HAL
open science

Systems Verification using Randomized Exploration of Large State Spaces

Nazha Abed, Stavros Tripakis, Jean-Marc Vincent

To cite this version:

Nazha Abed, Stavros Tripakis, Jean-Marc Vincent. Systems Verification using Randomized Exploration of Large State Spaces. SPIN, 2008, Los Angeles. hal-00953617

HAL Id: hal-00953617

https://inria.hal.science/hal-00953617

Submitted on 28 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Systems Verification using Randomized Exploration of Large State Spaces

Nazha Abed, Stavros Tripakis, and Jean-Marc Vincent

LIG, 51, avenue Jean Kuntzmann, 38330 Montbonnot Saint-Martin, France
Cadence Laboratories, 2150 Shattuck, Avenue 10th Floor, Berkeley, CA 94704
{Nazha.Abed,Jean-Marc.Vincent}@imag.fr, Tripakis@cadence.fr

Abstract. System verification is a technique used to improve the correctness of hardware and software systems. It aims to discover bugs in early development steps. A common approach of system verification consists of exploring and analyzing the reachable states graph, which represents the system behavior in an exhaustive manner. This graph is often too large to be entirely explored: its size grows exponentially in the number of system components. The verification task then becomes a task of partial exploration, subject to constraints on memory and verification time. Several methods of random partial exploration have been proposed based mostly on random walk. In this paper [1], we present a general strategy of randomized algorithms, in particular a Uniform Random Search to perform partial, but considerable, state space exploration with little memory and time requirements.

1 Introduction

To verify system correctness, one can proceed by exhaustive verification (e.g. model checking) or testing. Model checking [1] [2] [3] -the problem of deciding whether a property holds in a system specification- 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 leading the user to the error state is produced. Model checking however is not without its drawbacks, the most prominent of which is state space explosion: the phenomenon where the size of a system's state space grows exponentially in the size of its specification. State space 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

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

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.

Researchers have developed a plethora of techniques aimed at curtailing state space 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 to equilibrate the exploration of the state transition graph. In particular, the techniques of partial exploration based on randomized algorithms. These techniques have been shown to be very effective in practice to find errors or explore transition graphs. A randomized algorithm is one which contains an assignment to a variable based on the outcome of tossing a fair coin or a random number generator. Randomized algorithms are extensively used, basically for two reasons: simplicity and speed [4]. A consequence of using randomization is the fact that the correctness or termination statements is given with some controlled probability.

The randomized algorithms proposed in the literature are –in their quasi-totality– based on random walk. A random walk on a graph is a stochastic process of type "Markov chain". The algorithm starts from the initial state, and at each step, it chooses in a uniform way a successor of the current state and visits it. This choice is independent to 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 has been exploited for verification tasks either for graph covering and reachability analysis. It was applied first to model-checking by West in 1986 [8] which demonstrates that efficient sampling of the reachable state space by random walk suffices to ensure the effectiveness of testing real models. In the last few years, the studies succeeded in exploring this scheme, and random walk has been used for verification in the model checker

Lurch [9]. Some theoretical results are given when working on a restricted class of graphs. For example, in [12], an upper bound of the number of steps needed by Random walk to ensure, with probability $1-\epsilon$, the covering of all the graph is provided. It is given by:

$$\frac{1}{\epsilon}|V||E| \tag{1}$$

where V denotes the set of the graph states and E the graph edges. This bound is very large in practice and holds only for closed strongly connected graphs. These results are so restricted and not very useful in model checking. In general, the most results are based on experimentation performed on real and random graphs.

In [10], the authors define P_z as the probability of detecting a bug in one run of the random walk. This probability depends naturally on the existence of the bug and also on the capacity of the algorithm to detect it. An upper bound of the number R of repetitions needed by random walk to detect a counter-example, with probability $1-\delta$, under the assumption that $P_z \geq \epsilon$, is given by:

$$R = \frac{\ln(\delta)}{\ln(1-\epsilon)}$$

If, after R iterations, no bug is detected, the algorithm reports that the probability of finding bugs through further sampling, under the assumption that $P_z \geq \epsilon$ is less than δ. Note that P_z is, in general, unknown and difficult to estimate. Then, in order to ensure the required assumption, one has to choose ϵ little enough, which can render R too large.

Because it is completely memory free, the random walk method cannot distinguish between visited and not visited states, and so it may spend large time to visit repeatedly some few states (the redundancy property). Because of this, covering the entire graph (or a high portion of it) may need a prohibitively large amount of time (see equation 1 above). 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 can not be well explored with only simple random walk. This may be the case for example if one know that it is near to a target node. Guided search decides of the next exploration direction based on general information about the graph and the system semantic. 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 made in a deterministic fashion (BFS, DFS) or by random methods. In [20], the proposed algorithm uses a BFS exploration method with a randomized partial storage. When the memory is over, the algorithm proceeds at a lower speed but do not give up the verification. As reported in [20], this algorithm can save 30% of the memory with an average time penalty of 100%.

As we have seen, all the above mentioned random methods, based on Random Walk, improve the redundancy of exploration but the cover time still very large. In this paper, we propose methods that aim to further improve exploration by avoiding redundancy and reducing the cover time. First, a general scheme that encompasses all previously mentioned methods is given. Then, a Uniform Random Search (URS) algorithm is proposed based on a different selection function than random walk (RW). While RW is a depth-oriented algorithm, our algorithm 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 randomized exploration scheme lies in the fact that it explicitly uses a parameter N which represent the maximum number of states that can be stored in main memory at any given time. Thus, our algorithms are *resource-aware*. Main memory is the main bottleneck in exhaustive verification, for reasons we explain below.

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.

The rest of the paper is organized as follows: The proposed scheme and algorithm are detailed in section II. Section III gives some general theoretical results that are projected on two cases of regular graphs in sections IV and V. Experimental results are summarized in section VI, while section VII 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 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 on $[0, 1]$, 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 in *swapping* memory to/from disk with very few additional states explored. Instead, we choose a 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

General Random Exploration Algorithm

V : **set of stored nodes (visited)**;
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**
  $v \leftarrow \text{select}(V, P, I)$;
  $\text{check}(v)$; *//verify if the property holds*
  $(V, I) \leftarrow \text{update}(V, v, P, I)$;
done

A random exploration algorithm can be cast into the general scheme shown above. 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].

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. 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 is reached. This stop condition that takes into account the parameter N is very important in improving the exploration.

```
Uniform Random Search URS

V : set of stored nodes (visited);
N : Maximum size of V;
n : Maximum number of steps;
v, u : nodes;
i, j : integer;

V ← {v0};
i ← 0;
j ← 0;

While ((j ≤ N) and (i ≤ n)) do
    u ← pick uniformly one node from V;
    If (Succ(u) ≠ ∅) then
        v ← pick uniformly one node from Succ(u);
        If (v ∉ V) then
            check(v);
            V ← V ∪ {v};
            j ← j + 1;
        end If
    end If
    i ← i + 1;
done
```

[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 all *non-closed* nodes[2] – in each step of the algorithm – can be stored in the available memory, which is not always the case in practice. In this paper we use a simplified version of DRS, that we call SDRS. The latter, like URS, uses a parameter N modeling an upper bound on the number of states that can be stored at any given time. This puts the two algorithms in the same framework and allows comparisons. SDRS, has as *stop condition* the exhaustion of the states in memory. The *select* and *update* functions are the same as the simple random walk except the re-initialization of the current node (*update* function) which is made by a node chosen randomly in V and not by the initial node.

SDRS will be studied in detail and compared to the URS algorithm described above. According to [11], DRS outperforms the simple RW, because when blocked, it is reinitialized from a random visited state instead of the initial one and uses additional memory to distinguish from visited and non visited states which avoid much of redundant explorations. For this reason, we omit comparison with simple RW here and only compare with SDRS.

[2] A *closed* node is one that has all its successors visited.

```
Simplified Deep Random Search SDRS

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

V ← {v0};
v ← v0;
i ← 0;
j ← 0;

While ((j ≤ N) and (i ≤ n)) do
    If (Succ(v) = ∅) then
        v ← pick uniformly one node from V;
    else
        v ← pick uniformly one node from Succ(v);
        If (v ∉ V) then
            check(v);
            V ← V ∪ {v};
            j ← j + 1;
        end If
    end If
    i ← i + 1;
done
```

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

The used evaluation criteria are based on our initial objective, which is to come up with more robust exploration algorithms. On one hand, improve the cover time of existing randomized algorithms and on the other hand improve the reachability and the coverage of existent exhaustive methods. We define our criteria in two ways: stochastic and experimental.

One considered criterion to study the algorithms performance is the *mean cover time*. The cover time is the number of steps needed by a given algorithm which starts at the initial state to cover some percentage or all the graph nodes (i.e., to reach some coverage level). For undirected graphs, the mean cover time of any graph is polynomial [24]. For directed graphs –like the ones arising in model checking– it is in general exponential, except for some restricted classes of directed graphs [12]. These classes are so restricted that they are not very interesting for model checking. The mean cover time gives a good indication on the capacity of the algorithm to reach states and explore most of the graph. It informs us on the estimated time to reach all nodes. A randomized algorithm that has a better average cover time, has less redundancy in its exploration. 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 is the cover time of 95% of the graph. Some exploration algorithms will provide this answer in less time than others.

When the number of all reachable nodes is unknown, as is the case with very large real models, we compare the number of *covered* nodes (i.e., visited nodes). As the number of the visited nodes increases, the probability that a node already visited either is revisited 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 possible criterion consists of the *minimum reachability probability* over all reachable nodes. Reachability probability models the capacity of an algorithm to reach a target state. Indeed the problem of the model checking can be seen as the search of an error state in the state space. Due to the fact that the considered exploration algorithms are random, the list of the visited nodes V is a random variable that depends on the algorithm and the particular graph structure. It results from this, that 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, reiterating the randomized algorithm improves the probability of reaching states and finding errors.

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 is 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 General theoretical results

This section aims to efficiently compute various statistics for our algorithm URS in some interesting cases of study. We also provide the same statistics for the algorithm SDRS. This set of results allows a theoretical comparison of the two algorithms and demonstrates a superiority of URS in most studied cases. More precisely, what we investigate here, is 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. This will be done for two extreme types of graphs. The first one is trees. Many trees will be considered and parameterized by a density factor which the comparison results depend upon. The second one is grids. In contrast to tree graphs, a multi-dimensional grid represents many intersections between nodes which a priori can lead to significant change in the behaviour of the algorithms and therefore in their performances. Nevertheless, we will show that URS outperforms SDRS in most cases of trees and grids. Before we analyze the case of trees and grids separately, we first provide in this section some general results that apply to any graph.

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

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

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

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:*

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

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

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.

Note that the elementary recursion in lemma 1 (resp. in lemma 2) is satisfied by URS (resp. by SDRS) algorithm for any graph. In the next two sections, we specialize these results to trees and grids.

4 Case of Trees

We place ourselves first, in the context of 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 appears. 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:

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

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.

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

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

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

See appendix C 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 $d_{j,l}(\underline{k})$.

Applying the previous result, we obtain the mean cover time computed exactly for URS and SDRS and shown in figure 1 below 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 traced in function of the coverage level rather than the number of covered nodes. Giving the fact that our interest is focused here on the redundancy comparison, 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.

We can see in figure 1 that the URS algorithm takes in average less time than SDRS to cover a given proportion of the graph. This is observed mainly for

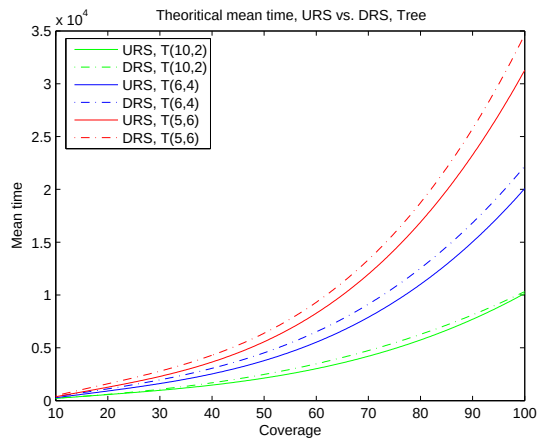

Fig. 1. Mean Cover Time for Tree

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 algorithm A on all graph G, between the probability $P_A(K_n = k)$ to cover k nodes in n steps and the reachability probabilities $P_A(v|K_n = k)$ to have, in n steps, reaching a node v and covering 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 $\gamma(\underline{k}) = \frac{1}{m(k-1)}$, :

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

Once, these probabilities are calculated, one sets

$$P_A(i,s) = \sum_{|\underline{k}| \leq N} P_A(i|\underline{K}_s = \underline{k}), \quad P_A^*(i,s) = \sum_{|\underline{k}| = N} P_A(i|\underline{K}_s = \underline{k})$$

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:*

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

We observe in figure 2, 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 traced 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.

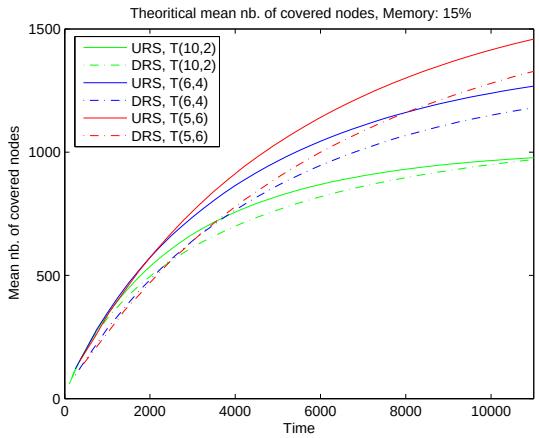

Fig. 2. Mean number of covered nodes for Tree

5 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 overcame 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 3 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).

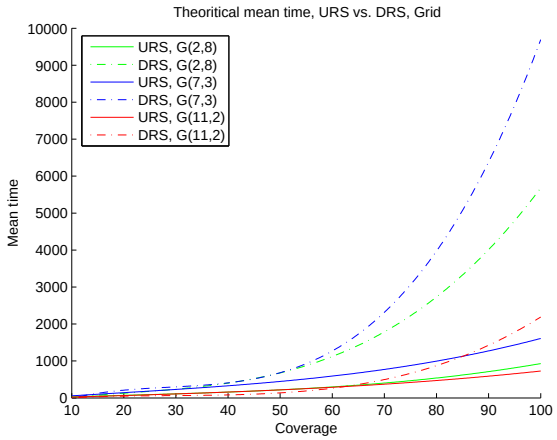

Fig. 3. Mean Cover Time for Grid

Moreover, for the repeated algorithms RURS and RSDRS, the mean number of covered nodes has been traced 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.

6 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 run them on several examples. Several measures were computed for each algorithm. The examples have been chosen according to the experiments needs.

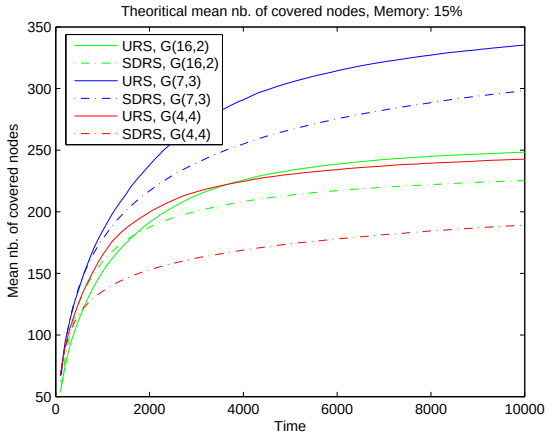

Fig. 4. Mean number of covered nodes for Grid

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. This facilitates the storage and the search. When a node is completely explored (having all its successors visited), it will be deleted from the table to avoid redundant revisits. 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 mx. 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*.

6.1 Cover time

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 the degree m is computed approximately by reference to a regular tree where the size of the tree is $M \approx m^h$. Thus, for a graph of size M, we let m be the h-root of 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 case, 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 time for each case study is reported in Table 2.

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)

We observe that the URS algorithm performs better in large graphs, for which the $DF > 0.07$. For a small $DF < 0.03$, SDRS performs better for non total coverage (coverage at 60%, 70%, 80% and 90%). For medium values of $DF \approx 0.05$, URS performs better also. These results are reported in the following curves which are compatible with the theoretical ones.

6.2 Partial vs. Exhaustive

We have also experimented on very large graphs of unknown reachable size. We have run BFS exhaustive exploration, and RURS, RSDRS partial algorithms on each one. The number of explored states was collected over all runs and

compared to the exhaustive execution. So, the execution evolution curves have been drawn as a function of time.

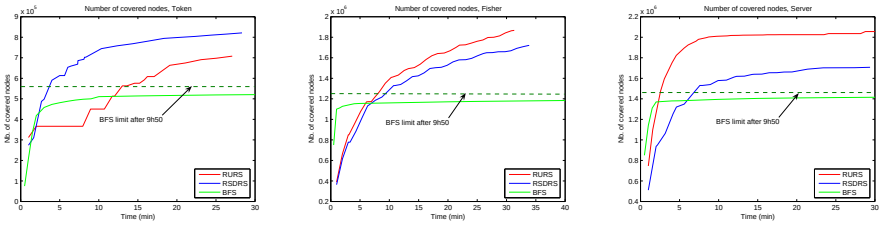

Fig. 5. The number of covered nodes evolution

We observe in Figure 5, that the exhaustive BFS exploration stagnates after a certain number of explored nodes. This limit corresponds to the number of states supported by the available memory. The repeated partial exploration algorithms URS and SDRS go beyond this limit, and can explore up to 40% more nodes than their deterministic counter part. 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. The URS algorithm is generally better than SDRS but we think that this depends on the graph DF which is unknown here.

We observe that in some cases (e.g., Figure 5: Fischer), the randomized exploration algorithms also stagnate after a certain amount of time. According to 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.

7 Conclusion

We have presented a generic randomized state space exploration scheme that unifies many randomized exploration algorithm variants. In particular, we have proposed the Uniform Random Search algorithm that we believe to be the first randomized algorithm that explicitly uses main memory resource limits to guide its behavior. URS is also 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 compared URS with a simplified version of Deep Random Search, a performant and optimized algorithm based on random walk [11]. We also propose "reinitialized" variants of the above two elementary algorithms, called RURS and

RSDRS, where each time the memory is full the algorithm is restarted and repeated several times. We have used a density factor parameter to classify graphs into "thin" and "large". We performed a detailed theoretical study to compute the mean cover time of URS and SDRS and the mean number of covered nodes of RURS and RSDRS. Many of our results are available only for special classes of graphs, namely trees and grids, but may give some insight of what happens in more general graph structures.

Both our theoretical and experimental results show that the URS algorithm explores in a more uniform fashion and so covers the state graph more rapidly in most cases. However, in some cases, in particular when the graph is thin, SDRS performs as well or better than URS. We have also shown via experiments, that these two algorithms, when repeated several times, can –in the case of very large graphs exceeding the size of main memory– explore a state space of more than 40% in addition to that explored by an exhaustive exploration based on breadth-first search.

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

A. Proof of lemma 1

The probability $\alpha(\underline{w}_k)$ to revisit a node among $\underline{w}_k$ is the sum, for each w_i in $\underline{w}_k$, $i = 1, ..., k$, of $1/k$, which is the probability to choose the father w_i, multiplied by the factor $\frac{|C(w_i) \cap \underline{w}_k|}{|C(w_i)|}$ expressing the probability to choose a child of w_i in $\underline{w}_k$. For $\beta(\underline{w}_k)$, the factor $\frac{1}{k-1}$ corresponds to the choice of a father v of w_k in $\underline{w}_{k-1}$, and then the choice of w_k, with probability $\frac{1}{|C(v)|}$. This ends the proof.

B. Proof of 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, at the end of step n, in a closed point (resp. in an open point at node v). We denote by $D(\underline{w}_k)$ the set of deadlock nodes in $\underline{w}_k$ and we set $1_{w_k}(v) = 1$ if $v = w_k$ and $1_{w_k}(v) = 0$ otherwise.

Then for $\mathbb{P}(\underline{w}_k, n, C)$, since it must be in a closed point, no node is newly reached at step n: at step $n-1$ the algorithm reached a deadlock node and at step n it, unsuccessfully, looked for a successor of this node so that it will be in a closed point by step n. So there is two cases: by step $n-1$, the exploration is in a closed point or in an open point at some deadlock node v. In the first case, it must restart at step n, with probability $\frac{1}{k}$, from a deadlock point, which gives the term $\frac{|D(\underline{w}_k)|}{k}$. In the second case, the exploration is open at node v, so it must continue in the set of successors of v. This set is empty so the exploration reaches a close point with probability 1. This gives the recursion:

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

For $\mathbb{P}(\underline{w}_k, n, O, v)$, there is 4 cases for the algorithm $SDRS$:

- Case 1: no new node is covered at step n and, by step $n-1$, it was in an open point at some node u: so u must be in $F(v) \cap \underline{w}_k$ (i.e. a father of v in $\underline{w}_k$) and at step n, v is chosen uniformly among $C(u)$ (: with probability $\frac{1}{|C(u)|}$). This gives the first term in the recursion (below).
- Case 2: no new node is covered at step n and, by step $n-1$, it was in a closed point: so at step n, it chooses, with probability $\frac{1}{k}$, a node u in $F(v) \cap \underline{w}_k$ and picks v uniformly among $C(u)$. This gives the second term in the recursion.
- Case 3: a new node v is covered at step n, it must be w_k since the sequence is stored in visiting order, and, by step $n-1$, the exploration was in an open point at some node u: so u must be in $F(v) \cap \underline{w}_{k-1}$ and at step n, v is chosen uniformly among $C(u)$. This gives the third term in the recursion. Note that the term $1_{w_k}(v)$ expresses the fact that one must have $v = w_k$, otherwise the third and fourth terms are not considered in the recursion.
- Case 4: a new node is covered at step n and, by step $n-1$, it was in a closed point: so at step n, it chooses, with probability $\frac{1}{k-1}$, a node u in $F(v) \cap \underline{w}_{k-1}$ and picks v uniformly among $C(u)$. This gives the fourth term in the recursion.

This ends the proof and one obtains the underlined recursion for $SDRS$:

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

C. Proof of theorem 1

We start by showing the first statement:

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

First, we have already obtained, before the statement of the theorem 1, the following recursion that expresses the probability to cover the vector $\underline{k} = (k_1, ..., k_h)$ in n steps by respect to URS Algorithm:

$$\mathbb{P}_{urs}(\underline{K}_n = \underline{k}) = \alpha(\underline{k})\, \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) + \sum_{j=1}^{h} \beta_j(\underline{k})\, \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_j) \tag{2}$$

The mean cover time of URS is then given by:

$$\begin{aligned} T_{urs}(\underline{k}) &= \sum_{n=k}^{\infty} n \mathbb{P}^{\mathcal{J}}_{urs}(\underline{K}_n = \underline{k}) \\ &= \sum_{n=k}^{\infty} n \sum_{i=1}^{h} \beta_i(\underline{k})\, \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \\ &= \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k}^{\infty} n\, \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \\ &= \sum_{i=1}^{h} \beta_i(\underline{k}) \times S(\underline{k} - 1_i), \end{aligned}$$

with

$$\begin{aligned}
S(\underline{k}-1_i) &= \sum_{n=k}^{\infty}((n-1)+1)\ \mathbb{P}_{urs}(\underline{K}_{n-1}=\underline{k}-1_i)\\
&= \sum_{n=k}^{\infty}\mathbb{P}_{urs}(\underline{K}_{n-1}=\underline{k}-1_i)\\
&+ \sum_{n=k}^{\infty}(n-1)\ \mathbb{P}_{urs}(\underline{K}_{n-1}=\underline{k}-1_i)\\
&= \sum_{n=k-1}^{\infty}\ \mathbb{P}_{urs}(\underline{K}_{n}=\underline{k}-1_i)\\
&+ \sum_{n=k-1}^{\infty}\ n\ \mathbb{P}_{urs}(\underline{K}_{n}=\underline{k}-1_i)\\
&= S^0_{urs}(\underline{k}-1_i)+S^1_{urs}(\underline{k}-1_i),
\end{aligned}$$

where

$$\begin{aligned}
S^0_{urs}(\underline{k}) &= \sum_{n=k}^{\infty}\mathbb{P}_{urs}(\underline{K}_{n}=\underline{k})\\
S^1_{urs}(\underline{k}) &= \sum_{n=k}^{\infty}n\ \mathbb{P}_{urs}(\underline{K}_{n}=\underline{k})
\end{aligned}$$

Then:

$$T_{urs}(\underline{k}) = \sum_{i=1}^{h}\beta_i(\underline{k})\Big(S^0_{urs}(\underline{k}-1_i)+S^1_{urs}(\underline{k}-1_i)\Big) \tag{3}$$

Using the recursion 2, one obtains:

$$\begin{aligned}
S^0_{urs}(\underline{k}) &= \sum_{n=k}^{\infty}\mathbb{P}_{urs}(\underline{K}_{n}=\underline{k})\\
&= \sum_{n=k}^{\infty}[\alpha(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1}=\underline{k})\\
&+ \sum_{i=1}^{h}\beta_i(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1}=\underline{k}-1_i)]\\
&= \alpha(\underline{k})\sum_{n=k}^{\infty}\mathbb{P}_{urs}(\underline{K}_{n-1}=\underline{k})\\
&+ \sum_{i=1}^{h}\beta_i(\underline{k})\sum_{n=k}^{\infty}\ \mathbb{P}_{urs}(\underline{K}_{n-1}=\underline{k}-1_i)
\end{aligned}$$

Note that for $n=k$, $\mathbb{P}_{urs}(\underline{K}_{n-1}=\underline{k})=0$, because in $k-1$ steps the algorithm cannot cover more than $k-1$ nodes, so it cannot cover the vector $\underline{k}$ which contains k nodes. Then

$$S^0_{urs}(\underline{k}) = \alpha(\underline{k}) \sum_{n=k+1}^{\infty} \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i)$$

So, by the variable change $n := n - 1$, one has:

$$\begin{aligned} S^0_{urs}(\underline{k}) &= \alpha(\underline{k}) \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \\ &+ \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k-1}^{\infty} \mathbb{P}_{urs}(\underline{K}_n = \underline{k} - 1_i) \\ &= \alpha(\underline{k}) S^0_{urs}(\underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k}) S^0_{urs}(\underline{k} - 1_i) \end{aligned}$$

Then,

$$S^0_{urs}(\underline{k}) = \frac{1}{1 - \alpha(\underline{k})} \sum_{i=1}^{h} \beta_i(\underline{k})\ S^0_{urs}(\underline{k} - 1_i) \tag{4}$$

Similarly,

$$\begin{aligned} S^1_{urs}(\underline{k}) &= \sum_{n=k}^{\infty} n \mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \\ &= \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_n = \underline{k}) + \sum_{n=k}^{\infty} (n-1) \mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \\ &= S^0_{urs}(\underline{k}) + \sum_{n=k}^{\infty} (n-1) [\alpha(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) \\ &+ \sum_{i=1}^{h} \beta_i(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i)] \\ &= S^0_{urs}(\underline{k}) + \alpha(\underline{k}) \sum_{n=k}^{\infty} (n-1) \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) \\ &+ \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k}^{\infty} (n-1) \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \end{aligned}$$

Then as previously, by making the variable change $n := n-1$, one obtains:

$$\begin{aligned}
S^1_{urs}(\underline{k}) &= S^0_{urs}(\underline{k}) + \alpha(\underline{k}) \sum_{n=k+1}^{\infty} (n-1)\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) \\
&+ \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k}^{\infty} (n-1)\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \\
&= S^0_{urs}(\underline{k}) + \alpha(\underline{k}) \sum_{n=k}^{\infty} n\mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \\
&+ \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k-1}^{\infty} n\ \mathbb{P}_{urs}(\underline{K}_n = \underline{k} - 1_i) \\
&= S^0_{urs}(\underline{k}) + \alpha(\underline{k}) S_1(\underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k}) S_1(\underline{k} - 1_i)
\end{aligned}$$

and then,

$$S^1_{urs}(\underline{k}) = \frac{1}{1-\alpha(\underline{k})} \Big(S^0_{urs}(\underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k})\ S^1_{urs}(\underline{k} - 1_i) \Big) \tag{5}$$

Consequently, by equation (3), one has:

$$T_{urs}(\underline{k}) = \sum_{i=1}^{h} \beta_i(\underline{k}) S^0_{urs}(\underline{k} - 1_i) + \sum_{i=1}^{h} \beta_i(\underline{k}) S^1_{urs}(\underline{k} - 1_i)$$

and then by applying equations (4) and (5), one obtains:

$$\begin{aligned}
T_{urs}(\underline{k}) &= (1-\alpha(\underline{k}))S^0_{urs}(\underline{k}) + [(1-\alpha(\underline{k}))S^1_{urs}(\underline{k}) - S^0_{urs}(\underline{k})] \\
&= (1-\alpha(\underline{k}))S^1_{urs}(\underline{k}) - \alpha(\underline{k})S^0_{urs}(\underline{k})
\end{aligned}$$

Now for the second recursion, related to SDRS, we first established similar recursions for the probability $P_{sdrs}(\underline{K}_n = \underline{k}, C)$ (resp. $P_{sdrs}(\underline{K}_n = \underline{k}, O)$) of covering $\underline{k} = (k_1, ..., k_h)$ (i.e. k_i nodes are covered at each level $i = 1, ..., h$) in n steps and being in a closed (resp. an open point) of the exploration:

$$\mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}) = \mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, C) + \mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, O)$$

and

$$\begin{aligned}
\mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, C) = \sum_{s=1}^{h+1} \frac{1}{m^{s-1}} \Big[&\alpha(\underline{k}, C)\ \mathbb{P}_{sdrs}(\underline{K}_{n-s} = \underline{k}, C) \\
&+ \sum_{j=h-s+2}^{h} \beta_j(\underline{k}, C)\ \mathbb{P}_{sdrs}(\underline{K}_{n-s} = \underline{k} - 1_{j,h}, C) \Big]
\end{aligned} \tag{6}$$

where:

$$\alpha(\underline{k}, C) = \frac{k_h}{k}$$
$$\beta_j(\underline{k}, C) = \frac{mk_{j-1} - (k_j - 1)}{(k - h + j - 1)m^{j-h}}$$
$$1_{j,j'} = 1_j + 1_{j+1} .... + 1_{j'}.$$

$$\mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, O) = \sum_{s=1}^{h} \frac{1}{m^s} \Big[\alpha(\underline{k}, O)\ \mathbb{P}_{sdrs}(\underline{K}_{n-s} = \underline{k}, C)$$
$$+ \sum_{j=1}^{h} \sum_{l=\max(j,0)}^{\min(s+j-1,h)} \beta_{j,l}(\underline{k}, O)\ \mathbb{P}_{sdrs}(\underline{K}_{n-s} = \underline{k} - 1_{j,l}, C) \Big] \tag{7}$$

where:

$$\alpha(\underline{k}, O) = \frac{k_s + ... + k_h}{k}$$
$$\beta_{j,l}(\underline{k}, O) = \frac{mk_{j-1} - (k_j - 1)}{(k - l + j - 1)m^{j-l}}$$

Then, the mean cover time of SDRS is given by:

$$T_{sdrs}(\underline{k}) = \sum_{n=k}^{\infty} n \mathbb{P}^{\mathcal{I}}_{sdrs}(\underline{K}_n = \underline{k})$$

Note that the innovation probability for $SDRS$ is composed by two terms:

$$\mathbb{P}^{\mathcal{I}}_{sdrs}(\underline{K}_n = \underline{k}) = \mathbb{P}^{\mathcal{I}}_{sdrs}(\underline{K}_n = \underline{k}, C) + \mathbb{P}^{\mathcal{I}}_{sdrs}(\underline{K}_n = \underline{k}, O)$$

where

$$\mathbb{P}^{\mathcal{I}}_{sdrs}(\underline{K}_n = \underline{k}, C) = \sum_{s=1}^{h+1} \frac{1}{m^{s-1}} \Big[\sum_{j=h-s+2}^{h} \beta_j(\underline{k}, C) \times$$
$$\times \mathbb{P}_{sdrs}(\underline{K}_{n-s} = \underline{k} - 1_{j,h}, C) \Big]$$
$$\mathbb{P}^{\mathcal{I}}_{sdrs}(\underline{K}_n = \underline{k}, O) = \sum_{s=1}^{h} \frac{1}{m^s} \Big[\sum_{j=1}^{h} \sum_{l=\max(j,0)}^{\min(s+j-1,h)} \beta_{j,l}(\underline{k}, O) \times$$
$$\times \mathbb{P}_{sdrs}(\underline{K}_{n-s} = \underline{k} - 1_{j,l}, C) \Big]$$

Then, as previously, T_{sdrs} can be expressed in function of mean statistics S^0_{sdrs} and S^1_{sdrs}, where:

$$S^0_{sdrs}(\underline{k}) = \sum_{n=k}^{\infty} \mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, C)$$
$$S^1_{sdrs}(\underline{k}) = \sum_{n=k}^{\infty} n\ \mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, C)$$

From equations (6) and (7), one obtain the recursions:

$$\begin{aligned} S^0_{sdrs}(\underline{k}) &= \sum_{j=1}^{h} \gamma^0_j(\underline{k})\ S^0_{sdrs}(\underline{k} - 1_{j,h}) \\ S^1_{sdrs}(\underline{k}) &= \sum_{j=1}^{h} \Big(\gamma^1_j(\underline{k}) S^1_{sdrs}(\underline{k} - 1_{j,h}) + \delta_j(\underline{k})\ S^0_{sdrs}(\underline{k} - 1_{j,h})\Big) \\ &\quad + \mu(\underline{k}) S^0_{sdrs}(\underline{k}) \end{aligned}$$

where,

$$\begin{aligned} \gamma^0_j(\underline{k}) = \gamma^1_j(\underline{k}) &= \frac{d_0(h-j+1,h)\beta_j(\underline{k},C)}{1 - d_0(0,h)\alpha(\underline{k},C)} \\ \delta_j(\underline{k}) &= \frac{d_1(h-j+1,h)}{d_0(h-j+1,h)} \gamma^0_j(\underline{k}) \\ \mu(\underline{k}) &= \frac{d_0(0,h)\alpha(\underline{k},C)}{1 - d_0(0,h)\alpha(\underline{k},C)} \\ d_0(j,j') &= \frac{\frac{1}{m^{j-1}} - \frac{1}{m^{j'}}}{m-1} \\ d_1(j,j') &= \frac{m}{m-1} d_0(j,j'+1) + \frac{\frac{j}{m^{j-1}} + \frac{j'+2}{m^{j'}}}{m-1} \end{aligned}$$

Finally similar computations as those made for URS algorithm, give the requested equation for $SDRS$:

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

where the coefficients in this equation are given by:

$$\begin{aligned} c_{j,l}(\underline{k}) &= \beta_{j,l}(\underline{k},O) d_0(l-j,l) \\ d_{j,l}(\underline{k}) &= \beta_{j,l}(\underline{k},O)(d_1(l-j,l) - d_0(l-j,l)) \\ a(\underline{k}) &= 1 - d_0(0,h)\alpha(\underline{k},C) \\ b(\underline{k}) &= d_1(0,h)\alpha(\underline{k},C) \end{aligned}$$

D. Proof of lemma 3

We denote by $\Omega^k_{A,n}$ the set of the k-length sequences $\underline{w}$ that the algorithm A can perform in n steps. Let $1_{\underline{w}}$ the characteristic function of $\underline{w}$: $1_{\underline{w}}(v) = 1$ if $v \in \underline{w}$ and $1_{\underline{w}}(v) = 0$ otherwise. Note that $\sum_{v \in G} 1_{\underline{w}}(v) = k$ for all $\underline{w} \in \Omega^k_{A,n}$. Then,

$$\mathbb{P}_A(K_n = k) = \sum_{\underline{w} \in \Omega^k_{A,n}} \mathbb{P}_A(\underline{w}) = \sum_{\underline{w} \in \Omega^k_{A,n}} \frac{\sum_{v \in G} 1_{\underline{w}}(v)}{k} \mathbb{P}_A(\underline{w})$$
$$= \frac{1}{k} \sum_{v \in G} [\sum_{\underline{w} \in \Omega^k_{A,n}} 1_{\underline{w}}(v) \mathbb{P}_A(\underline{w})] = \frac{1}{k} \sum_{v \in G} \mathbb{P}_A(v | K_n = k).$$

which ends the proof.

E. Proof of lemma 3

By lemma 3, one has: $Cov(n) = \sum_{v \in G}^{h} \mathbb{P}_{RA}(v, n)$. So, all we need to show is the second equality which is a recursive expression of $\mathbb{P}_{RA}(i, n)$ meaning $\mathbb{P}^*_A$ and $\mathbb{P}_A$. In this expression the second term $\mathbb{P}_A(i, n)$ corresponds to the case where no repetition occurred during the time n, while the sum on $n1$ corresponds to the case of some re-initializations, such that the first one occurred at step n_1. Then, there is two possibilities : i was reached before step $n1$, which has a probability $\mathbb{P}^*_A(i, n_1)$ to occur, or i is not reached until n_1 and must be reached after in the $n - n_1$ remaining time, which leads to a probability $(1 - \mathbb{P}^*_A(i, n_1))\mathbb{P}_{RA}(i, n - n_1)$. This ends the proof.

F. Proof of lemma 4

Since belonging to $\{0, ..., L-1\}$, the x_i's are simply the L-ary decomposition coefficients of the integer q. These can be computed as following: for $i = d, ..., 1$, $x_i = E(q/L^i)$ and $q = q - x_i L^i$. This ends the proof.