Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Properties of State Spaces and Their Applications

Radek Pelánek[⋆]

Department of Information Technologies, Faculty of Informatics
Masaryk University Brno, Czech Republic
e-mail: pelanek@fi.muni.cz

Received: date / Revised version: date

Abstract. Explicit model checking algorithms explore the full state space of a system. State spaces are usually treated as directed graphs without any specific features. We gather a large collection of state spaces and extensively study their structural properties. Our results show that state spaces have several typical properties, i.e., they are not arbitrary graphs. We also demonstrate that state spaces differ significantly from random graphs and that different classes of models (application domains, academic vs industrial) have different properties. We discuss consequences of these results for model checking experiments and we point out how to exploit typical properties of state spaces in practical model checking algorithms.

1 Introduction

Model checking is an automatic method for formal verification of systems. In this paper we focus on explicit model checking which is the state-of-the-art approach to verification of asynchronous models (particularly protocols). This approach explicitly builds the full *state space* of the model (also called Kripke structure, occurrence or reachability graph). The state space represents all (reachable) states of the system and transitions among them. Verification algorithms use the state space to check specifications expressed in a temporal logic. The main obstacle of model checking is the state explosion problem — the size of the state space can grow exponentially with the size of the model description. Hence, model checking algorithms have to deal with extremely large graphs.

The classical model for large unstructured graphs is the *random graph* model of Erdős and Renyi [13] — every pair of vertices is connected with an edge by a given probability p. Large unstructured graphs are studied in many diverse areas such as social sciences (networks of acquaintances, scientist collaborations), biology (food webs, protein interaction networks), and computer science (Internet traffic, world wide web). Recent extensive studies of these graphs revealed that they share many common structural properties and that these properties significantly differ from properties of random graphs. This observation led to the development of more accurate models for complex graphs (e.g., 'small worlds' and 'scale-free networks' models) and to a better understanding of processes in these networks, e.g., spread of diseases and vulnerability of computer networks to attacks. See [1] for an overview of this research and further references.

1.1 Questions

In model checking, we usually treat state spaces as arbitrary graphs. However, since state spaces are generated from short descriptions, it is clear that they have some special properties. This line of thought leads to the following questions:

1. What are typical properties of state spaces? What do state spaces have in common?
2. Can state spaces be modeled by random graphs? Is it reasonable to use random graphs instead of state spaces for model checking experiments?
3. How can we apply properties of state spaces? Can we exploit these typical properties to traverse a state space more efficiently? Can some information about a state space be of any use to the user or to the developer of a model checker?
4. Are state spaces similar to such an extent that it does not matter which models we choose for benchmarking our algorithms? Is there any significant difference between toy academical models and real life

[⋆] Partially supported by GA ČR grant no. 201/07/P035.

case studies? Are there any differences between state spaces of models from different application domains?

In this paper we address these questions by an experimental study of a large number of state spaces of asynchronous systems.

1.2 Related Work

Many authors point out the importance of the study of models occurring in practice (e.g., [15]). But to the best of our knowledge, there has been no systematic work in this direction. In many articles one can find remarks and observation concerning typical values of individual parameters, e.g., diameter [7,37], back level edges [4,40], degree [18], stack depth [18]. Some authors make implicit assumptions about the structure of state spaces [10,24] or claim that the usefulness of their approach is based on characteristics of state spaces without actually identifying these characteristics [39]. Another line of work is concerned with visualization of large state spaces with the goal of providing the user with better insight into a model [17].

The paper follows on our previous research, particularly on [29,31–34]. The paper syntheses common topics of these works and present them in an uniform setting. The paper also presents several new observations (e.g., labels in state spaces, product graphs) and describes possible applications in more detail.

1.3 Organization of the Paper

Section 2 describes the benchmark set that we used to obtain experimental results reported in the paper. Section 3 introduces parameters of state spaces and presents results of measurements of these parameters over the benchmark set. Section 4 is concerned with parameters of state space traversal techniques (breadth-first search, depth-first search, and random walk). In Section 5 we compare properties of state spaces from different classes (application domains, industrial vs toy, models vs random graphs). Possible applications of all the reported results are discussed in Section 6. Finally, the last section provides answers to the questions raised above.

2 Background

In our previous study [29] we have used state spaces generated by six different model checkers. This study demonstrates that most parameters are independent of the specification language used for modeling and the tool used for generating a state space. The same protocols modeled in different languages yield very similar state spaces.

For this study we use models from our BEnchmark set for Explicit Model checkers (BEEM) [31]. Models in the set are implemented in a low-level modeling language based on communicating extended finite state machines (DVE language). Most of the models are well-known examples and case studies. Models span several different application areas (e.g., mutual exclusion algorithms, communication protocols, controllers, leader election algorithms, planning and scheduling, puzzles).

The benchmark set includes more than 50 parametrised models (300 concrete instances). For this study we use instances which have state space sizes smaller than 150,000 states (120 instances). We use only models of restricted size due to the high computational requirements of the performed analysis. However, our results show that properties of state space do not change significantly with the size of the state space.

The benchmark set is accompanied by an comprehensive web portal [31], which provides detailed information about all models. The web portal also includes detailed information about state spaces used in this paper. All the data about properites of analyzed state spaces are available for download (in XML format) and can be used for more detailed analysis.

The DVE modeling language is supported by an extensible model checking environment — The Distributed Verification Environment (DiVinE) [5]. We use the environment to perform all experiments reported in this paper. The benchmark set also contains (automatically generated) models in Promela, which can be used for independent experiments in the well-known model checker Spin [21].

3 State Space Parameters

A *state space* is a relational structure which represents the behavior of a system (program, protocol, chip, ...). It represents all possible states of the system and transitions between them. Thus we can view a state space as a simple directed graph $G = (V, E, v_0)$ with a set of vertices V, a set of directed edges $E \subseteq V \times V$, and a distinguished initial vertex v_0. Note that we use simple graphs, i.e., graphs without self-loops and multiple edges. This choice have a minor impact on some of the reported results (e.g., degrees of vertices), but it does not influence conclusions of the study. We also suppose that all vertices are reachable from the initial vertex. In the following we use *graph* when talking about generic notions and *state space* when talking about notions which are specific to state spaces of asynchronous models.

3.1 Degrees

Out-degree (*in-degree*) of a vertex is the number of edges leading from (to) this vertex. *Average degree* is the ratio $|E|/|V|$. The basic observation is that the average degree is very small – typically around 3 (Fig. 1). Maximal

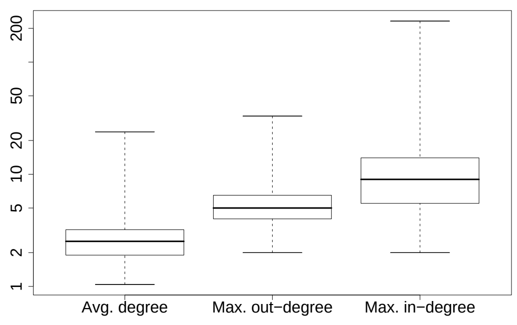


Fig. 1. Degree statistics. Values are displayed with the boxplot method. The upper and lower lines are maximum and minimum values, the middle line is a median, the other two are quartiles. Note the logarithmic y-axis.

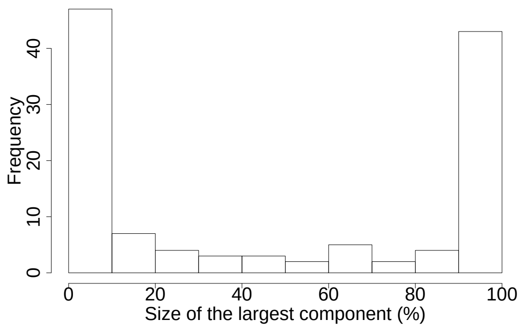


Fig. 2. Histogram of sizes of the largest SCC component in a state space.

in-degree and out-degree are often several times higher than the average degree but with respect to the size of the state space they are small as well. Hence state spaces do not contain any 'hubs'. In this respect state spaces are similar to random graphs, which have Poisson distribution of degrees. On the other hand, scale free networks discussed in the introduction are characterized by the power-law distribution of degrees and the existence of hubs is a typical feature of such networks [1].

The fact that state spaces are sparse is not surprising and was observed long ago — Holzmann [18] gives an estimate 2 for average degree. It can be quite easily explained: the degree corresponds to a 'branching factor' of a state; the branching is due to parallel components of the model and to the inner nondeterminism of components; and both of these are usually rather small. In fact, it seems reasonable to claim that in practice $|E| \in O(|V|)$. Nevertheless, the sparseness is usually not taken into account either in the construction of model checking algorithms or in the analysis of their complexity.

3.2 Strongly Connected Components

A *strongly connected component* (SCC) of G is a maximal set of states $C \subseteq V$ such that for each $u, v \in C$, the vertex v is reachable from u and vice versa. The *quotient graph* of G is a graph (W, H) such that W is the set of SCCs of G and $(C_1, C_2) \in H$ if and only if $C_1 \neq C_2$ and there exist $r \in C_1, s \in C_2$ such that $(r, s) \in E$. The *SCC quotient height* of the graph G is the length of the longest path in the quotient graph of G. Finally, a component is *terminal* if it has no successor in the quotient graph.

For state spaces, the height of the SCC quotient graph is small. In all but one case it is smaller than 200, in 70% of cases it is smaller than 50.

There is an interesting dichotomy with respect to the structure of strongly connected components, particularly concerning the size of the largest SCC (see Fig. 2). A state space either contains one large SCC, which includes nearly all states, or there are only small SCCs. The largest component is usually terminal and often it is even the only terminal.

3.3 Labels

So far we have considered state spaces as plain directed graphs. However, state spaces do not have 'anonymous' edges and states:

– Vertices are state vectors which consist of variable valuations and process program counter values.
– Edges are labelled by actions which correspond to actions of the model.

Distribution of edge labels is far from uniform. Typically there are few labels which occur very often in a state space, whereas most labels occur only in small numbers. More specifically, for most models the most often occurring label appears on approximately 6% of all edges, the five most often occurring labels appears on approximately 20% of all edges. This result does not depend on number of labels, i.e., the 20% ratio taken by the five most common labels holds approximately for both small models with thirty different labels as well as for realistic models with hundreds of different labels.

State vectors can be divided into parts which correspond to individual processes in the model (i.e., program counter of the process and valuation of local variables). The number of distinct valuations of these local parts is small, in most cases smaller then 255, which means that the state of each process can be stored in 1 byte. Moreover, the distribution of different valuations is again non-uniform, i.e., some valuations of the local part occur in most states (typically valuations with repeated value 0), whereas other valuations occur only in few states.

The number of differences in state vectors of two adjacent vertices is small, typically the action changes the state vector in 1 to 4 places. Distribution of these

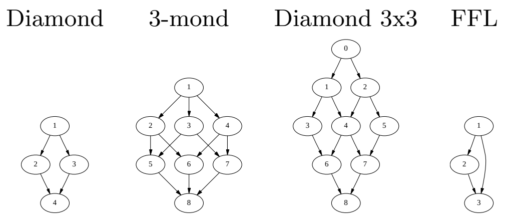


Fig. 3. Illustrations of motifs

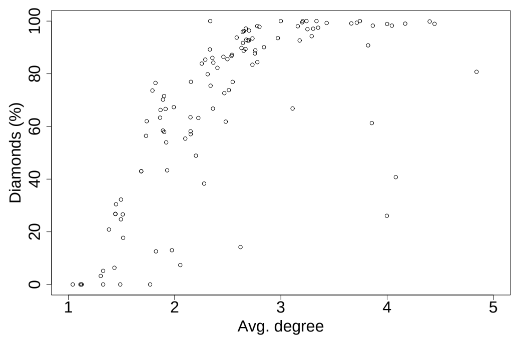


Fig. 4. Relationship between occurrence of diamonds and the average degree. The occurrence of diamonds is reported as a ratio of the number of states which are roots of some diamond to all states.

changes is again non-uniform. This is not surprising since changes in the state vector are caused by (non-uniformly distributed) labels.

For more details see the BEEM webpage [31], which contains specific results for each model.

3.4 Local Structure and Motifs

As the next step we analyze the local structure of state spaces. In order to do so, we employ some ideas from the analysis of complex networks. A typical characteristic of social networks is *clustering* — two friends of one person are friends together with much higher probability than two randomly picked persons. Thus vertices have a tendency to form clusters. This is a significant feature which distinguishes social networks from random graphs.

In state spaces we can expect some form of clustering as well — two successors of a state are more probable to have some close common successor than two randomly picked states. Specifically, state spaces are well-known to contain many 'diamonds'. We try to formalize these ideas and provide some experimental base for them.

The *k-neighborhood* of v is a subgraph induced by a set of vertices with the distance from v smaller or equal to k. The *k-clustering coefficient* of a vertex v is the ratio of the number of edges to the number of vertices in the k-neighborhood (not counting v itself). If the clustering coefficient is equal to 1, no vertex in the neighborhood has two incoming edges within this neighborhood. A higher coefficient implies that there are several paths to some vertices within the neighborhood. For state spaces, the clustering coefficient linearly increases with the average degree. Most random graphs have clustering coefficients close to 1.

Another inspiration from complex networks are so-called 'network motifs' [28, 27]. Motifs are studied mainly in biological networks and are used to explain functions of network's components (e.g., function of individual proteins) and to study evolution of networks.

We have systematically studied motifs in state spaces. We find the following motifs to be of specific interest either because of abundant presence or because of total absence in many state spaces:

– *Diamonds* (we have studied several variations of structures similar to diamond, see Fig. 3). Diamonds are well known to be present in state spaces of asynchronous concurrent systems due to the interleaving semantics. Diamonds display an interesting dependence on the average degree (Fig. 4). There is a rather sharp boundary for value 2 of the average degree: for a state space with average degree less than two there is a small number of diamonds, for state spaces with average degree larger than two there are a lot of them.
– *Chains* of states with just one successor. We have measured occurrences of chains of length 3, 4, 5. Chains occur particularly in state spaces with average degree less than two (i.e., their occurrence is complementary to diamonds).
– *Short cycles* of lengths 2, 3, 4, 5. Short cycles are nearly absent in most state spaces.
– *Feed forward loop* (see Fig. 3). This motif is a typical for networks derived from biological systems [28]; in state spaces it is very rare.

The bottom line of these observations is that the local structure depends very much on the average degree. If the average degree is small, then the local structure of the state space is tree-like (without diamonds and short cycles, with many chains of states of degree one). With the high average degree, the state space has many diamonds and high clustering coefficient.

4 Properties of Search Techniques

In verification, the basic operation is the traversal of a state space. Therefore, it is important to study not only 'static' parameters of state spaces but also their 'dynamics', i.e., properties of search techniques. Here we consider three basic techniques for state space traversal and their properties.

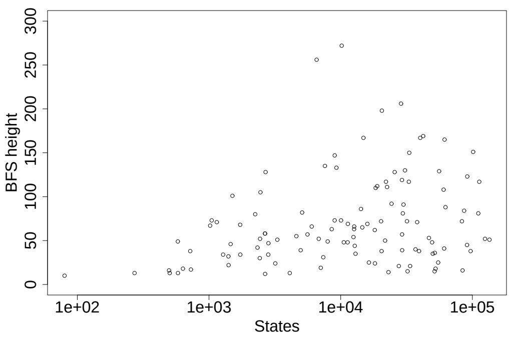


Fig. 5. The BFS height plotted against the size of the state space. Note the logarithmic x-axis. Three examples have BFS height larger than 300.

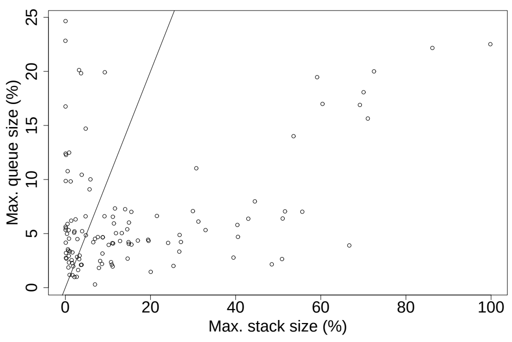


Fig. 7. A comparison of maximal queue and stack sizes expressed as percentages of the state space size.

4.1 Breadth-First Search (BFS)

Let us consider BFS from the initial vertex v_0. A BFS *level* with an index k is a non-empty set of states with minimal distance from v_0 equal to k. The *BFS height* is the largest index of a level. An edge (u, v) is a *back level edge* if v belongs to a level with a lower or the same index as u. The *length* of a back level edge is the difference between the indices of the two levels.

In our benchmarks, the BFS height is small (Fig. 5). There is no clear correlation between the state space size and the BFS height; it depends rather on the type of the model.

The sizes of levels follow a typical pattern. If we plot the number of states on a level against the index of a level we get a *BFS level graph*[1]. See Fig. 6. for several examples of BFS level graphs. Note that in all cases the graph has a 'bell-like' shape.

The ratio of back level edges to all edges in a state space varies between 0% and 50%; the ratios are uniformly distributed in this interval. Most edges are short — they connect two close levels (as already observed by Tronci et al. [40]). However, for most models there exist some long back level edges.

4.2 Depth-First Search (DFS)

Next we consider the DFS from the initial vertex. The behavior of DFS (but not the completeness) depends on the order in which successors of each vertex are visited. Therefore we have considered several runs of DFS with different orderings of successors.

If we plot the size of the stack during DFS we get a *stack graph*. Fig. 6. shows several stack graphs; for more graphs see [31]. The interesting observation is that the shape of the graph does not depend much on the ordering of successors. The stack graph changes a bit of course, but the overall appearance remains the same. This suggests, that DFS is rather 'stable' with respect to the ordering of successors. Each state space, however, has its own typical stack graph; compare to BFS level graphs, which all have more or less bell-like shape.

For implementations of the breadth- and depth-first search one uses queue and stack data structures. Fig. 7. compares the maximal size of a queue and a stack during the traversal. The maximal size of a stack is smaller then maximal size of a queue in 60% of cases, but the relative size of a queue is always smaller than 25% of the state space size whereas the relative size of a stack can go up to 100% of the state space size. These results have implications for practical implementation of model checking tools (see Section 6).

4.3 Random Walk

Finally, we consider a simple random walk technique. The technique 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 search also uses periodic re-start in order to avoid the situation when the random walk gets trapped in a small terminal strongly connected component.

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 vertices of the graph are visited. For undirected graphs the covering time is polynomial. For directed graphs the covering time can be exponential. Even in those cases when it is not exponential, 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 vertices which were visited after a given number of steps.

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

[1] Note that the word 'graph' is overloaded here. In this context we mean graph in the functional sense.

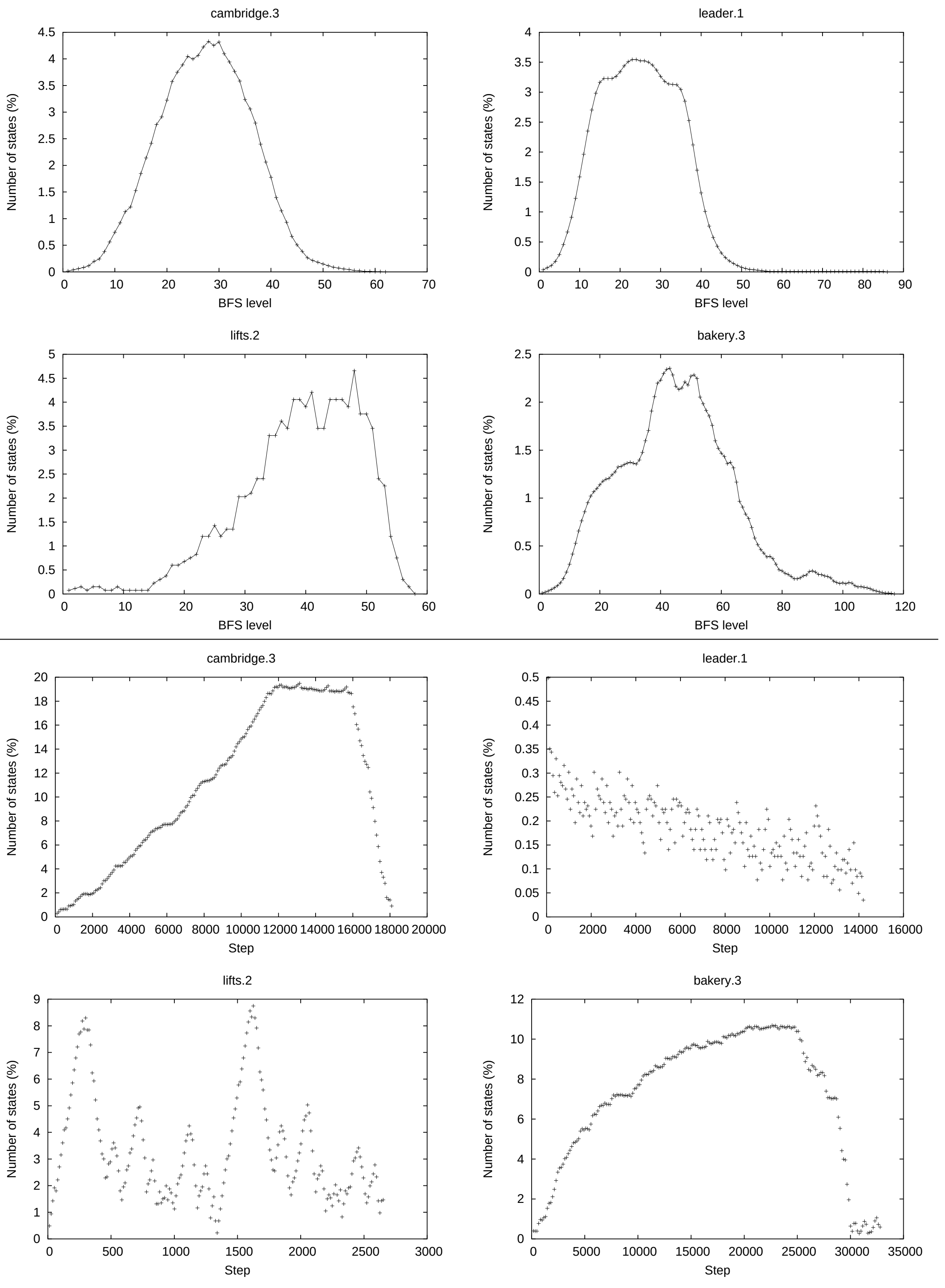


Fig. 6. BFS level graphs (first four) and stack graphs (second four).

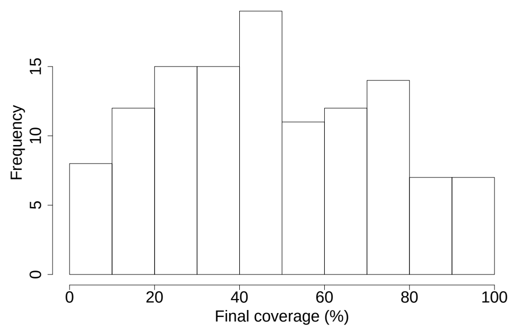


Fig. 8. Histogram of random walk coverage after number of steps equal to 10 times the size of the state space. Frequency means the number of state spaces for which the final coverage was in the given interval.

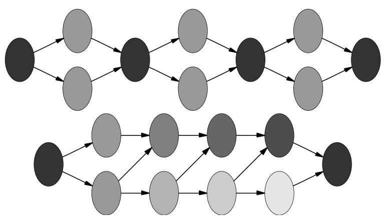


Fig. 9. Graphs with similar properties but different random walk coverage. The color correspond to probability of a visit by a random walk; darker vertices have higher change of a visit.

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. Fig. 8 gives the coverage after this limit. Note that the resulting coverage is very much graph-dependent. In some cases the random walk can cover the whole graph, whereas sometimes it covers less than 3% of states.

A natural question is whether there is any correlation between the efficiency (coverage) of the random walk and structural properties of a state space. Unfortunately, it seems that there is no straightforward correlation with any of the above studied graph properties. The intuition for this negative result is provided by Fig. 9. The two displayed 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 of these types occur naturally in model checking.

Finally, we measure the probability of visiting individual states in order to find whether the probability of a visit has a uniform distribution or whether some states are visited more frequently than others. We find 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!

5 Comparisons

In this section we compare properties of state spaces of models from different classes.

5.1 Application Domains

We have classified models according to their application domains and studied the parameters of each class. State spaces from each domain have some distinct characteristics; see [31] for description of the classification and [32] for more specific results.

– Mutual exclusion algorithms: state spaces usually contain one large strongly connected component and contain many diamonds.
– Communication protocols: state spaces are not acyclic, have a large BFS height and long back level edges, usually contain many diamonds.
– Leader election algorithms: state spaces are acyclic and contain diamonds.
– Controllers: state spaces have small average degree, a large BFS height and long back level edges, usually contains many diamonds.
– Scheduling, planning, puzzles: state spaces are often acyclic, with a very small BFS height, large average degree, many short back level edges; state space are without prevalence of diamonds or chains.

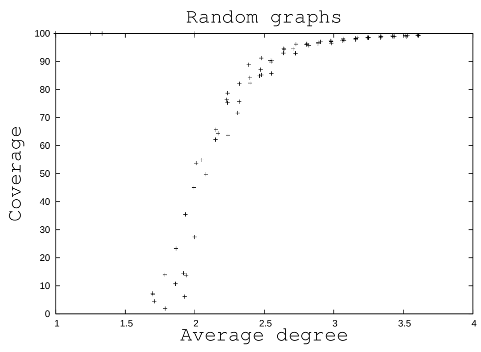
We expect that similar distinct characteristic exists for other application domains as well.

5.2 Random Graphs

Let us compare properties of state spaces and properties of random graphs, which are often used in experiments with model checking algorithms. We use the classical Erdős-Renyi model of a random graph [13].

Although distances (BFS height, diameter) in state spaces are small, distances in random graphs are even smaller. For most state spaces we observe that there are only a few typical lengths of back level edges and a few typical lengths of cycles (this is caused by the fact that back level edges correspond to specific actions in a model). However, random graphs have no such feature.

State spaces are characterized by the presence (respectively absence) of specific motifs, particularly diamonds (respectively short cycles). More generally, state spaces shows significant clustering and the size of k-neighborhood grows (relatively) slowly. Random graphs do not have clustering and the size of k-neighborhood grows quickly.



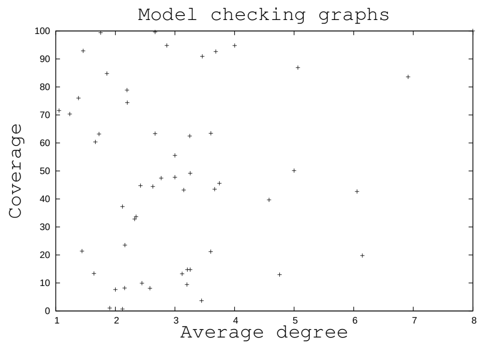


Fig. 10. Correlation between the average degree and random walk coverage for random graphs and model checking graphs.

If we plot the size of the queue (stack) during BFS (DFS) (as done in Fig 6) then we obtain for each state space a specific graph, which is usually at least a bit ragged and irregular. In contrast, for most random graphs we obtain very similar, smooth graphs.

Finally, we provide a specific example which demonstrates how the use of random graphs can obfuscate experimental analysis. Fig. 10 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. If we did the experiments only with random graphs, we could be misled into wrong conclusions about the effectiveness and applicability of random walk technique.

5.3 Toy versus Industrial Examples

We have manually classified examples into three categories: toy, simple, and complex. The major criterion for the classification was the length of the model description. State spaces sizes are similar for all three categories, because for toy models we use larger values of model parameters (as is usual in model checking experiments).

The comparison shows differences in most parameters. Here we only briefly summarize the main trends; more detailed figures can be found on the Beem web page [31].

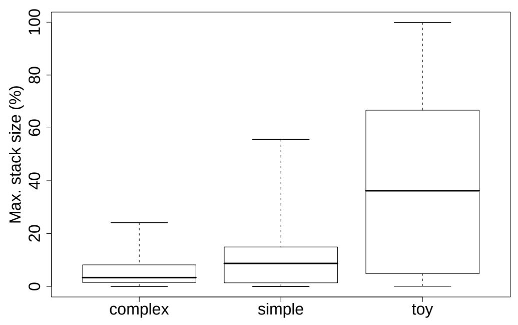


Fig. 11. The maximal stack size (given in percents of the state space size) during DFS. Results are displayed with the boxplot method (see Fig. 1 for explanation).

– The maximal size of the stack during DFS is significantly shorter for complex models (Fig. 11).
– The BFS height is larger for state spaces of complex models. The number of back level edges is smaller for state spaces of complex models but they have longer back level edges.
– The average degree is smaller for state spaces of complex models. Since the average degree has a strong correlation with the local structure of the state space (see Section 3.4), this means that also the local structure of complex and toy models differs.
– Generally, the structure is more regular for state spaces of toy models. This is demonstrated by BFS level graphs and stack graphs which are smoother for state spaces of toy models.

These results stress the importance of having complex case studies in model checking benchmarks. Particularly experiments comparing explicit and symbolic methods are often done on toy examples. Since toy examples have more regular state spaces, they can be more easily represented symbolically.

5.4 Product Graphs

During the verification of temporal properties, algorithms often work with the ‘augmented state space’ rather then directly with the state space. Particularly, the verification of linear temporal logic is based on the construction of so-called product graph: a negation of a temporal logic formulae is transformed into an equivalent Büchi automaton, a product of a state space and the automaton is computed, and the product graph is searched for accepting cycles [41]. What are the properties of product graphs? Is there any significant difference from properties of plain state spaces?

The Beem benchmark [31] also contains temporal properties. We have used these properties to construct product graphs and we have studied their properties. Our experiments indicate that the structure of product graphs is very similar to structure of plain state spaces. Since the results are so similar, we do not provide explicit results and figures. The only difference worth mentioning is that the height of the SCC quotient graphs is slightly larger for product graphs, but it is still rather small.

6 Applications

In previous sections we outlined many interesting properties of state spaces. Are these properties just an interesting curiosity? Or can we exploit them in the verification process? In this section we outline several possible applications of described properties.

6.1 Algorithm Tuning

Knowledge of typical properties of state spaces can be useful for tuning the performance of model checking algorithms.

In Section 4 we demonstrate that the size of a queue (stack) during the state space search can be quite large, i.e., it may happen that the applicability of a model checker becomes limited by the size of a queue (stack) data structure. Therefore, it is important to pay attention to these structures when engineering a model checker. This is already done in some model checkers – SPIN can store part of a stack on disc [20], UPPAAL stores all states in the hash table and maintains only references in a queue (stack) [12].

BFS parameters (particularly BFS height and sizes of BFS levels) can be used to set parameters of algorithms appropriately: algorithms that exploit magnetic disk often work with individual BFS levels [38]; random walk search [33] and bounded search [23] need to estimate the height of the state space; techniques using stratified caching [16] and selective storing of states [6] can also take the shape of the state space into account.

The local structure of a state space (e.g., presence or absence of diamonds) can also be used for tuning parameter values, particularly for techniques which employ local search, e.g., random walk enhancements [33, 36], sibling caching and children lookahead in distributed computation [25], or heuristic search.

Typical motifs and state vector characteristics (number of local states, number of changes in state vector) can be employed for efficient storage of states (e..g, state compression [19]). The fact that distribution of edge labels is not uniform is important for selection of a covering set of transitions, which can be used for partial order reduction or selective storing [6].

6.2 Automation of Verification

Any self-respecting model checker has a large number of options and parameters which can significantly influence the run-time of verification. In order to verify any reasonable system, it is necessary to set these parameters properly. This can be done only by an expert user and it requires lot of time. Therefore, it is desirable to develop methods for automatic selection of techniques and parameter values. We discuss in detail two concrete examples.

6.2.1 Memory Reduction Techniques

The main obstacle to model checking is memory requirements. Researchers have developed a large number of memory reduction techniques which aim at alleviating this problem. Most of these techniques introduce time/memory trade-offs. Each of these techniques has specific advantages and disadvantages and is suitable only for some type of models (state spaces). State space parameters can be employed for the selection of a suitable technique; in the following we outline several specific examples.

The sweep line technique [11] deletes from memory states that will never be visited again. This technique is useful only for models with acyclic state spaces or with small SCCs. This technique also requires short back level edges. The same requirement holds for caching based on transition locality [40].

For acyclic state spaces it is possible to use specialized algorithms, e.g., dynamic partial order reduction [14] or a specialized bisimulation based reduction [30, p. 43-47].

For state spaces with many diamonds it is reasonable to employ partial order reduction, whereas for state spaces without diamonds this reduction is unlikely to yield significant improvement. On the other hand, selective storing of states [6] can lead to good memory reduction for state spaces with many chains.

The heuristic algorithm based on bayesian meta heuristic [35] works well for models with high average degree (greater than 10). This fact calls into question the applicability of the approach to industrial models (see Section 5.3). On the other hand, the IO-efficient algorithm for model checking [2] works better for models with small vertex degrees.

6.2.2 Cycle Detection Algorithms

Cycle detection algorithms are used for LTL verification. Currently, there is a large number of different cycle detection algorithms, particularly if we consider distributed algorithms for networks of workstations [3]. Analysis of state space parameters can be helpful for an automatic selection of a suitable algorithm.

For example, a distributed algorithm based on localization of cycles [24] is suitable only for state spaces with small SCCs (which are, unfortunately, not very common). Similarly, the classical depth-first search based algorithm [22] can be reasonably applied only for state spaces with small SCCs, because for state spaces with large SCCs it tends to produce very long counterexamples (long counterexamples are not very useful in practice). On the other hand, the explicit one-way-catch-them-young algorithm [9] has complexity $O(nh)$, where n is the number of states and h is the height of the SCC quotient graph, i.e., this algorithm is more suitable for state space with one large component. The complexity of BFS-based distributed cycle detection algorithm [4] is proportional to the number of back level edges.

6.3 Estimation of State Space Size

The typical pattern of the BFS level graph (see Section 4.1) can be used for estimating the number of reachable states. Such an estimate has several applications: it can be used to set verification parameters (e.g., size of a hash table, number of workstations in a distributed computation) and it is also valuable information for the user of the model checker — at least, users are more willing to wait if they are informed about the remaining time [26].

We outline a simple experiment with state space size estimation based on BFS levels. We generate a sample consisting of the first k BFS levels. Then we estimate how many times the number of reachable states is larger than the size of the sample. More specifically, we do just an order of magnitude estimate. Let R be the ratio of the total number of reachable states to the size of the sample. We use the following three classes for estimates: class 1 ($1 \leq R < 4$), class 2 ($4 \leq R < 32$), class 3 ($32 \leq R$).

We use three techniques for estimating the classification: human, classification tree [8] and a neural networks. All techniques are trained on a training set and then evaluated using a different testing set. All three techniques achieve similar results — the success rate is about 55%, with only about 3% being major mistake (class 1 classified as class 3 or vice versa). These results can be further improved by a combination with other estimation techniques and by using domain specific information. See [34] for more details about this experiment and for description of several other techniques for estimating state space parameters.

7 Answers

Finally, we provide answers to questions that were raised in the introduction and we discuss directions for the future work. Although we have done our measurements on a restricted sample of state spaces, we believe that it is possible to draw general conclusions from the results. Results of measurements are consistent — there are no significant exceptions from reported observations.

What are typical properties of state spaces?

State spaces are usually sparse, without hubs, with one large SCC, with small diameter and small SCC quotient height, with many diamond-like structures.

These properties can not be explained theoretically. It is not difficult to construct artificial models without these features. This means that observed properties of state spaces are not the result of the way state spaces are generated nor of some features of specification languages but rather of the way humans design/model systems.

Can state spaces be modeled by random graphs?

In Section 5.2 we have discussed many properties in which state spaces differ from random graphs. Unfortunately, random graphs are often used for experiments with model checking algorithms. We conclude that random graphs have significantly different structure than state spaces and thus that this practice can lead to wrong conclusions (see Section 5.2 for a specific example). Thou shalt not do experiments on random graphs.

Are state spaces similar to such an extent that it does not matter which models we choose for benchmarking our algorithms?

Although state spaces share some properties in common, some can significantly differ. Behavior of some algorithms can be very dependent on the structure of the state space. This is clearly demonstrated by experiments with random walk. For some graphs one can quickly cover 90% of the state space by random walk, whereas for other we were not able to get beyond 3%. So it is really important to test algorithms on a large number of models before one draws any conclusions.

Particularly, there is a significant difference between state spaces corresponding to complex and toy models. Moreover, we have pointed out that state spaces of similar models are very similar. We conclude that it is not adequate to perform experiments just on few instances of some toy example. Thou shalt not do experiments (only) on Philosophers.

How can we apply properties of state spaces?

Typical properties can be useful in many different ways. In Section 6 we discuss two broad types of applications:

– Tuning of model checking algorithm, i.e., using the knowledge of typical properties to improve the performance of model checking algorithms.
– Automation of verification, i.e., using the knowledge of parameter values to choose a suitable verification technique or algorithm.

We outline many specific examples of applications and we believe that there are (potentially) many more. Moreover, we outlined also one untypical application — estimation of the state space size based on the typical behaviour of BFS.

Acknowledgment

I thank Ivana Černá, Pavel Krčál, and Pavel Šimeček for valuable discussions and comments on this project. I also thank anonymous reviewers for their comments on the first version of this paper.

References

1. R. Albert and A. L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, 2002.
2. T. Bao and M Jones. Time-efficient model checking with magnetic disk. In *Proc. of Tools and Algorithms for the Construction and Analysis (TACAS'05)*, volume 3440 of *LNCS*, pages 526–540. Springer, 2005.
3. J. Barnat, L. Brim, and I Cerná. Cluster-based ltl model checking of large systems. In *Proc. of Formal Methods for Components and Objects (FMCO'05), Revised Lectures*, volume 4111 of *LNCS*, pages 259–279. Springer, 2006.
4. J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *Proc. Automated Software Engineering (ASE 2003)*, pages 106–115. IEEE Computer Society, 2003.
5. J. Barnat, L. Brim, I. Černá, P. Moravec, P. Rockai, and P. Šimeček. Divine - a tool for distributed verification. In *Proc. of Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006. The tool is available at `http://anna.fi.muni.cz/divine`.
6. G. Behrmann, K.G. Larsen, and R. Pelánek. To store or not to store. In *Proc. Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 433–445, 2003.
7. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1999)*, volume 1579 of *LNCS*, pages 193–207, 1999.
8. L. Breiman. *Classification and Regression Trees.* CRC Press, 1984.
9. I. Černá and R. Pelánek. Distributed explicit fair cycle detection. In *Proc. SPIN workshop*, volume 2648 of *LNCS*, pages 49–73, 2003.
10. A. Cheng, S. Christensen, and K. H. Mortensen. Model checking coloured petri nets exploiting strongly connected components. In *Proc. International Workshop on Discrete Event Systems*, pages 169–177, 1996.
11. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 450–464, 2001.
12. A. David, G. Behrmann, K. G. Larsen, and W. Yi. Unification & sharing in timed automata verification. In *Proc. SPIN Workshop*, volume 2648 of *LNCS*, pages 225–229, 2003.
13. P. Erdős and A. Renyi. On random graphs. *Publ. Math.*, 6:290–297, 1959.
14. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of Principles of programming languages (POPL'05)*, pages 110–121. ACM Press, 2005.
15. M. B. Dwyer G. S. Avrunin, J. C. Corbett. Benchmarking finite-state verifiers. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):317–320, 2000.
16. J. Geldenhuys. State caching reconsidered. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 23–39. Springer, 2004.
17. J. F. Groote and F. van Ham. Large state space visualization. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, pages 585–590, 2003.
18. G. J. Holzmann. Algorithms for automated protocol verification. *AT&T Technical Journal*, 69(2):32–44, 1990.
19. G. J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proc. SPIN Workshop.* Twente Univ., 1997.
20. G. J. Holzmann. The engineering of a model checker: the gnu i-protocol case study revisited. In *Proc. SPIN Workshop*, volume 1680 of *LNCS*, pages 232–244, 1999.
21. G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, 2003.
22. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.
23. P. Krčál. Distributed explicit bounded ltl model checking. In *Proc. of Parallel and Distributed Methods in verifiCation (PDMC'03)*, volume 89 of *ENTCS*. Elsevier, 2003.
24. A. L. Lafuente. Simplified distributed LTL model checking by localizing cycles. Technical Report 176, Institut für Informatik, Universität Freiburg, July 2002.
25. F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Proc. of SPIN Workshop*, volume 2057 of *LNCS*, pages 80–102. Springer, 2001.
26. D. H. Maister. The psychology of waiting lines. In J. A. Czepiel, M. R. Solomon, and C. Suprenant, editors, *The Service Encounter.* Lexington Books, 1985.
27. R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon. Superfamilies of evolved and designed networks. *Science*, 303(5663):1538–1542, March 2004.
28. R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.
29. R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.
30. R. Pelánek. *Reduction and Abstraction Techniques for Model Checking.* PhD thesis, Faculty of Informatics, Masaryk University, Brno, 2006.
31. R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
32. R. Pelánek. Model classifications and automated verification. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'07)*, 2007. To appear.
33. R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'05)*, pages 98–105. ACM Press, 2005.

34. R. Pelánek and P. Šimeček. Estimating state space parameters. Technical Report FIMU-RS-2008-01, Masaryk University Brno, 2008.
35. K. Seppi, M. Jones, and P. Lamborn. Guided model checking with a bayesian meta-heuristic. In *Proc. of Application of Concurrency to System Design (ACSD'04)*, page 217. IEEE Computer Society, 2004.
36. 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.
37. U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration.* PhD thesis, Technical University of Munich, 1997.
38. U. Stern and D. L. Dill. Using magnatic disk instead of main memory in the Murphi verifier. In *Proc. Computer Aided Verification (CAV 1998)*, volume 1427 of *LNCS*, pages 172–183, 1998.
39. E. Tronci, G. D. Penna, B. Intrigila, and M. Venturini. A probabilistic approach to automatic verification of concurrent systems. In *Proc. Asia-Pacific Software Engineering Conference (APSEC 2001)*, pages 317–324. IEEE Computer Society, 2001.
40. E. Tronci, G. D. Penna, B. Intrigila, and M. V. Zilli. Exploiting transition locality in automatic verification. In *Proc. Correct Hardware Design and Verification Methods (CHARME 2001)*, volume 2144 of *LNCS*, pages 259–274, 2001.
41. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In D. Kozen, editor, *Proc. of Logic in Computer Science (LICS '86)*, pages 332–344. IEEE Computer Society Press, 1986.