancy*. Because of this, covering the entire graph (or a high portion of it) may need a prohibitively large amount of time. Also, the frequency (probability) of visits may be very variable from one state to another (some states are more frequently visited than others). This frequency depends on the graph structure as well as the algorithm behavior.

Several methods have been proposed to avoid these drawbacks. Some of these methods try to force exploration direction, like the re-initialization methods that restart the random walk process periodically to avoid blocking in a small closed components for a long time. The re-initialization can be made from a random state of the previous walk and not necessary from the initial state. This has the advantage to minimize redundancy and reach deep states [11]. The local exhaustive search combined to random walk [14] explores better some regions of interest (dense regions for example) which cannot be well explored with only simple random walk. This may be the case for example if one knows that it is near a target node. Guided search decides of the next exploration direction based on general information about the graph and system semantics. In [15], the authors use a metric to estimate reachability probability of a target node. To gain in memory and time, the parallelization method of random walk seems to be very useful and efficient. It explores more states [14] and reduces significantly the error probability [12].

Other methods use some additional memory to keep a subset of the visited states. These states are used to report the counter example trace as done in tracing methods or to limit revisits of same nodes and improve the coverage as done in caching methods [16] [17]. Caching is an exploration algorithm that focuses on the strategy of nodes storing and deletion from the cache. The exploration scheme can be deterministic, as in a breadth-first or depth-first search (BFS, DFS), or random. In [20], the proposed algorithm uses BFS with a randomized partial storage. When the memory is full, the algorithm proceeds at a lower speed but does not give up. As reported in [20], this algorithm can save 30% of the memory with an average time penalty of 100%. Other methods that use randomization in a verification context include [**?**,**?**,**?**,**?**,**?**,**?**]. **COMPLETE**

All the methods mentioned above that are based on random walk improve the redundancy of exploration but the cover time can still be very large. In this paper, we propose methods that aim to further improve exploration by avoiding redundancy and reducing the cover time. First, we propose a generic scheme that aims to encompass special instances of algorithms. Then, we propose the Uniform Random Search (URS) algorithm, which is based on a different selection function than random walk (RW). While RW is a depth-oriented algorithm, URS can go in depth, in breadth or in a uniform fashion. We can also control the rate of depth or breadth exploration by tuning a mixing parameter.

A major novelty of our exploration scheme lies in the fact that it explicitly uses a parameter N that represents the maximum number of states that can be stored in main memory at any given time. Thus, our algorithms are *resource-aware*, and in particular, *memory-aware*. Main memory is the main bottleneck in exhaustive verification, for reasons we explain below. Our scheme tries to overcome this by explicitly taking into account the resource constraints and using them to make decisions during the exploration.

The randomized algorithms proposed are sound, which means that if a bug is found then the model is indeed incorrect. As in [12,10], they are probabilistically complete, in the sense that if after several iterations no bugs are found, then the system is correct with some probability which depends on the number of iterations and visited states.

WE NEED TO COMMENT WHAT'S THE DIFFERENCE WITH YOUR CONCUR PAPER

The rest of the paper is organized as follows: The proposed scheme and algorithm are detailed in section 2. Section 3 gives some general theoretical results that are projected on two cases of regular graphs. Experimental results are summarized in section 4, while section 5 contains our conclusion.

2 Context and Algorithms

We model a system as a directed transition graph $G(M, v_0, Succ)$, where, M is a finite set of nodes representing the system states, v_0 is the initial node ($v_0 \in M$) and *Succ* is the transition function: it takes as input a node v and returns as output the set of all successors of v. We do not dispose of the entire transition graph. We can, however, construct and explore it gradually by means of the initial state and the transition function *Succ*. We assume that the available main memory can store at most N states. N can be computed by dividing the size of the memory, by the size of the memory representation of each state. To generate randomized algorithms, a pseudo-random numbers generator is given. The generated numbers can be considered as uniformly distributed in $[0, 1]$, based on which, other distribution laws can be generated if necessary.

To verify a given safety property stated as an invariant ϕ, the simplest method is to explore the graph G and verify ϕ for each state $s \in G$. If we use an exhaustive deterministic exploration, the computer's memory will be rapidly filled by the N first reachable states (N depends on the available memory as said above). Then, the computer will typically spend most of its time *swapping* memory to/from disk with very few additional states explored. This is clearly non-scalable: running the model-checker for several days may result in only a few additional visited states than running it only a few hours. Instead, we choose a memory-aware, randomized, partial exploration, and repeat it several times with different paths (consequence of randomization) to cover as many reachable states as possible.

One wishes, naturally, that the randomized algorithm explores the state space efficiently, i.e., quickly and using reasonable memory resources. Since the memory size is given and finite, a good exploration is defined mainly according to the time it takes: one can hope to cover with a randomized algorithm a considerable percentage of the reachable graph in less time than with the exhaustive algorithm which will be quickly blocked because of the swapping.

2.1 A generic randomized exploration scheme

A random exploration algorithm can be cast into the generic scheme shown in Figure 1. P represents the algorithm parameters, for example the memory size N, the number of initial parallel runs in the case of a parallel random walk [14], ect. This last parameter, among others, can be modified during the algorithm execution according to the available resources and exploration needs. The set I contains global information on the graph structure, for instance, mean number of successors per node, mean number of loops, strongly connected components, etc. Note that this type of information can be collected on the fly and used to guide and optimize the exploration [15].

```
$V$ : set of stored nodes;
$P$ : algorithm parameters;
$I$ : global information;
$v$ : node;

$V \leftarrow V_0$; //Set of initial nodes
$P \leftarrow Par$; //Algorithm parameters
$I \leftarrow I_0$; //Initial global information

While (not stop condition) do
    $v \leftarrow \text{select}(V, P, I)$;
    check($v$); //verify if the property holds
    $(V, I) \leftarrow \text{update}(V, v, P, I)$;
done
```

Fig. 1. The general randomized exploration scheme

A specific algorithm that fits the above scheme is defined by specifying the *stop condition* and the two functions *select* and *update*. With these three parameters, one can define many variants of the general algorithm, including many found in the literature. The *stop condition* can be, for example, the presence of a deadlock, exhaustion of the expected number of steps or simply reaching a target state. Some algorithms in the literature emphasize state storage and deletion strategies (FIFO, LFU, LRU, random ...), like the caching techniques [17] [16], so they focus in optimizing the *update* function. The *update* function modifies the sets V and I in order to optimize the consumed resources and make the evolution of the exploration effective. As mentioned in the introduction, our interest is mainly the exploration strategy itself, that is the *select* function. The *select* function chooses at each step the next node v, to be visited from the set of successors of V; the already visited states still in memory. This choice can be guided by the information in I.

In this scheme, the random walk algorithm has as *stop condition* the reachability of a deadlock point or the reach of a target node according to the algorithm goal. The *select* function is a uniform random choice between the successors of the current node (the single stored in V), when the *update* function consists on simply replacing the current node by the one lastly chosen. In presence of a deadlock, the current node takes the value of the initial state and so on.

As we are interested in the exploration strategy, we propose a Uniform Random Search (URS) algorithm based on a new *select* function: see Figure 2. We have a set V of already visited states. V is of size N: that is, the algorithm ensures that there are never more than N states in V. Initially this set contains the initial state v_0. At each step i, the URS algorithm picks uniformly one visited state u from V, and then uniformly chooses one successor v of u. Note that this does not imply a uniform choice from all the visited node successors. If v is not already visited then it is checked with respect to the safety property and added to the set of visited states. The algorithm stops, and eventually restarts, when the memory is full ($j = N$) or when the expected number of steps, n, is reached.

```
Uniform Random Search – URS

$V$ : set of stored nodes;
$N$ : Maximum size of $V$;
$n$ : Maximum number of steps;
$v, u$ : nodes;
$i, j$ : integers;

$V \leftarrow \{v_0\}$;
$i \leftarrow 0$;
$j \leftarrow 0$;

While $((j \leq N)$ and $(i \leq n))$ do
    $u \leftarrow$ pick uniformly a node in $V$;
    If $(Succ(u) \neq \emptyset)$ then
        $v \leftarrow$ pick uniformly a node in $Succ(u)$;
        If $(v \notin V)$ then
            check$(v)$;
            $V \leftarrow V \cup \{v\}$;
            $j \leftarrow j + 1$;
        end If
    end If
    $i \leftarrow i + 1$;
done
```

```
Simplified, Memory-Aware Deep Random Search – SDRS

$V$ : set of stored nodes;
$N$ : Maximum size of $V$;
$n$ : Maximum number of steps;
$v$ : current node;
$i, j$ : integers;

$V \leftarrow \{v_0\}$;
$v \leftarrow v_0$;
$i \leftarrow 0$;
$j \leftarrow 0$;

While $((j \leq N)$ and $(i \leq n))$ do
    If $(Succ(v) = \emptyset)$ then
        $v \leftarrow$ pick uniformly a node in $V$;
    else
        $v \leftarrow$ pick uniformly a node in $Succ(v)$;
        If $(v \notin V)$ then
            check$(v)$;
            $V \leftarrow V \cup \{v\}$;
            $j \leftarrow j + 1$;
        end If
    end If
    $i \leftarrow i + 1$;
done
```

Fig. 2. The URS and SDRS algorithms

[11] presents an extended random-walk based algorithm called Deep Random Search (DRS). The *stop condition* of DRS does not consider the limited memory

size and supposes that the set of "non-closed" nodes (i.e., that have at least one non-visited successor) fits in main memory. In this paper we use a simplified, but memory-aware, version of DRS, that we call SDRS. Like URS, SDRS can keep at most N states in memory at any given time. This puts the two algorithms in the same framework and allows comparisons. SDRS has the same stop condition as URS. The *select* and *update* functions of SDRS are the same as for simple random walk except that the current node is reset to a node chosen randomly in V and not to the initial node.

When the main memory is full, the algorithms are stopped, the memory is emptied and the algorithms are restarted. This can be repeated several times. The re-initialization can be done from the initial state or from another randomly chosen state from the set V of states visited during the last exploration. Note that the initialization from the initial state often does not result in a very high degree of redundancy because the number of states in each repetition is very large and can usually match the graph's diameter. In the rest of the paper, we will consider two situations in our analysis and experimental results. In one situation we suppose that the main memory is large enough to contain the entire state space of the graph under exploration. In this case, we will speak of the versions of the algorithms URS and SDRS where these do not have to be reinitialized. In the second, more realistic case for industrial-size examples, the main memory cannot store the entire state space, and the algorithms are run multiple times, after re-initialization as described above. In this case, we will denote the algorithms by RURS and RSDRS to emphasize the fact that they are re-initialized.

2.2 Evaluation criteria

URS and SDRS are only two of the many possible memory-aware, randomized exploration algorithms one can think of. The question is, which algorithm is better, in which cases, and what exactly does "better" mean? To answer these questions, we need some criteria to evaluate performance of such algorithms. We define such criteria in two ways: stochastic and experimental.

One useful criterion is *mean cover time*. The cover time is the number of steps needed by a given algorithm which starts at the initial state to *cover* (i.e., visit) some percentage of the set of reachable states.[3] The mean cover time gives a good indication on the capacity of the algorithm to reach states and explore most of the graph. Cover time also reflects what can be termed *response time*, with an error ϵ. For example, if one needs a response about the system correctness with probability of error $\epsilon = 0.05$, the necessary time for giving this response can be defined as the cover time of 95% of the graph. Some exploration algorithms will provide this answer in less time than others.

When the set of reachable states is unknown, we compare the number of *covered* nodes (i.e., visited nodes). As the number of the visited nodes increases,

[3] For the random walk, in the case of undirected graphs, the mean cover time of any graph is polynomial [24]. In the case of directed graphs it is in general exponential, except for some restricted classes [12]. These classes are so restricted that they are not very interesting for model checking.

the probability that a node already visited is revisited typically increases (redundancy). It results from this, that the number of newly visited nodes decreases according to the execution time T_e. From this fact, the coverage progression is, typically, a logarithmic curve according to T_e. This is confirmed by our theoretical and experimental results.

Another useful criterion is the *minimum reachability probability* over all reachable nodes. Reachability probability is the probability that a given node v of a graph G is visited by a given algorithm A, denoted denoted $\mathbb{P}_{G,A}(v)$. Note that the model checking problem can be often seen as searching for a target (e.g., error) state. The reachability probability of a target state is thus meaningful. Due to the fact that the considered exploration algorithms are random, the list of visited nodes V is a random variable that depends on the algorithm and the particular graph structure. Thus, the membership of a given node v to V is a random variable of which the probability $\mathbb{P}_{G,A}(v)$ for a given graph G and a given algorithm A differs from a node to another. The minimum reachability probability criterion is the minimum over all nodes of these probabilities:

$$\pi_{min}(G, A) = \min_{v} \mathbb{P}_{G,A}(v)$$

In general, re-iterating the randomized algorithm improves the probability of reaching states and finding errors.

Note that reachability probability depends on the resources that are available to an algorithm A, for instance, the available memory and time. In the case of URS and SDRS, for example, it depends on parameters N and n. Thus, another useful criterion is the *mean number of covered nodes*, for given resource parameters.

In practice, there are several types of graphs, and an algorithm performs differently depending on the form of the explored one. To compute precise analytic results, we have analyzed regular classes of graphs: trees and grids. Regular graphs are suitable to study analytically the behavior of exploration algorithms for several reasons:

- Although the model checking graphs are not regular, they contain frequently regular components [7].
- One can manipulate regular graphs to compute probabilistic measures analytically, which is practically impossible for graphs of irregular topology.
- By tuning the two parameters of a regular tree (depth and degree), we can get large or deep graphs and define a density factor suitable to our study.
- Trees and grids constitute two extreme cases of general graphs. In trees, there are no intersections between the successors, and in grids, there is intersection between all successors. Other graphs can be considered as an intermediate case between this two ones.

3 Theoretical results

This section aims at a theoretical comparison of randomized exploration algorithms in terms of various statistics. More precisely, we investigate exact computations of the mean cover time, the mean number of covered nodes and other related criteria such as reachability probabilities, for URS and SDRS. We do this for two simple types of graphs: trees and grids. We first provide some general results that apply to any graph.

For URS, the ordered sequence $V_n = (v_1, ..., v_n)$ of visited nodes in n steps can be represented as a sequence $\underbrace{w_1, \overbrace{...}^{\alpha_1}, w_2, \overbrace{.....}^{\alpha_2}, w_3, \overbrace{.....}^{\alpha_3}, ..., w_{k-1}, \overbrace{.....}^{\alpha_{k-1}}, w_k}_{W_n=(w_1,...,w_k)}, \overbrace{...}^{\alpha_k}$ where each w_i corresponds to a novel visited node followed by α_i redundant visits, that is the considered sequence V_n is constituted by $n - k$ repeated nodes interlaced in an ordered set of k distinct nodes $\underline{w}_k = (w_1, ..., w_k)$. Let $\underline{w}_{k-1} = (w_1, ..., w_{k-1})$ and denote by $F(w_i)$ (resp. $C(w_i)$) the set of fathers (resp. children) of the node w_i, $i = 1, .., k$.

Lemma 1. *The probability $\mathbb{P}(\underline{w}_k, n)$ to cover node $\underline{w}_k$ in n steps by URS is:*

$$\mathbb{P}(\underline{w}_k, n) = \alpha(\underline{w}_k)\mathbb{P}(\underline{w}_k, n-1) + \beta(\underline{w}_k)\mathbb{P}(\underline{w}_{k-1}, n-1)$$

$$\alpha(\underline{w}_k) = \frac{1}{k}\sum_{i=1}^{k} \frac{|C(w_i) \cap \underline{w}_k|}{|C(w_i)|}, \quad \beta(\underline{w}_k) = \frac{1}{k-1} \sum_{v \in F(w_k) \cap \underline{w}_{k-1}} \frac{1}{|C(v)|}$$

Note that $\alpha(\underline{w}_k)$ is a redundancy factor, equal to the probability to revisit a node at step n (no node is newly covered), while $\beta(\underline{w}_k)$ is an innovation factor expressing the probability to cover at step n a new node, which must be w_k, since the set $\underline{w}_k$ is stored in order of visits.

The elementary recursion for $SDRS$ is a bit more complicated than for URS and one must distinguish closed and open points of exploration. The exploration is said to be in a closed point at step n, if it has reached a deadlock at step $n-1$, it attempted, unsuccessfully, in step n to choose a successor from this deadlock and so it will be reinitialized in step $n+1$ from a uniformly randomly chosen state of V_n. An open point is a point of the walk which is not a closed point.

Lemma 2. *Let $\mathbb{P}(\underline{w}_k, n, C)$ (resp. $\mathbb{P}(\underline{w}_k, n, O, v)$ be the probability to cover in n steps the set of nodes $\underline{w}_k$ and to be, by step n, in a closed point (resp. in an open point at node v). Then:*

$$\mathbb{P}(\underline{w}_k, n, C) = \frac{|D(\underline{w}_k)|}{k}\mathbb{P}(\underline{w}_k, n-1, C) + \sum_{v \in D(\underline{w}_k)} \mathbb{P}(\underline{w}_k, n-1, O, v)$$

$$\begin{aligned}\mathbb{P}(\underline{w}_k, n, O, v) = \sum_{u \in F(v) \cap \underline{w}_k} \Big[& \frac{\mathbb{P}(\underline{w}_k, n-1, O, u)}{|C(u)|} + \frac{\mathbb{P}(\underline{w}_k, n-1, C)}{k|C(u)|} \\ & + 1_{w_k}(v)\Big(\frac{\mathbb{P}(\underline{w}_{k-1}, n-1, O, u)}{|C(u)|} + \frac{\mathbb{P}(\underline{w}_{k-1}, n-1, C)}{(k-1)|C(u)|}\Big)\Big]\end{aligned}$$

where $D(\underline{w}_k)$ is the set of deadlock nodes in $\underline{w}_k$ and $1_{w_k}(v) = 1$ if $v = w_k$ and $1_{w_k}(v) = 0$ otherwise.

The discussion on relation between redundancy and cover time has been omitted (it is commented in the .tex so it does not appear below) but I find it interesting...

3.1 Case of Trees

We consider an m-ary tree of depth h, that is, every non-leaf node has m successors, and every path from the root to a leaf has length h. Recall that n denotes the number of successive steps in a run of the algorithm.

The elementary recursion in lemma 1 (resp. in lemma 2) leads to a much more simplified one, depending only on the numbers of nodes of $\underline{w}_k$ in each level of the tree and not on $\underline{w}_k$ itself. Consider $\underline{K}_n = (K_n^1, ..., K_n^h)$, the vector of random variables expressing the number of explored nodes at each level $j = 1, ..., h$, at step n, and let $\mathbb{P}_{urs}(\underline{K}_n = \underline{k})$ the probability to cover the vector $\underline{k} = (k_1, ..., k_h)$ in n steps by URS algorithm. For SDRS, we distinguish $\mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, C)$ and $\mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, O)$ that denote the probabilities of covering $\underline{k}$ in the closed and open cases respectively. For URS, for example, the aggregation (summation) of the elementary recursion in lemma 1 on the set of all sequences $\underline{w}_k$ having k_j nodes in the level j, $j = 1, ..., h$, gives the following simplified recursion :

$$\mathbb{P}_{urs}(\underline{K}_n = \underline{k}) = \alpha(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) + \sum_{j=1}^{h} \beta_j(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_j)$$

where $\underline{k} - 1_j = (k_1, ..., k_j - 1, ..., k_h)$, $1 \leq j \leq h$. In the r.h.s. of this equation, as in the elementary one, two terms appear. The first one $\mathbb{P}^{\mathcal{R}}_{urs}(\underline{K}_n = \underline{k}) = \alpha(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k})$ is a redundancy term, while the second $\mathbb{P}^{\mathcal{I}}_{urs}(\underline{K}_n = \underline{k}) = \sum_{j=1}^{h} \beta_j(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_j)$ is the innovation term. The repetition factor $\alpha(\underline{k})$ is given by $\alpha(\underline{k}) = \frac{mk_h+k-1}{mk}$. The innovation ones are $\beta_j(\underline{k}) = \frac{mk_{j-1}-k_j+1}{m(k-1)}$.

These recursions were, in fact, computed in the goal to obtain the result of theorem 1 below related to the mean cover time. The mean time $T_A(k)$ to cover k nodes by an algorithm A (URS or SDRS) can be expressed in function of the innovation probabilities as following:

$$T_A(k) = \sum_{|\underline{k}|=k} T_A(\underline{k}), \quad T_A(\underline{k}) = \sum_{n=k}^{\infty} n\mathbb{P}^{\mathcal{I}}_A(\underline{K}_n = \underline{k})$$

With some further investigations, the mean times $T_{urs}(\underline{k})$ and $T_{sdrs}(\underline{k})$ of covering $\underline{k}$ by URS and SDRS, respectively, are given in the following theorem:

Theorem 1.

$$T_{urs}(\underline{k}) = (1 - \alpha(\underline{k}))S^1_{urs}(\underline{k}) - \alpha(\underline{k})S^0_{urs}(\underline{k})$$

$$T_{sdrs}(\underline{k}) = \sum_{j=1}^{h} \sum_{l=j}^{h} \Big[c_{j,l}(\underline{k})S^1_{sdrs}(\underline{k} - 1_{j,l}) + d_{j,l}(\underline{k})S^0_{sdrs}(\underline{k} - 1_{j,l})\Big] + a(\underline{k})\ S^1_{sdrs}(\underline{k}) - b(\underline{k})\ S^0_{sdrs}(\underline{k})$$

$$\underline{k} - 1_{j,l} = (k_1, ..., k_j - 1, ..., k_l - 1, ..., k_h)$$

See appendix **??** for the proof and the explicit formula of the intermediate statistics and the coefficients $a(\underline{k})$, $b(\underline{k})$, $c_{j,l}(\underline{k})$ and $d_{j,l}(\underline{k})$. **the appendix will be removed, we don't have space... we cannot refer to a theorem where some of the terms are not defined...**

Applying the previous result, we obtain the mean cover time computed exactly for URS and SDRS and shown in Figure 3 (left) for three parameterized trees. The notation $T(h, m)$ means that the considered tree is of height h and degree m. Note that the mean cover time is plotted in function of the coverage level (percentage of reachable nodes that are covered) rather than in terms of number of covered nodes. Given the fact that our primary interest here is redundancy, the case of a set of covered nodes going beyond the memory size is not considered. It was, then, possible to make the comparison up to the full coverage where we obtained the more significant difference in term of mean cover time between the two algorithms.

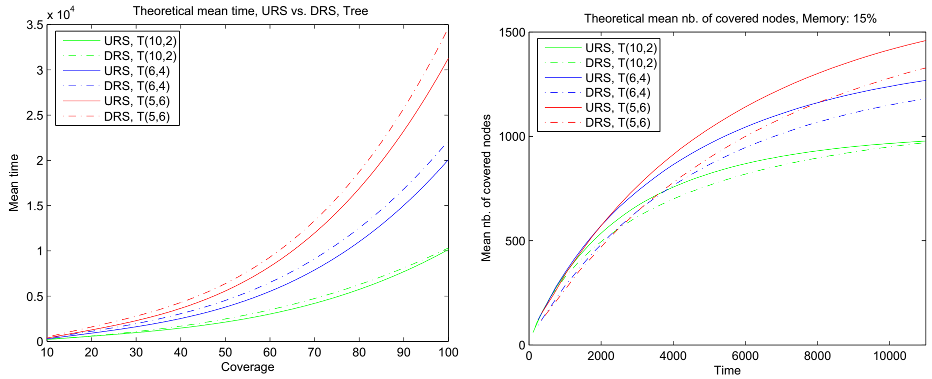

Fig. 3. Mean cover time (left) and mean number of covered nodes (right) for Trees

We can see in Figure 3 that the URS algorithm takes on average less time than SDRS to cover a given proportion of the graph. This is observed mainly for proportions more than 70% and for large trees. We define the *density factor DF* of an m-ary tree of depth h by the ratio $\frac{m}{h}$. In fact, the higher the density factor is, the larger the difference between the cover times of the algorithms is. In the case of a "thin" tree, which has small DF (typically < 0.05), SDRS can perform better than URS but this can be obtained only for extremely thin graphs.

In the following of this section we return to the more actual case, when the graph to explore is too large with respect to the memory size. We start by noting the relation in lemma 3, that holds for all algorithms A on all graphs G, between the probability $\mathbb{P}_A(K_n = k)$ to cover k nodes in n steps and the reachability probabilities $\mathbb{P}_A(v|K_n = k)$ to have, in n steps, reached a node v and covered exactly k nodes. Note that, in the case of trees, these last probabilities depend only on the node level i and not on the node v itself, because of symmetry. In the case of a grid, we must compute the probability to reach corner and non corner nodes at each level i.

Lemma 3.

$$\mathbb{P}_A(K_n = k) = \frac{1}{k} \sum_{v \in G} \mathbb{P}_A(v | K_n = k)$$

As we said above, the criterion considered here is the mean number of covered nodes function of time. Thanks to lemma 3, this can be computed basing on reachability probabilities that we first compute by returning to the elementary recursions of the algorithms. In fact, as previously, by summing these recursions on the set of the sequences $\underline{w}_k$, containing the node i and having in each level $j = 1, ..., h$, k_j nodes, one obtains recursive formula for the reachability probabilities $\mathbb{P}_{urs}(i|\underline{K}_n = \underline{k})$, $\mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}, C)$, $\mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}, O)$, and then $\mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}) = \mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}, C) + \mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}, O)$. These probabilities are defined exactly as previously except the fact that the node i is now considered to be covered. Note that these probabilities are associated with URS and SDRS without repetition and then computed for a number of covered nodes k less than the re-initialization threshold (the memory size) N. For example, for URS, one obtains, with $\gamma(\underline{k}) = \frac{1}{m(k-1)}$, :

$$\begin{aligned}\mathbb{P}_{urs}(i|\underline{K}_n = \underline{k}) = \alpha(\underline{k})\ \mathbb{P}_{urs}(i|\underline{K}_{n-1} = \underline{k}) + \sum_{j=1}^{h} \beta_j(\underline{k})\ \mathbb{P}_{urs}(i|\underline{K}_{n-1} = \underline{k} - 1_j) \\ + \gamma(\underline{k})\ \Big[\mathbb{P}_{urs}(i-1|\underline{K}_{n-1} = \underline{k} - 1_i) - \mathbb{P}_{urs}(i|\underline{K}_{n-1} = \underline{k} - 1_i)\Big]\end{aligned}$$

Once these probabilities are calculated, one sets

$$\mathbb{P}_A(i,s) = \sum_{|\underline{k}| \le N} \mathbb{P}_A(i|\underline{K}_s = \underline{k}), \ \ \mathbb{P}^*_A(i,s) = \sum_{|\underline{k}| = N} \mathbb{P}_A(i|\underline{K}_s = \underline{k})$$

where N denotes the memory size and A denotes indifferently one of the algorithms URS or SDRS. Their repeated versions will be noted RA. Then, the mean number of covered nodes of RA in function of time n is given in the theorem 2:

Theorem 2. *If N is the memory size or a prefixed threshold of re-initialization, then the mean number of covered nodes by RA is given in function of time n as:*

$$\begin{gathered} Cov(n) = \sum_{i=0}^{h} m^i \mathbb{P}_{RA}(i,n), \ \ where \\ \mathbb{P}_{RA}(i,n) = \mathbb{P}_A(i,n) + \sum_{n_1=M}^{n} [\mathbb{P}^*_A(i,n_1) + (1 - \mathbb{P}^*_A(i,n_1))\mathbb{P}_{RA}(i, n-n_1)] \end{gathered}$$

Figure 3 (right) shows the evolution of the number of covered nodes in function of time. These curves, representing the behavior of the repeated algorithms RURS and RSDRS, are plotted for three trees. The repeated algorithms are experimented for a memory size (N) of 15% w.r.t. the size of the graph. We have considered other memory sizes (10% and 20%), but the results are similar: RURS algorithm performs, clearly, better than RSDRS, especially near to the total coverage rate. We observe also that the difference between RURS and RSDRS in the number of covered nodes is more important as more as the *DF* is greater.

Note that by using the reachability probabilities $\mathbb{P}_A(i,n)$ (resp. $\mathbb{P}_{RA}(i,n)$), one can compute the minimum reachability probabilities for URS and SDRS (resp. for RURS and RSDRS) in function of time. This criterion can be very interesting in practice if, in order to detect efficiently an eventual bug in the system, which corresponds to a defective node in the modeling graph, one can take account of the worst case where the bug is localized in a node of minimum reachability probability. Note that the number of such nodes can be great as in the case of tree like graphs.

3.2 Case of Grids

We place ourselves here in the context of multi-dimensional grid. As in the previous section, we are interested in efficient computations of statistics like the mean cover time and the mean number of covered nodes for URS and SDRS. We will analyse this matter basing on the fundamental recursion in lemma 1 and 2. We first note that all possible (macroscopic and then less difficult to compute) recursion for URS or SDRS should be a summation of the corresponding elementary one on some suitably chosen set S_k of sequences $\underline{w}_k$: the coefficients in the elementary recursion must be constant on S_k and the set of the $\underline{w}_{k-1}$'s, when $\underline{w}_k \in S_k$, must be easy to identify. For clarity sake, we analyse in details the equation in lemma 1 for our algorithm URS. The coefficients $\alpha(\underline{w}_k)$ and $\beta(\underline{w}_k)$ in this recursion must be constant on S_k and the set of the $\underline{w}_{k-1}$'s, when $\underline{w}_k \in S_k$, must be easily parameterizable. This seems to be very difficult to obtain, or impossible, even in the case of infinite, oriented, grid, but this problem will be overcome as explained below. In this case the output degree of the nodes is the same, say d, and one has:

$$\alpha(\underline{w}_k) = \frac{\sum_{i=1}^{k} |C(w_i) \cap \underline{w}_k|}{k.d}, \quad \beta(\underline{w}_k) = \frac{|F(w_k) \cap \underline{w}_{k-1}|}{(k-1).d}$$

The difficulties to sum the elementary recursion satisfied by URS and SDRS, are due essentially to the great rate of communications (intersections) in the case of the grid. However, this is the same reason for which these recursions are useful in practice to calculate exact exploration statistics in this case, especially by meaning some managements. In fact due to intersections, the number of ordered sequences, with distinct nodes, generated by the algorithms is reasonable in many cases of study. Note also that the sizes of grids to be considered are in general little, as are grids in model-checking domain.

Figure 4 gives the results of comparisons of the mean covering time for three grids, where $G(L,d)$ means that the grid is of degree d and the length of each side is $L+1$. It is clear that the URS algorithm outperform SDRS. Its superiority is even more clear than in the case of graphs without intersections (tree).

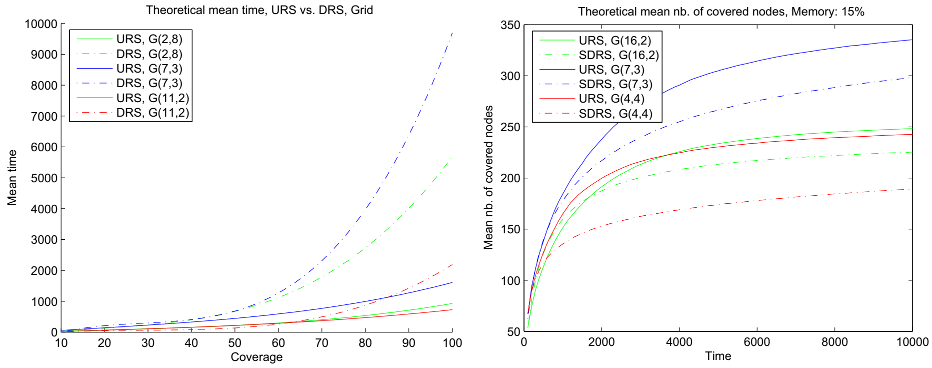

Fig. 4. Mean cover time (left) and mean number of covered nodes (right) for Grids

Moreover, for the repeated algorithms RURS and RSDRS, the mean number of covered nodes has been plotted in function of time for different grids. The reported result in Figure 4 corresponds to a memory size of 15% w.r.t. the size of the graph. As for trees, the algorithms RURS and RSDRS are experimented for three grid graphs and for three memory sizes (N) of 10%, 15% and 20% w.r.t. the size of the graphs. The results are similar for the three memory sizes: the performances RURS are clearly better than RSDRS. The superiority of RURS is more marked for high coverage and great values of the DF. This superiority is, again, more clear for grids than for trees.

4 Experimental results

We complement our theoretical analysis with a set of experimental results. We implemented the two algorithms URS and SDRS on the model checker IF [28] and ran them on several examples. Several measures were computed for each algorithm. The examples have been chosen according to the experimental needs. First, to compute the mean cover time, we have chosen some examples of medium size, in order to be able to repeat the algorithms a sufficient number of times to achieve full coverage of the reachable state space. These examples have different *density factors*, which allows us to analyse their behavior according to this parameter. Second, in order to compare the randomized algorithms with the exhaustive BFS algorithm implemented in IF, we have used the same examples, with more processes and/or data, to get graphs of very large (unknown) sizes.

Our implementations of URS and SDRS use a hash table to keep visited nodes V. When a node is completely explored (having all its successors visited), it will be deleted from the table to avoid redundant revisits. **ON THE CONTRARY, IF YOU DELETE IT FROM THE TABLE THEN YOU MIGHT VISIT IT AGAIN SINCE IT IS NO LONGER IN V: are you sure this is correct? I would suggest to remove it.** In this work, we have described the

URS and SDRS algorithms, but our implementation is more general, following the generic scheme, in particular in terms of the select function. Other variants of this scheme apart from URS and SDRS will be reported in future work. Our implementation allows the user to define the rate of leaves or internal nodes to be explored –which reflects depth- or breadth-oriented exploration– by tuning a *mixing parameter*. Choosing this parameter appropriately may require an a-priori knowledge of the graph structure (density and diameter), although, in some cases, this parameter may be computed and adapted *on the fly*.

4.1 Cover time

WE NEED TO GIVE SOME IDEA ON WHICH TYPE OF MACHINE THESE EXAMPLES WERE RUN: MODEL, HOW FAST PROCESSOR, HOW MUCH MAIN MEMORY

Each algorithm was tested on different graph examples: the *Quicksort* algorithm, the *Token Ring Protocol*, *Fischer's Mutual Exclusion Protocol* and a *Client/Server Protocol*. Table 1 shows the size (i.e., number of states) and the diameter (i.e., length of the longest acyclic path) of each example. The table also shows the density factor of the graph of each example, defined as $DF = \frac{m}{h}$, where h is the graph diameter and m is the graph degree: m is computed approximately by reference to a regular tree of size $M \approx m^h$. Thus, for a graph of size M, we let $m = \sqrt[h]{M}$.

Example	Quicksort	Token	Fischer	Server
Size (no. states)	6032	20953	34606	35182
Diameter	19	72	14	28
DF (density factor)	0.083	0.016	0.150	0.052

Table 1. Graphs description

For each example, we repeated the experiment 100 times and we computed the mean cover time of 60%, 70%, 80%, 90% and 100% of the graph. The resulting times (in seconds) are reported in Table 2. We observe that the URS algorithm performs better except for the Token example. But even in this example URS performs better for 100% coverage.

4.2 Resource-Aware vs. Exhaustive Verification

We have also experimented on very large graphs of unknown reachable size. These have been obtained by scaling-up the number of processes and/or data of the Token, Fischer and Server examples. Here we also compared URS and SDRS with an exhaustive BFS algorithm. Note that URS and SDRS re-initialize in these examples, since the state space does not fit in main memory: thus we denote

Cov. level	Algo	Quicksort	Token	Fischer	Server
60%	URS	0.389	3.283	1.841	4.490
60%	SDRS	0.641	0.752	4.070	7.441
70%	URS	0.609	4.301	2.765	5.507
70%	SDRS	0.871	1.084	5.726	8.893
80%	URS	0.882	5.744	3.809	6.821
80%	SDRS	1.411	1.584	8.173	11.966
90%	URS	1.703	8.047	5.955	9.974
90%	SDRS	4.202	2.480	13.327	19.158
100%	URS	7.723	21.247	46.097	41.452
100%	SDRS	12.459	25.221	125.091	99.460

Table 2. Mean cover time (seconds)

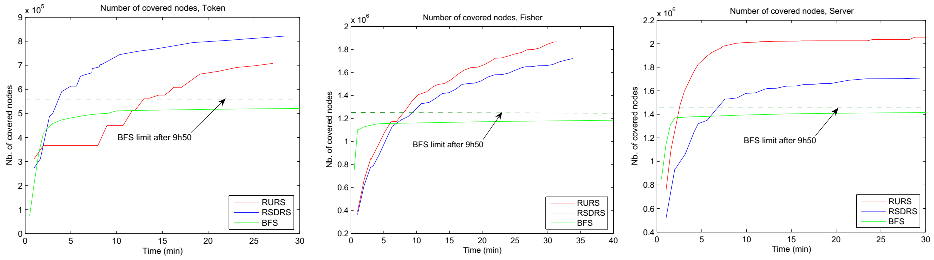

Fig. 5. The number of covered nodes evolution

them by RURS and RSDRS in the plots that follow. The number of explored states was collected over all runs and is plotted in Figure 5 as a function of time.

BFS stagnates as it approaches the limit of the the number of states that can fit in main memory. URS and SDRS go beyond this limit, and can explore up to 40% more nodes. Notice that the BFS limit occurs at a different number of nodes for each of the three case studies, even though they all use the same amount of main memory. This is because in each case study the amount of bytes needed to store a single state is different: it is higher in Token than in Server, and slightly higher in Server than in Fischer.

We observe that in the case of Fischer the randomized algorithms also stagnate after a certain amount of time. According to what we observed in our previous experiments on medium-size graphs, this happens when reaching close to 90% of the graph. In this case, exploring the "last" states becomes increasingly difficult because of redundancy.

5 Conclusions and future work

We have proposed resource-aware randomized state space exploration as a direction for research in scalable verification methods. In particular, we have proposed the URS algorithm that we believe to be the first memory-aware exploration scheme, that explicitly uses main memory resource limits to guide its behavior. Also, URS is not performing a typical random walk, in the sense that it may choose to "branch" from different nodes along a random walk path. We have proposed comparison criteria such as mean cover time and used these to compare URS with a simplified version of the DRS algorithm proposed in [11]. We have also shown via experiments, that these two algorithms, when repeated several times, can explore a state space of more than 40% in addition to that explored by an exhaustive exploration based on breadth-first search.

As part of future work we would like to experiment with industrial case studies, for instance, from the hardware or software domains. We would also like to implement and test other resource-aware verification algorithms. For instance, instead of re-initializing when the memory is full, we could have a scheme where some states in the visited set V are removed and replaced by new states. Different policies to choose which states to remove could then be envisaged.

References

1. J. P. Queille and J. Sifakis. "Specification and verification of concurrent systems in Cesar". In *Proceedings of the International Symposium in Programming*, volume 137 of Lecture Notes in Computer Science, Berlin, 1982. Spriner-Verlag.
2. E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of finite state concurrent systems using temporal logic specifications". In *ACM TOPLA*, 8(2),1986.
3. E. M. Clarke, O. Grumberg, and D. Peled. "Model Checking". *MIT Press*, 1999

4. M. O. Rabin. “Probabilistic algorithms”. In *J. Traub, editor, Algorithms and Complexity: New Directions and Recent Results*, pages 2-39. Academic Press, New York, 1976.
5. R. Motwani, P. Raghavan. ”Randomized Algorithms”. Cambridge University Press 2005
6. J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. ”Symbolic model checking: 10^{20} states and beyond”. In *5th Conference on Logic In Computer Science (LICS)*, pages 428-439, june 1990
7. R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing Random Walk state space exploration. In *FMICS '05: Processing of the 10th international workshop on formal methods for industrial critical systems*, pages 98-105. ACM Press, 2005
8. C. H. West. Protocol validation by random state exploration. In *International Symposium on Protocol Specification, testing and Verification,* 1986
9. D. Owen. and T. Menzies Lurch. A lightweight alternative to model checking. In *Proc. of Software Engineering and Knowledge Engineering (SEKE'2003)*, pages 158-165
10. R. Grosu and S. A. Smolka. ”Monte Carlo model-checking”. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *LNCS*, pages 271-286. Springer, 2005.
11. R. Grosu, X. Huang, S.A. Smolka, W. Tan and S. Tripakis. ”Deep Random Search for Efficient Model Checking of Timed Automata”. In *Proc. of MW'06, the 7th Monterey Workshop on Composition of Embedded Systems*, pages 37-48, Paris, October 2006
12. P. Haslum. Model checking by random walk. In *Proc. of ECSEL Workshop*, 1999
13. M. Mihail and C. H. Papadimitriou. ”On the random walk method for protocol testing”. In *Proc. Computer-Aided Verification (CAV 1994)*, volume 818 of LNCS, pages 132-141, 1994.
14. H. Sivaraj and G. Golpalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *Proc. of Parallel and Distributed Model Checking (PDMC'03)*, volume 89 of *ENTCS*, 2003
15. A. Kuehlmann, K. L. McMillan, and R. K. Brayton. Probabilistic state space search. In *Proc. of Computer-Aided Design (CAD 1999)*, pages 574-579. IEEE Press, 1999.
16. J. Geldenhyus. State caching reconsidered. In *SPIN Workshop*, volume 2989 of *LNCS*, pages 23-39, 2004
17. P. Godefroid, G. J. Holzmann, and D. Pirottin. ”State space caching revisited”. In *Proc. of Computer Aided Verification (CAV 1992),* volume 663 of *LNCS*, pages 178-191, 1992
18. P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. 2nd International Conference on Computer Aided Verification,* volume 531 of *LNCS*, pages 176-185, 1990
19. P. Godefroid. On the costs and benefits of using partial order methods for the verification of concurrent systems. In *Proc. Workshop on Partial Order Methods in Verification,* DIMACS series, volume 29, pages 289-303, 1996
20. E. Tronci, G. D. Penna, B. Intrigila, and M. Venturini. A Probabilistic approach to automatic verification of concurrent systems. In *Proc. of Asia-Pacific Software Engineering Conference (ASPEC 2001)*, 2001
21. F. Lin, P. Chu, and M. Liu. Protocol verification using reachability analysis: The state space explosion problem and relief strategies. *Computer Communication Review.* volume 17(5):126-134, 1987

22. G. J. Holzmann. An analysis of bi-state hashing. In *Proc. of Protocol Specification, Testing and Verification*, pages 301-314, 1995
23. G. J. Holzmann. Automated protocol validation in Argos, assertion proving and scatter searching. In *IEEE trans. on Software engineering,* volume 13(6):683-696, 198717(5):126-134, 1987
24. U. Feige. "A Tight Upper bound on the cover time for Random walks on graphs". In *Random Structures and Algorithms.* Volume 6(1), pages 51-54, 1995
25. U. Stem, and D. L. Dill. Improved probabilistic verification by hash compaction. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206-224, 1995
26. R. Nalumasu, and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative provision implementation. In *Formal Methods for System Design*, volume 20(3), pages 206-224, 1995
27. E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9(1-2): 77-104, 1996
28. M. Bozga, J. C. Fernandez, L. Ghirvu, S. Graf, J. P. Krimm, and L. Mounier. "IF: a Validation Environment for Timed Asynchronous Systems". In *Proc. Computer-Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 543-547, 2000