Enhancing Random Walk State Space Exploration*

Radek Pelánek, Tomáš Hanžl, Ivana Černá, and Luboš Brim

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

Abstract. We study the behaviour of the random walk method in the context of model checking and its capacity to explore a state space. We describe the methodology we have used for observing the random walk and report on the results obtained. We also describe many possible enhancements of the random walk and study their behaviour and limits. Finally, we discuss some practically important but often neglected issues like counterexamples, coverage estimation, and setting of parameters. Similar methodology can be used for studying other state space exploration techniques like bit-state hashing, partial storage methods, or partial order reduction.

1 Introduction

In this work, we are concerned with verification of closed systems (i.e., systems given together with their environment). Verification of such system can be viewed as a search in the state space of the system for an error state. There are two basic approaches to the verification problem. *Testing* explores *some* paths through the state space; the selection is almost exclusively based on informal and heuristical methods or on a random choice. This approach is fast, has low memory requirements and is successful at finding obvious bugs. The disadvantages are that it is incomplete and it often misses corner case bugs. *Model checking* explores *all* paths through the state space. This approach can find corner case bugs and can guarantee the correctness of the system. The disadvantage is that it is very computationally expensive. In this work we try to combine advantages of both approaches: we take the random walk method (testing approach) and try to enhance it with some exhaustiveness (model checking approach). Particularly, we address the following issues:

– How successful is random walk at exploring state spaces? How large portion of the state space can be effectively explored by the basic random walk method? What is the behaviour of random walk on practical examples? Can it be theoretically explained and predicted?

* Supported by the Grant Agency of Czech Republic grant No. 201/03/0509

– How can we enhance random walk method by using additional memory? Should the available memory be used rather for local exhaustive searches or for caching already visited states? What are the other possible ways how to use the memory?

We use experimental approach to address these issues. We performed experiments on a large set of graphs corresponding to state spaces of real systems as well as on random and regular graphs. Our results are both positive and negative. On the positive side, we find that with the enhanced random walk it is feasible to visit most states in the state space with reasonable memory requirements (up to 20 times smaller than for classical exhaustive search). On the negative side, we find that the behaviour of random walk methods is very dependent on the specific state space, that it is very difficult to predict and that 100% state space coverage is not usually possible.

Related Work The random walk method was first applied to model checking by West [24] who demonstrated on a case study that the random walk could be a reasonable technique for finding bugs in real models. Recently, random walk has been used for verification in the model checker Lurch [18, 17]. Formal foundation for model checking by random walk were given by Grosu and Smolka [9].

There is an extensive *theoretical work* about random walks (in the mathematical setting a special case of Markov chains). However, most of the results concern undirected graphs whereas the state spaces encountered in model checking are represented as directed graphs. For directed graphs just pessimistic, exponential time bound on the expected coverage time, is known. There have been several attempts to restrict the class of models in order to guarantee the effectiveness of the random walk, e.g. Eulerian directed graph [10] and systems of symmetric dyadic flip flops [16]. Unfortunately, the resulting classes of models are very small and are not of practical interest in model checking.

Pure random walk does not use any memory at all working with an actual state only and does not store any information about previously visited parts of the state space. *Partial search* methods presented in [12, 15, 13, 22, 21] can be seen as enhancing the random walk by some additional memory. Other partial search methods are based on bit-state hashing [11] and on genetic manipulations [7].

The probabilistic approach is also employed by *partial storage* methods. These methods cover the whole state space and terminate. However, during the exploration only some states are stored reducing thus the overall memory requirements. Partial storage methods include state space caching [6, 23, 5], selective storing [2], and sweep line method [3].

Guided search combines the random exploration with the static analysis of the model. This approach has been used for guiding toward an error state in A^* search algorithm [14, 8, 4, 20] mainly.

A general experience based on all the above mentioned works is that there is no an universal solution in the framework of the random walk based partial methods. The right choice of a method and/or its parameters depends on the application and its specific properties. In addition, most of these papers propose a new single heuristic and demonstrate its potential on a small set of examples. The experimental results reported are neither explained nor the proposed method is compared to others.

The possible way how to make the random walk based partial search universally applicable is thus not to come up with "just another heuristic". What is really needed is a formation of a systematic framework for comparing existing methods accompanied with their exact evaluation on real-life models. The benefit of having such a sound basis should be a (semi-automatic or even automatic) method guiding the user in tuning the random walk based search for the given model.

Contributions In this work we try to make a first step toward the above stated goals. We thoroughly study the behaviour of the random walk method in model checking and its capacity to explore the state space. We describe the methodology used for comparing known heuristics and the obtained results. We also describe many possible enhancements of the random walk and study their behaviour and limits. Based on our experimental work we formulate guidelines for using the random walk method in model checking, state its limits, and detail what can and cannot be expected from the method. Finally, we discuss some practically important but often neglected issues like generating the counterexamples, coverage estimation, and setting of various parameters. Similar methodology can be used for other state space exploration techniques like bit-state hashing, partial storage methods, or partial order reduction.

2 Experimental Setting

The work presented in this paper relies on experiments. It contains observations based on results of measuring various characteristics related to the random walk technique, rather than formal analytical theorems and statements. Therefore, we start by describing the types of graphs that have been used in our experiments. The graphs can be grouped into the following three categories.

Random graphs Random graphs have been used quite often to demonstrate the behaviour of model checking algorithms and techniques. In [19] we have argued that graphs which occur in model checking applications have different structural properties than random graphs. Our experience is that the behaviour of the random walk on random graphs significantly differs from that on model checking graphs (see Section 3). Therefore we have used random graphs for comparisons only.

Regular graphs Regular graphs (e.g. grids, chains, circles) are also unsatisfactory as models of real-life systems. Nevertheless, regular graphs are quite suitable for understanding the behaviour of algorithms. In our experiments we have used manually constructed regular graphs for testing (and usually falsifying) hypothesizes about the behaviour of the random walk.

Model checking graphs Most of the experiments have been conducted on graphs originated from real-life state spaces. We have used a large set of graphs from our previous work [19]. These graphs have been attained from six explicit model checking tools. The list of all the models is given in Table 1, and all the graphs can be downloaded from [1]. These graphs do not contain any information about the model (neither atomic propositions in states nor labels on edges). We have used the model checking graphs to evaluate how much does the random walk depend on structural properties of graphs.

Moreover, we have also performed several experiments on graphs with nodes labeled by atomic propositions and action names added to the edges. These state spaces have been generated using our own explicit model checking tool DiVinE. The graphs have been used in experiments focused on the evaluation of the correspondence between the behaviour of the random walk and the properties of the models.

All the graphs used in experiments as well as details of measurements can be found on the web page `http://fi.muni.cz/~xpelanek/random_walk/`.

3 Pure Random Walk

In this section we consider the basic form of the random walk to perform the simple reachability task on a state space graph. The algorithm starts in the initial state of the graph. In each step it randomly chooses a successor of the current state and visits it. If the current state does not have any successors the algorithm re-starts from the initial state. The algorithm terminates when a target state is found or when some in advance given limit on the number of steps is exceeded. Similarly to other randomised algorithms, we always run the random walk several times to obtain expected behaviour.

From the theoretical point of view the most relevant characteristic of the random walk is the *covering time*, i.e. the expected number of steps after which all vertexes of the graph are visited. For undirected graphs the covering time is polynomial. For directed graphs the covering time can be exponential. For restricted classes of directed graphs, like Eulerian graphs or models of special protocols [16], the covering time is polynomial. These classes are too restrictive to be of any practical interest for model checking.

Our goal is to find out how the random walk behaves on graphs resulting from verification problems. Although the covering time is not really exponential in practise, it is still too high to be measured experimentally even for medium sized graphs (hundreds of states). For this reason we have measured the *coverage*, i.e. the ratio of vertexes which were visited after a given number of steps to all states. In order to get a deeper insight, we have investigated how various graph properties can influence the coverage. Here we summarise our observations. Unless stated otherwise, the observations relates to experiments on model checking graphs.

Coverage The coverage increases with the number of computation steps in a log-like fashion, i.e. at the beginning of the computation the number of newly visited states is high and it rapidly decreases with time. After a threshold point is reached the number of newly visited states drops nearly to zero. After this point it is meaningless to continue in the exploration. Our experience indicates that this happens when the number of steps is about ten times the size of the graph. This is the basic limit on the number of steps that we have used in our experiments.

Table 1 summarises the coverage achieved by the pure random walk on our set of model checking graphs. Note that the resulting coverage is very much graph dependent. In some cases the pure random walk can cover the whole graph, sometimes it covers less than 1% of states.

Correlation with graph properties In [19] we have studied typical structural properties of state spaces. A natural question is whether there is any correlation between the efficiency (coverage) of the random walk and these properties. For example, we have examined the relation between the coverage of the random walk an the number of strongly connected components, the average degree, the ratio of back level edges, and the frequency of diamonds.

We have found out that there is no straightforward correlation with any of these graph properties. The behaviour of the random walk is not determined by a single characteristic of the given graph but rather by an interplay of several of them. This means that it might not be possible to predict the efficiency of the random walk just from the knowledge of global properties of the state space. The intuition why this is so is illustrated in Fig. 1. The two graphs have similar global graph properties, but the efficiency of the random walk is very different. While the first graph is easily covered, the random walk will behave poorly on the second one. Note that graphs possessing the demonstrated properties occur naturally in model checking.

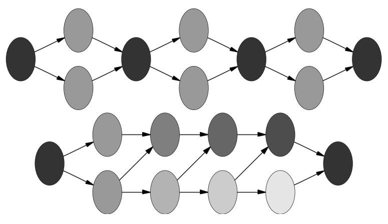

Fig. 1. Graphs with similar properties but different random walk coverage.

Another point we would like to stress is that using random graphs for testing specific random walk based model checking heuristics can be very misleading. Fig. 2 demonstrates the correlation between the average vertex degree and the random walk coverage both for random graphs and model checking graphs. There is a clear correlation for random graphs. For model checking graphs such a correlation has not been observed.

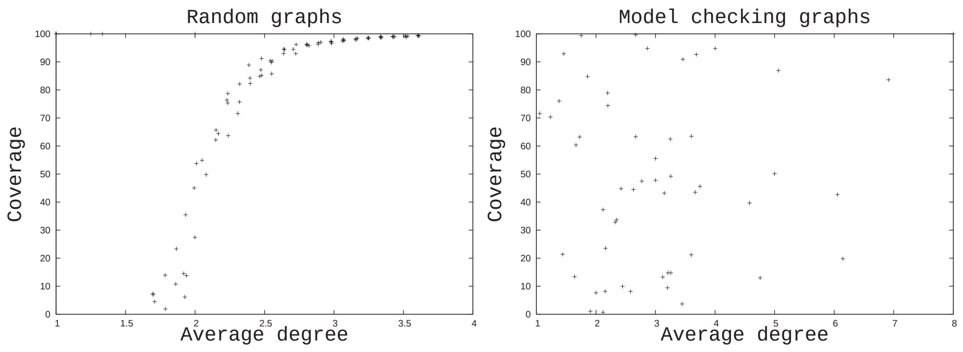

Fig. 2. Correlation between the average degree and coverage for random graphs and model checking graphs.

Distribution of visits Our next goal is to find out whether the probability of visiting a given state has an uniform distribution or whether some states are visited more frequently than the others. We have found out that the frequency of visits has the power law distribution. Thus the probability that a given state is visited is far from being uniform. This leads to the conclusion that the subgraph visited by the random walk cannot be considered to be a random sample of the whole graph!

We have tried to figure out reasons why some states are visited much more often than the others. Similarly to the global coverage, it turns out that there is no single reason. The following explanations come from our experiments.

– If the graph contains many deadlock states, then states with small depth (distance from the initial state) are frequently visited as the random walk returns to the initial state very often.
– If the random walk gets trapped in a small terminal strongly connected component it continues visiting states in this component only.
– Another scenario leading to frequent visits of states with small depth is the presence of many long back level edges.
– An uneven number of visits can be caused by the presence of diamond-like structures in the graph (see Fig. 3). For the random walk it is very unlikely to get into the corner of the diamond, but at the same time the probability

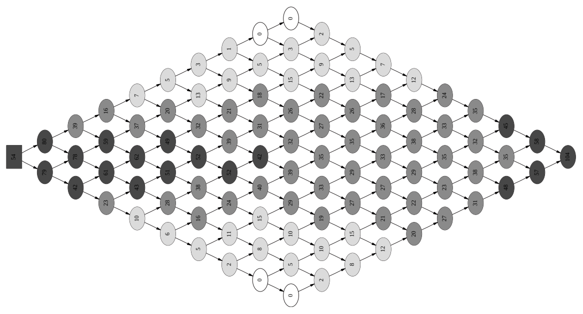

Fig. 3. Behaviour of the random walk on a diamond-like structure; darker vertices are visited more often

of visiting the middle states is high. The diamond-like structures are quite frequent in state spaces due to the interleaving semantics.

We conclude that the power law distribution of visits is a negative feature of the random walk. It means that the random walk spends most of the time repeatedly visiting just a few states. Several of the random walk enhancements presented in the next section try to improve on this.

4 How to Enhance the Random Walk?

In this section we describe several methods for improving the performance of the random walk. Generally, the enhancements make more effective use of memory and/or employ various heuristics to decide on the next direction of the exploration. Most of the methods have been presented previously, but usually in an ad hoc manner and without any rationale. We provide a systematic overview of these methods and give grounds for particular methods. Typically, the methods are intended to eliminate some of the negative features of the random walk method in model checking. We discuss experimental results and experiences as well.

4.1 Enhancement Methods

Re-initialisation Re-initialisation helps to avoid the situation when the random walk is getting trapped in a small terminal strongly connected component. To this end the computation is periodically stopped and the walk returns to (is re-initialised from) the initial state. The question is how to choose the number of computation steps after which the random walk should be re-initialised. If this limit is too small the algorithm returns to the initial state too often and redundantly revisits states with small depth. On the other hand, with a large re-initialisation limit we risk that the algorithm gets trapped. In situations where the limit cannot be derived from the model a randomly chosen limit performs better than a fixed one.

In order to avoid revisits of states with small depth one can use some of the available memory and store a set of states from which the algorithm will be re-initialised. The set can be for example computed as a frontier of a partial breadth-first search. After re-initialisation the algorithm is re-started from a randomly chosen state from the stored set.

Local Exhaustive Search Experiments with the model checking graphs provide an evidence that the number of visits of individual states during the random walk is distributed non-uniformly. To improve on this it may be useful to combine the random walk with a local exhaustive search. There are many possibilities how to implement the idea.

At first, we have to decide *when* to start a local search. The basic two possibilities are: after a predefined number of computation steps and after a randomly chosen number of steps (respecting a fixed probability distribution). Yet another possibility is to use a heuristic to determine a stage in the computation where the walk is near to a target state.

At second, we have to decide *how* to do the local search. We can use breadth-first search, depth-first search, or their clever combination. During the local search we either store the respective data structure (queue, stack), or we temporally store all visited states. After finishing the local search the random walk can either be re-initialised from the state where the local search has been started, or from a state saved in the respective data structure. The later possibility gives for example a higher chance to get into diamond corners.

Some of the ideas presented above have been employed by Sivaraj and Gopalakrishnan in [21] where the authors combine the random walk and the breadth-first local search in a distributed environment.

Caching Caching helps to avoid too frequent re-visits of individual states. Frequently visited states are stored in the cache with a high probability. Again, there are several issues to be considered here.

- How to manipulate the states in the cache? A state in the cache either can be revisited by the random walk with a smaller probability than the other states or cannot be revisited at all.
- How is the cache updated? There are two items to be decided: what is the strategy for selecting a state to be stored in and to be removed from the cache. The most straightforward way is to use a randomised management but it is also possible to make use of some heuristics.
- How is the cache implemented? The basic option is a standard hash table. Since the method is probabilistic anyway there is no need to solve collisions and a lossy compression to store states can be employed.

The caching method has been investigated mainly in connection with the full exploration [6, 5, 23]. It is used in situations where the available memory is not sufficient to store all states. Tronci et al. [22] use caching with partial search.

Pseudo-Parallel Walks The pure random walk search maintains only one currently visited state. Its performance can be increased if several random searches are performed simultaneously in an interleaving manner. In this case the method maintains an array of current states and iteratively selects their successors. This idea is closely related to the breadth-first search with a restricted size of queue (sometimes called beam search). Again, there are several issues to be decided here. Should individual random walks try to avoid each other e.g. with some kind of look-ahead? Are individual searches interleaved in a regular or random fashion?

Parallelisation of the random walk method has been examined in several papers under different names. Tronci et al. [22] combine caching with a breadth-first search with fixed sized queue. Sivaraj and Gopalakrishnan [21] combine parallel walks and breadth-first search. Groce and Visser [8] call their technique *beam search* and combine it with heuristics based on source code. Jones and Sorber [13] use parallel random walks enhanced with a biological motivated heuristic for verification of LTL properties.

Traces Traces provide yet another way how to enhance the random walk method via more effective usage of the available memory. The concept is to store not just the currently visited state but also the trace (path) from the initial state to the current state. Though the traces are primarily useful for reporting counterexamples, they can be used for effective search. With the help of traces the search can move both in forward and backward directions. This is useful for example for models with many deadlock states where instead of re-initialisation the search can just move one step backward and continue through another successor.

There are several possibilities how to store the traces during the search.

– The full trace is stored as a list of states.
– A fragment of the full trace (e.g. each k-th state from the full trace) is stored as a list of states.
– The full trace or its fragment are stored in a compressed way. The possibilities are to store list of actions, changes with respect to the predecessor, or the ordinal of the successor (for most state spaces the maximal out-degree is less than sixteen [19] and for these spaces it is sufficient to use four bits per state to record the ordinal).

The compressed representation increases the time needed for manipulating the trace, however it can extremely decrease the space requirements (in fact the size of a trace can be approximately the same as the size of the current state).

Guiding Guiding is a heuristic which helps to decide on the next direction of the exploration. The idea is to use the semantics of the model to prioritise some of the current state successors. This information can be used for guiding the search. It helps to

– select a successor to be visited next,

– decide when to do a local exhaustive search,
– decide when and what to store into the cache, and
– select a current state which successor is to be visited next (for parallel walks).

As usually, there are many ways how to gain the information from the model.

– Measure the code coverage (e.g. branch, state, path coverage) and prefer decisions leading to a higher coverage [8].
– For highly concurrent models try to maximise/minimise the number of process inter-leavings and the number of messages in buffers. Assign different probabilities to individual concurrent processes [4, 8].
– Estimate the distance of the currently visited state from the target state and use this estimation for decisions. This estimation can be computed by analysing components of the model [4, 14] or it can be approximated from the state space of a more abstract model [20].
– Alternatively, the user can provide some indications, e.g. by assigning fixed preferences to particular branches in the code.

The guiding technique has been frequently used for guiding the full search (A^* search).

4.2 Experiments

All of the above mentioned enhancements can be combined in a huge number of ways. A combination is determined by a choice of methods and allocation strategy (how to allocate the available memory among different objectives like local exhaustive search, cache, pseudo-parallel walks etc.).

It is clear that it is not feasible to perform exhaustive comparison of all potential combinations. For our experiments we have chosen combinations of methods and allocation strategies which seem to be intuitively plausible. Afterwards we have manually tuned some of the parameters. A complete list of measurements and results is available at `http://fi.muni.cz/~xpelanek/random_walk/`.

The main message gained from the experiments is that there is no superior enhancement of the random walk method. Each combination works well for different type of graphs. Sometimes it happens that the enhanced method, which uses relatively large amount of memory, performs worse then the pure random walk. For practical verification it is therefore very important not to stick to just one method!

Table 1 provides an overview of our observations. The table compares the coverage accomplished by the pure random walk with the best coverage we have been able to achieve using limited resources. The best coverage has been achieved for different graphs by different methods. The results reported in Table 1 have been obtained without any kind of guiding. Note that for most graphs it is feasible to cover more than 70% of states with memory consumption between 3% and 6% of the memory needed to perform the full search. We believe that it is not possible to get much further without very good guiding heuristics (which

Table 1. Resulting coverage after number of steps 10× size of the graph. For each model the table gives the coverage of pure random walk and the coverage of best method with memory consumption restricted to 3% (Best3), 6% (Best6), and 15% (Best15) of memory requirements of full search.

Model	Pure RW	Best3	Best6	Best15	Model	Pure RW	Best3	Best6	Best15
divine/farmer2	100.0	100.0	100.0	100.0	murphi/mcslock1	47.4	85.8	86.5	87.1
maso/pako	100.0	100.0	100.0	100.0	spin/petersonN	47.1	81.6	89.2	90.3
mcrl/chatbox	100.0	100.0	100.0	100.0	divine/abp	45.2	73.4	79.8	84.4
divine/shuffle-3x3	99.8	100.0	100.0	100.0	murphi/sci3	44.9	56.7	66.4	72.4
murphi/ns	99.4	99.4	99.5	99.7	cadp/site123medium	43.7	55.4	65.9	75.8
murphi/sort5	99.4	99.4	99.4	99.4	cadp/HAVi_asyn	43.4	48.8	60.2	70.0
mcrl/hef_wrong	99.2	99.3	99.4	99.4	spin/test2-rw6	43.0	56.9	100.0	100.0
divine/phil-basic-6	95.7	97.3	99.3	99.5	mcrl/onebit	42.9	97.2	98.0	99.2
spin/leader	93.5	98.9	99.6	99.6	spin/pftp.red	37.5	40.6	87.9	92.9
spin/sliding2	93.1	93.3	95.1	95.1	spin/test2-ring-5	35.0	76.1	88.2	91.6
cadp/bitalt	91.4	93.3	95.9	97.0	cadp/overtaking	34.6	88.1	91.7	91.7
murphi/cache4	87.7	90.7	92.6	94.2	divine/small	33.3	79.1	82.2	83.8
divine/bridge-4-5102025	86.8	91.1	93.7	95.6	spin/test2-abp	26.4	52.7	92.0	98.6
murphi/eadash	84.4	86.7	86.7	91.7	cadp/inv2d-p0-r2-a1	21.3	23.7	71.8	76.5
spin/snoopy.red	78.7	89.0	96.0	96.0	spin/n-s-original.red	19.7	93.1	99.8	99.8
cadp/scen1_orig_3	76.8	92.4	98.9	99.7	maso/puzzle50	16.5	48.6	78.4	80.5
murphi/peterson3	72.2	72.2	72.2	85.6	spin/smcs	14.7	62.7	73.2	90.9
spin/fgs	71.0	72.9	72.9	72.9	maso/elevator2	14.6	71.7	80.1	83.4
cadp/brp_protocol	70.6	75.6	82.3	84.3	spin/sort	13.3	72.3	82.6	92.9
spin/phil5	66.7	66.7	88.0	91.1	divine/machine	12.9	61.6	90.5	90.5
divine/el_fifo3	65.6	95.1	96.9	98.5	murphi/arbiter	10.4	35.8	48.9	62.0
mcrl/1394-fin	64.0	72.9	79.2	82.3	cadp/inres_protocol_int_6	10.2	54.7	72.4	73.5
divine/msmie-1-2	62.3	63.5	78.0	89.6	cadp/co4-3-1	9.0	67.7	75.3	87.7
divine/brp5	62.0	94.6	97.0	99.5	spin/mobile1	8.2	94.5	95.0	95.2
maso/abp	59.4	68.0	80.1	95.9	cadp/rel_rel	3.5	48.2	59.5	80.7
maso/ring5	51.0	57.2	63.9	85.9	spin/brp.red	1.0	81.2	88.7	88.7
cadp/cache	50.2	86.8	89.5	92.8	spin/cambridge00	0.6	9.5	21.3	26.4
mcrl/lift4-modif	49.9	89.2	89.2	89.2					

are difficult to compute automatically). Our experience is that once we try a few different methods we get quite close to the best coverage. Hence it seems that there is no need to try large number of combinations.

Although there is no dominant combination of methods and no universal way of choosing parameters, we can provide some general guidelines about how to partition the available memory among different methods. The good starting point is:

– 10% for re-initialization states
– 10% for pseudo-parallel walks
– 20% for cache
– 60% for local exhaustive search

5 Related Issues

In this section we address related issues concerning the practical applicability of the random walk based methods in model checking.

5.1 How to find a (short) counterexample?

The goal of the reachability analysis is to decide reachability of some of the target states. If a target state is reachable then the task is to find *a path* into it (so called counterexample). The methods discussed so far only decide the reachability of a target state. Since the diameters of the state spaces are typically small [19], the are short counterexamples and the random walk method can be used for their computations.

To find a counterexample one can use the trace technique, see Section 4. To find a short counterexample one can either use the local exhaustive search, start a new random walk, or tune the parameters of the searching procedure so that the states with a small depth are preferred.

Our experience indicates that in the case where an error can be detected by the random walk it is feasible to find a short counterexample by iterating the search several times.

5.2 How to estimate the coverage?

In a case when the random walk does not find any target state the the user cannot distinguish a correct model from an erroneous one. An estimate of the searched fraction of the state space could be of great value. However, this is very difficult to provide.

Tronci et al. [22] try to estimate the fraction of the visited states by saving random samples of the state space and by measuring the number of visits of the sample states. Though this routine works well on their few experiments, it is not a generally valid technique. The part of the state space used for the estimation is not a random sample, see Section 3. Based on an observation of the states visited by a random walk one cannot work out properties of the whole state space.

Grosu and Smolka [9] give a Monte Carlo algorithm for model checking which for given ϵ guarantee that the probability that an error will be found by further random walks is smaller than ϵ. But this does not mean that the probability of existence of an error is smaller than ϵ. This discrepancy does occur in real examples. Thus one may argue that the guarantee given by their Monte Carlo algorithm can be rather misleading.

Coverage metrics as encountered in the white-box testing, e.g. statement, branch, or path coverage, can be used for estimating the coverage. These metrics have well known disadvantages: on the one side 100% statement coverage does not imply 100% state space coverage, on the other side we can have 100% state space coverage even with statement coverage less than 100%. Nevertheless, the coverage metrcs can supply a useful information in practise.

5.3 How to choose a method and its parameters?

As we have already stated there is no superior method and combination of parameters. So the question is how to choose an appropriate method for a given application. Here we can provide two recommendations.

– Similar models have similar state spaces and on similar state spaces the methods have similar behaviour. It is meaningful to narrow the model (e.g. by abstraction or by setting smaller parameter values), generate its full state space, test different random walk methods on the narrowed state space, choose the one with the best behaviour and use the chosen method for the original model.
– Try several methods and hope.

6 Conclusions and Future Work

The paper provides an extensive overview of the random walk in model checking and its possible enhancements and studies the behaviour of both the random walk and its enhancements on realistic model checking examples.

Our reflections on the method are both positive and negative. On the positive side, we have found out that with the random walk it is feasible to visit most of the states in state spaces which are notably larger (up to 20 times) comparing to those than can be managed by classical full search. Since there is no need for communication, the random walk method can be performed in a distributed environment very effectively. The distribution multiplies the feasibility of the random walk by additional factor.

On the negative side, we indicate that the full 100% coverage is achievable (in a reasonable time) only for a few models. Moreover, we argue that in the case that the random walk fails to find an error it is not possible to provide an accurate estimation of the coverage.

The comparison of different methods clearly shows that none of them is superior. The choice of the best method is model-dependent.

Table 6 summarises the appropriateness of variants relative to the ratio of the available memory to the size of the searched state space.

M/S	Method	Coverage
≥ 1	full search, full storage	full coverage
$[0.1, 1)$	full search, partial storage	full coverage
$[0.01, 0.1)$	partial search	high coverage
< 0.1	partial search	low coverage

Table 2. Appropriateness of methods relative to the ratio of the available memory M to the size of the searched state space S.

Future work should aim at proposing of mechanisms for (semi)automatic selection of appropriate methods and/or their parameters for a given application. To this end even broader and more extensive case studies can be at hand. Yet another area deserving a deep inside is the application of the random walk method for the verification of more complex properties than just reachability (particularly the accepting cycle detection and LTL model checking).

References

1. `http://www.fi.muni.cz/~xpelanek/state_spaces`.
2. G. Behrmann, K.G. Larsen, and R. Pelánek. To store or not to store. In *Proc. Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, 2003.
3. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 450–464, 2001.
4. S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proc. SPIN workshop*, volume 2057 of *LNCS*, pages 57–79, 2001.
5. J. Geldenhuys. State caching reconsidered. In *SPIN Workshop*, volume 2989 of *LNCS*, pages 23–39, 2004.
6. P. Godefroid, G.J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. of Computer Aided Verification (CAV 1992)*, volume 663 of *LNVS*, pages 178–191, 1992.
7. P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 266–280, 2002.
8. A. Groce and W. Visser. Heuristics for model checking java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. to appear.
9. 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.
10. P. Haslum. Model checking by random walk. In *Proc. of ECSEL Workshop*, 1999.

11. G. J. Holzmann. An analysis of bitstate hashing. In *Proc. of Protocol Specification, Testing, and Verification*, pages 301–314, 1995.
12. G.J. Holzmann. Algorithms for automated protocol verification. *AT&T Technical Journal*, 69(2):32–44, February 1990.
13. M.D. Jones and J.Sorber. Parallel random walk search for LTL violations. In *Proc. of Parallel and Distributed Model Checking (PDMC 2002)*, volume 68 of *ENTCS*, pages 156–161, 2002.
14. 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.
15. F. Lin, P. Chu, and M. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *Computer Communication Review*, 17(5):126–134, 1987.
16. 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.
17. D. Owen and T. Menzies. Lurch: a lightweight alternative to model checking. In *Proc. of Software Engineering & Knowledge Engineering (SEKE'2003)*, pages 158–165, 2003.
18. D. Owen, T. Menzies, M. Heimdahl, and J. Gao. On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate. In *Proc. of IEEE/NASA Software Engineering Workshop (SEW'03)*, pages 75–81. IEEE, 2003.
19. R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22, 2004.
20. K Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2004)*, number 2988 in LNCS, pages 487–511, 2004.
21. H. Sivaraj and G. Gopalakrishnan. 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.
22. 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 (APSEC 2001)*, 2001.
23. E. Tronci, G. D. Penna, B. Intrigila, and M. V. Zilli. Exploiting transition locality in automatic verification. In *Proc. of Correct Hardware Design and Verification Methods (CHARME 2001)*, volume 2144, pages 259–274, 2001.
24. C. H. West. Protocol validation by random state exploration. In *International Symposium on Protocol Specification, Testing and Verification*, 1986.