Typical Structural Properties of State Spaces

Radek Pelánek *

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic
xpelanek@fi.muni.cz

Abstract. Explicit model checking algorithms explore the full state space of a system. We have gathered a large collection of state spaces and performed an extensive study of their structural properties. The results show that state spaces have several typical properties and that they differ significantly from both random graphs and regular graphs. We point out how to exploit these typical properties 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. The state space is used to check specifications expressed in a suitable temporal logic. The main obstacle of model checking is *state explosion* — the size of the state space grows exponentially with the size of the model description. Hence, model checking has to deal with extremely large graphs.

The classical model for large unstructured graphs is the *random graph* model of Erdős and Renyi [11]. In this model every pair of nodes is connected with an edge with a given probability p. Large graphs are studied in many diverse areas, such as social sciences (networks of acquaintances), biology (food webs, protein interaction networks), geography (river 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 differ significantly from properties of random graphs. This observation led to the development of more accurate models for large graphs occurring in practice (e.g., 'small worlds' and 'scale-free networks' models) and to a better understanding of processes in these networks. For example, it improved the understanding of the spread of diseases and vulnerability of computer networks to attacks; see Barabasi [2] and Watts [32] for a high-level overview of this research and further references.

* Supported by GA ČR grant no. 201/03/0509

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 do state spaces have in common? What are their typical properties?
2. Can state spaces be modeled by random graphs or by some class of regular graphs in a satisfactory manner?
3. Can we exploit these typical properties to traverse or model check a state space more efficiently? Or at least to better analyze complexity of algorithms? Can some information about a state space be of any use to the user or to the developer of a model checker?
4. Is there any significant difference between toy academical models and real life case studies? Are state spaces similar to such an extent that it does not matter which models we choose for benchmarking our algorithms?

Methodology The basic approach is the following: we measure many graph parameters of a large collection of state spaces and try to draw answers from the results. We restrict ourselves to asynchronous models, because these are typically investigated by explicit model checkers. We consider neither labels on edges nor atomic propositions in states and thus we focus only on structural properties of graphs. For generating state spaces we have used four well-known model checkers (SPIN [22], CADP [14], Murphi [10], μCR [16]L) and two experimental model checkers. In all, we have used 55 different models including many large case studies (see Appendix A). In this report we summarize our observations, point out possible applications, and try to outline some answers. The project's web page [1] contains more details about investigated state spaces and the way in which they were generated. Moreover, interested reader can find on the web page all state spaces in a simple textual format together with a detailed report for each of them, summary tables for each measured parameter, and more summary statistics and figures.

Related work Many authors point out the importance of the study of models occurring in practice (e.g., [13]). 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 [5, 28], back level edges [31, 3], degree, stack depth [20]. Some authors make implicit assumptions about the structure of state spaces [7, 23] or claim that the usefulness of their approach is based on characteristics of state spaces without actually identifying these characteristics [30]. Groote and van Ham [17] try to visualize large state spaces with the goal of providing the user with better insight into a model.

Organization of the paper Section 2 describes studied parameters, results of measurements, their analysis, and possible application. Section 3 compares different classes of state spaces. An impatient reader can jump directly to Section 4 where our observations are summarized and where we provide some answers. Finally, the last section outlines several new questions for future research.

2 Parameters of State Spaces

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[1] $G = (V, E, v_0)$ with a set of vertices V, a set of directed edges $E \subseteq V \times V$, and a distinguished initial vertex v_0. Moreover, we suppose that all vertices are reachable from the initial one. 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.

2.1 Degrees

Out-degree (*in-degree*) of a vertex is the number of edges leading from (to) this vertex. *Average degree* is just $|E|/|V|$. The basic observation is that the average degree is very small – typically around 3 (Fig. 1). Maximal in-(out-)degree is often several times higher than the average degree but with respect to the size of the state space it is 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 [2].

The fact that state spaces are sparse is not surprising and was observed long ago[2]. 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 due to the inner nondeterminism of components; and both of these are usually very 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.

In many cases the average degree is even smaller than two, since there are many vertices with degree one. This observation can be used for saving some memory during the state space traversal [4].

2.2 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 the 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

[1] We consider state spaces as simple graphs, i.e., we do not consider self-loops and multiedges. Although these may be significant for model checking temporal logics, they are not that important for the structural properties we consider here.

[2] Holzman [20] gives an estimate 2 for average degree.

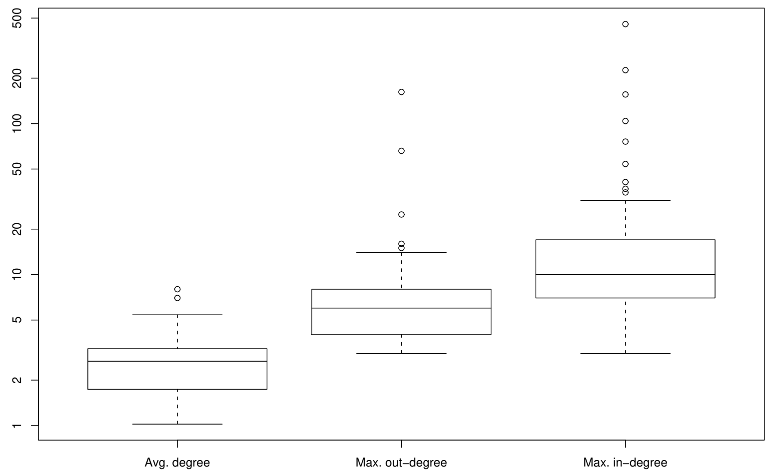

Fig. 1. Degree statistics. Values are displayed with the boxplot method. The upper and lower lines are maximum and minimum values, the middle line is a median, the other two are quartiles. Circles mark outliers. Note the logarithmic y-axis.

path in the quotient graph of G. A component is *trivial* if it contains only one vertex. 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. The structure of quotient graph has one of the following types:

– there is only one SCC component (18% of cases),
– there are only trivial components (the graph is acyclic) (14% of cases),
– there is one large component which contains most states; the largest component is usually terminal and often it is even the only terminal.

The number of SCCs can be very high, but this is mainly due to trivial components. The conclusion is that most states lie either in the largest component or in some trivial component and that the largest component tends to be 'at the bottom' of the SCC quotient graph.

SCCs play an important role in many model checking algorithms and the above stated observation can be quite significant with respect to practical applicability of some approaches, for example:

– The runtime of symbolic SCC decomposition algorithms [12, 25] depends very much on the structure of the SCC quotient graph. The thorough analysis [27] shows that the complexity of these algorithms depends on the SCC

quotient height, the number of SCC, and the number of nontrivial SCC. We note that symbolic algorithms are usually used for synchronous models (whereas our state spaces correspond to asynchronous ones) and thus our observations are not directly applicable here. However, the distributed explicit cycle detection algorithm [6] has complexity proportional to the SCC quotient height as well.
- The existence of one large component shows the limited applicability of some algorithms. The 'sweep line' method [8] of the state space exploration saves memory by deleting states which are in a fully explored SCC. The distributed cycle detection based on partitioning the state space with respect to SCCs [23] assigns to each computer in a network one or more SCCs.
- On the other hand, some algorithms could be simpler for state spaces which have one big component. For example during random walk there is a little danger that the algorithm will stuck in some small component.

2.3 Properties of Breadth-First and Depth-First Search

The basic model checking procedure is a reachability analysis – searching a state space for an error state. Here we consider two basic methods for state space traversal and their properties.

Breadth-First Search (BFS) Let us consider the BFS from the initial vertex v_0. A *level* of the BFS with an index k is a set of states with distance from v_0 equal to k. The *BFS height* is the largest index of a non-empty 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.

- The BFS height is small (Fig. 2). 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*[3]. See Fig. 3 for several such graphs. Usually this graph has a 'bell-like' shape.
- The relative number of back level edges is (rather uniformly) distributed between 0% and 50%. Most edges are local — they connect two close levels (as already observed by Tronci et al. [31]). However, for most models there exist some long back level edges. For exact results and statistics see [1].
- For most systems we observe that there are only a few typical lengths of back level edges and that most back level edges have these lengths. This is probably caused by the fact that back level edges correspond to jumps in the model. There are typically only a reasonably small number of different jumps in a model.

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

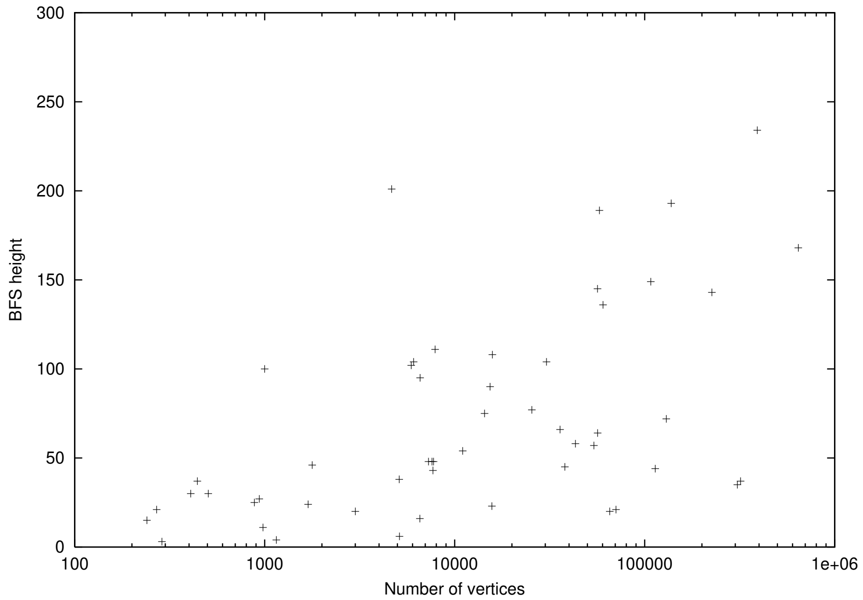

Fig. 2. The BFS height plotted against the size of the state space. Note the logarithmic x-axis. Three examples have height larger than 300.

- Tronci et al. [31, 24] exploit locality of edges for state space caching. Their technique stores only recently visited states. The efficiency (and termination) of their algorithm rely on the fact that most edges are local and hence target states of edges are usually in the cache. In a similar way, one could exploit typical lengths of back level edges or try to estimate the maximal length of a back level edge and use this estimate as a key for removing states from the cache.
- The algorithm for distributed cycle detection by Barnat et al. [3] has complexity proportional to the number of back level edges.
- The typical shape of the BFS level graph can be exploited for a prediction of the size of a state space. Particularly, when a model checker runs out of memory it may be useful to see the BFS level graph — it can help the user to decide, whether it will be sufficient just to use a more powerful computer (or a distributed computation on several computers) or whether the model is hopelessly big and it is necessary to use some reduction and/or abstraction. This is easy to implement (and add to existing model checkers) and in our experience it can be very useful to the user.

Depth-First Search (DFS) Next we consider the depth-first search from the initial vertex. The behavior of DFS (but not the completeness) depends on the

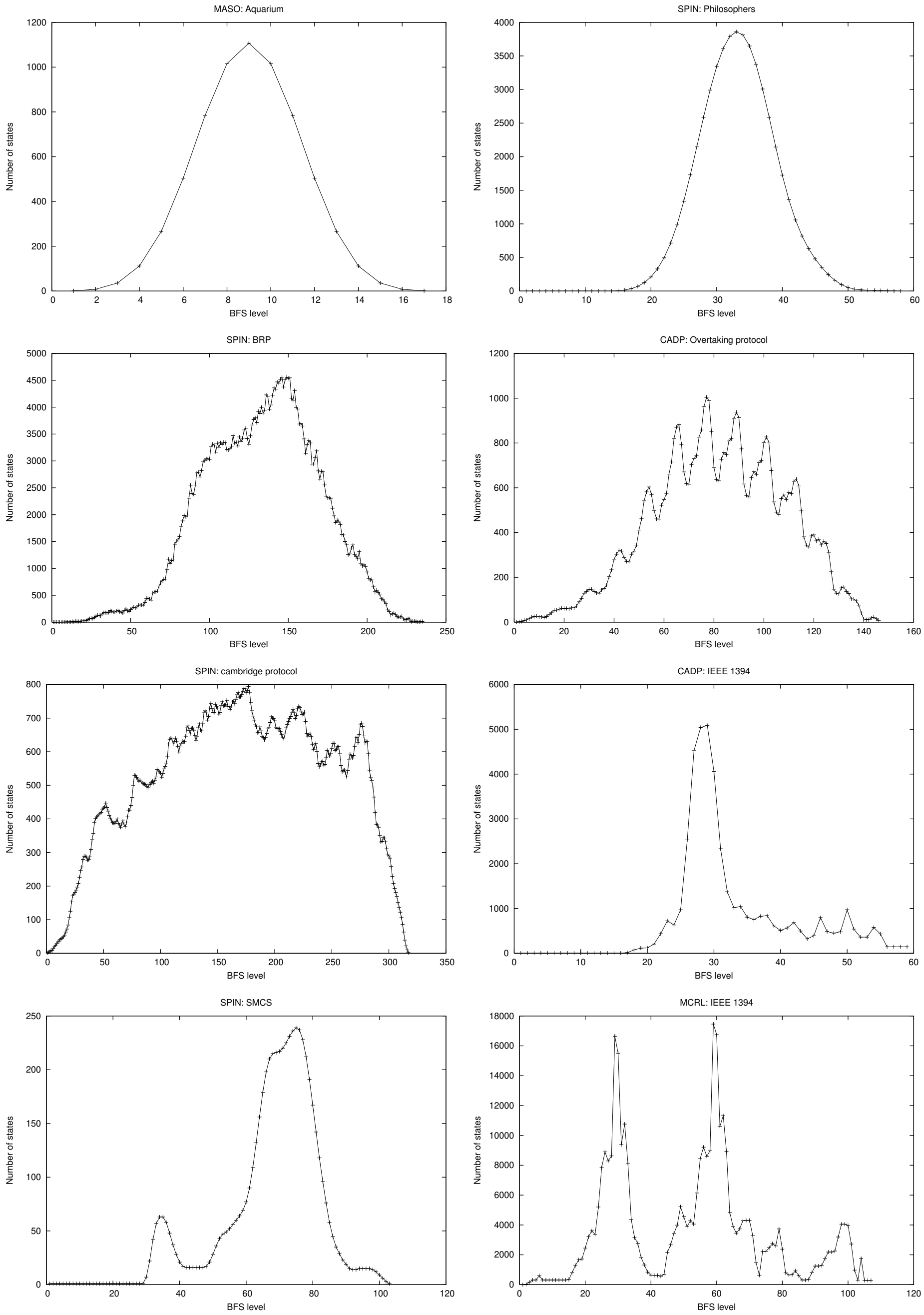

Fig. 3. BFS level graphs. For simple models the curve is really smooth and bell-like. For more complex models it can be a bit ragged. The last two graphs show that there are exceptions which have more 'peaks' (but these are rare exceptions).

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. 4 shows several stack graphs (for more graphs see [1]). 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. Moreover, each state space has its own typical graph. In contrast, all random graphs have rather the same, smooth stack graph.

When we count the length of cycles encountered during DFS we find out that there are several typical lengths of cycles which occur very often; after the observation of the typical lengths of back level edges this does not come as a great surprise.

These observations point out interesting structural properties of state spaces (and their differences from random graphs) but do not seem to have many direct applications. The only one is the stack cycling technique [21] which exploits the fact that the size of the stack does not change too quickly and stores part of the stack on the magnetic disc. Stack graphs could provide better insight into how to manage this process.

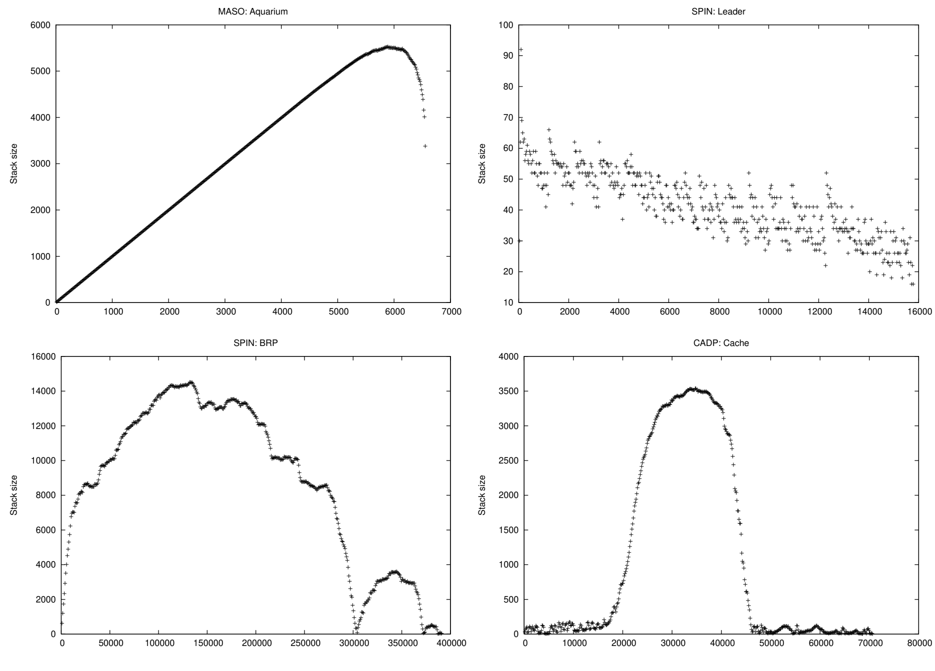

Fig. 4. Stack graphs. The first one is the stack graph of a very simple model. Stack graphs of random graphs are similar to this one. The other three stack graphs correspond to more complex models.

Queue and Stack Size For implementations of the breadth- and depth-first search one uses queue and stack data structures. These data structures are in most model checkers treated differently from a set of already visited states. This set (usually implemented as a hash table) is considered to be the main memory consumer. Therefore its size is reduced using sophisticated techniques: states are compressed with lossless compression [19] or bit-state hashing [18], stored on magnetic disc [29], or only some states are stored [4, 15]. On the other hand, the whole queue/stack is typically kept in memory without any compression. Our measurements show that the sizes of these structures are often as much as 10% of the size of a state space; see Fig. 5 for results and comparison of queue and stack sizes. Thus 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. We note that this is already done is some model checkers – SPIN can store part of a stack on disc [21], UPPAAL stores all states in the hash table and maintains only references in a queue/stack [9].

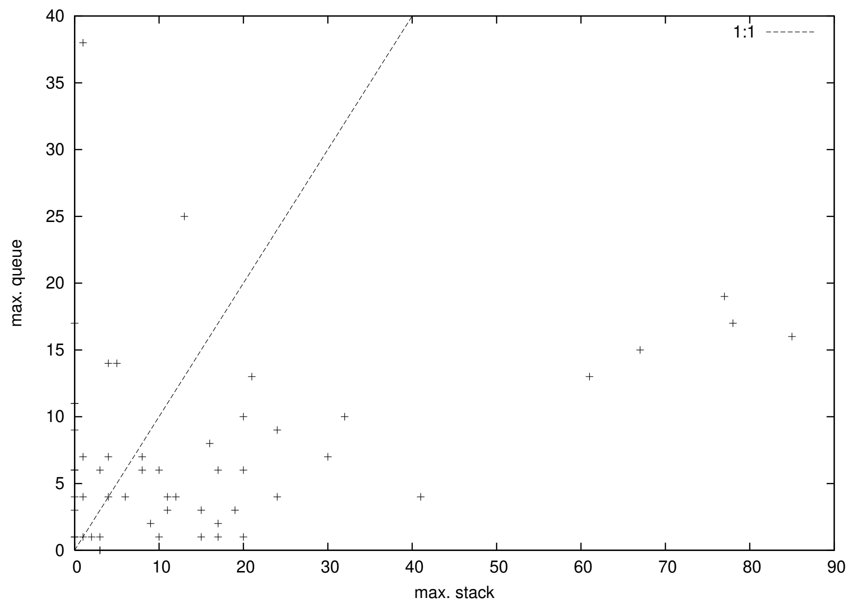

Fig. 5. A comparison of maximal queue and stack sizes expressed as percentages of the state space size. Note that the relative size of a queue is always smaller than 40% of the state space size whereas the relative size of a stack can go up to 90% of the state space size.

2.4 Distances

The *diameter* of a graph is the length of the largest shortest path between two vertices. The *girth* of a graph is the length of the shortest cycle. Since diameter and girth are expensive to compute[4] we can determine them only for small state spaces.

However, experiments for small graphs reveal that we can compute good estimates of these parameters with the use of the breadth- and depth-first search. The BFS height can be used to estimate the diameter. For most state spaces the diameter is smaller than 1.5 times the BFS height. Note that for general graphs the diameter can be much larger than the BFS height. DFS can be used to estimate the girth – it is not guaranteed to find the shortest cycle but our experience shows that in practice it does.

It is a 'common belief' (only partially supported in some papers) that the diameter is small. Our experiments confirm this belief. In most cases the diameter is smaller than 200, often much smaller[5]. The girth is in most cases smaller than 8.

The fact that the diameter is small is practically very important. Several algorithms (e.g., [28, 12, 25]) and the bounded model checking approach [5] directly exploit this fact. Moreover, the fact that the diameter is small suggests that many of the very long counterexamples (as produced by some model checkers) are caused by a poor search and not by the inherent complexity of the bug.

2.5 Local Structure

As the next step we try to analyze the local structure of state spaces. In order to do so, we employ some ideas from the analysis of social networks. A typical characteristics 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.

- A *diamond* rooted at v_1 is a quadruple (v_1, v_2, v_3, v_4) such that $\{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4)\} \subseteq E$.
- The *k-neighborhood* of v is the set of vertices with distance from v smaller or equal to k.

[4] In the context of large state spaces even quadratic algorithms are expensive.

[5] Diameters of state spaces are very small with respect to their size and to the theoretical worst case. But compared to random graphs or 'small world' networks it is still rather large (the diameter of these graphs is proportional to the logarithm of their size).

– 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 the 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. Random graphs have clustering coefficients close to 1.

The measurements confirm that the local structure of state spaces significantly differ from random graphs (see [1] for more details):

– The size of neighborhood grows much more slowly for state spaces than for random graphs (Fig. 6). This is because the clustering coefficient of state spaces increases (rather linearly) with average degree.
– Diamonds display an interesting dependence on the average degree. For a state space with average degree less than two there are a small number of diamonds. For state spaces with average degree larger than two there are a lot of them.
– Although girth is small for all state spaces, short cycles are abundant only in some graphs — only one third of state spaces have many short cycles.
– The local structure is similar in all parts of a state space.

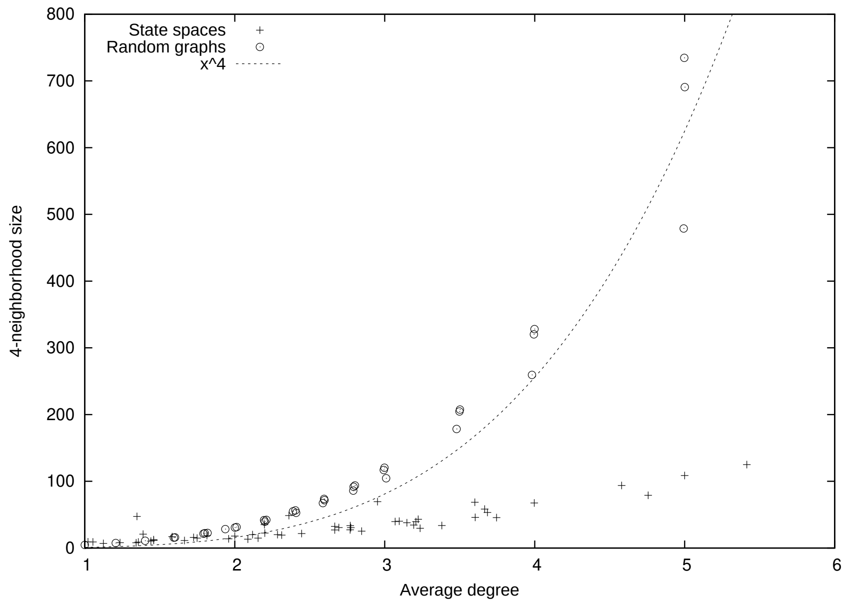

Fig. 6. Relationship between the size of 4-neighborhood and the average degree, and a comparison with random graphs.

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 states of degree one). Whereas with the high average degree it has many diamonds and high clustering coefficient. The rather surprising consequence is that the local structure depends on the model only in as much as the average degree does.

This is just the first step in understanding the local structure of state spaces, so it is difficult to give any specific applications. Some of these properties could be exploited by traversal methods which do not store all states [4]. Since the size of a neighborhood grows rather slowly, it might be feasible to do some kind of 'look-ahead' during the exploration of a state space (this is not the case for arbitrary graphs).

3 Comparisons

3.1 Specification Languages and Tools

Most parameters seem to be independent of the specification language used for modeling and the tool used for generating a state space. In fact, the same protocols modeled in different languages yield very similar state spaces. This can be seen as an encouraging result since it says that it is fair to do experimental work with just one model checker.

We have noticed some small differences. For example, Promela models often have sparser state spaces. But because we do not have the same set of examples modeled in all specification languages, we cannot fully support these observations at the moment.

3.2 Toy versus Industrial Examples

We have manually classified examples into three categories: toy (16), simple (25), and complex (14) (see Appendix A). The major criterion for the classification was the length of the model description. The comparison shows differences in most parameters. Here we only briefly summarize the main trends; more detailed figures can be found on the project's web page [1]:

- The average degree is smaller for state spaces of complex models. This is important because the average degree has a strong correlation with the local structure of the state space (see Section 2.5).
- The maximal size of the stack during DFS is significantly shorter for complex models (Fig. 7).
- The BFS height and the diameter are 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.

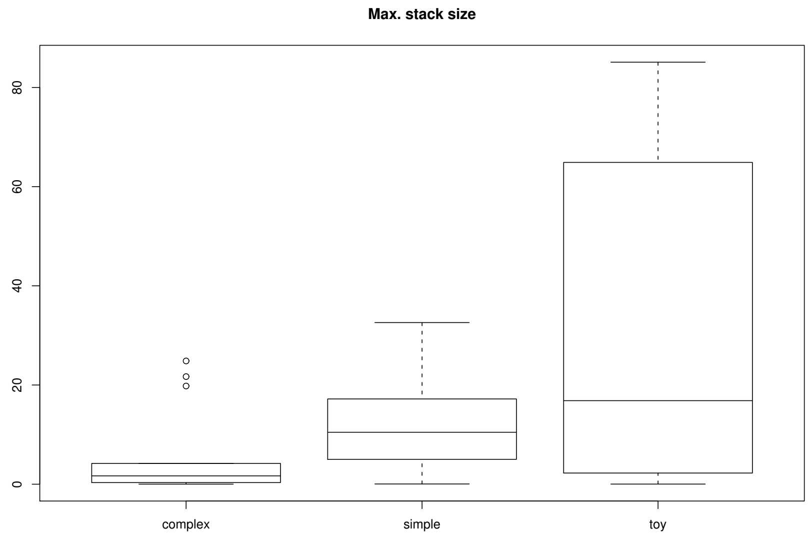

Fig. 7. The maximal stack size (given in percents of the state space size) during DFS. Results are displayed with the boxplot method (see Fig. 1 for explanation).

– Global structure is more regular for state spaces of toy models. This is demonstrated by BFS level graphs and stack graphs which are much 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.

3.3 Similar Models

We also compared the state spaces of similar models — parametrized models with different values, abstracted models, models with small syntactic change. Moreover, we have compared full state spaces and state spaces reduced with partial order reduction and strong bisimulation.

The resulting state spaces are very similar — most parameters are (nearly) the same or are appropriately scaled with respect to the change in the size of the state space. The exception is that a small syntactic change in a model can sometimes produce a big change of the state space. This occurs mainly in cases where the small change corresponds to some error (or correction) in the model. This suggests that listing state space's parameters could be useful for users during modeling — the significant change of parameters between two consecutive versions of a model can serve as a warning of a potential error (this can be even done automatically).

4 Conclusions: Answers

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. We used several different model checkers and models were written by several different users. 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 or by some class of regular graphs?

State spaces are neither random nor regular. They have some internal structure, but this structure is not strictly regular. This is illustrated by many of our observations:

- local clustering (including diamonds) is completely absent in random graphs
- stack graphs and BFS level graphs are quite structured and ragged, while for both regular and random graphs they are much smoother
- typical values of lengths of back level edges and cycles
- the diameter is larger than for random graphs but small compared to the size of the state space (definitely much smaller than for common regular graphs)

Can we exploit typical properties during model checking?

Typical properties can be useful in many different ways. Throughout the paper we provide several pointers to work that exploits typical values of parameters and we give some more suggestions about how to exploit them.

Values of parameters are not very useful for non-expert users who are not usually aware of what a state space is, but they may be useful for advanced users of the model checker. Properties of the underlying state space can provide users with feedback on their modeling and sanity checks — users can confront obtained parameters with their intuition (particularly useful for SCCs) and compare parameters of similar models, e.g., original and modified model.

The parameter values can be definitively useful for developers of tools, particularly for researchers developing new algorithms — they can help to explain the behavior of new algorithms.

Are there any differences between toy and complex models?
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. We can illustrate it on an experiment with random walk. We have performed series of very simple experiments with random walks on generated state spaces. 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 instancies of some toy example[6] and we support calls for a robust set of benchmark examples for model checking [13].

5 Future Work: New Questions

- What more can we say about state spaces when we consider atomic propositions in states (respectively good/bad states)? What is the typical distribution of good/bad states? How many are there? Where are they? What are the properties of product graphs used in LTL model checking (product with Büchi automaton) and branching time logic model checking (game graphs)? Do they have the same properties or are there any significant differences?
- Can we estimate structural properties of a state space from static analysis of its model?
- In this work we consider mainly 'static' properties of state spaces. We have briefly mentioned only the breadth- and depth-first search, but there are many other possible searches and processes over state spaces (particularly random walk and partial searches). What is the 'dynamics' of state spaces?
- What is the effect of efficient modeling [26] on the resulting state space?
- State spaces are quite structured and regular. But how can we capture this regularity exactly? How can we employ this regularity during model checking? Can the understanding of the local structure help us to devise symbolic methods for asynchronous models?

Acknowledgment

I thank my supervisor, Ivana Černá, for many valuable discussions and suggestions, and for comments on previous drafts of this paper. Pavel Krčál, Tomáš Hanžl, and other members of the ParaDiSe laboratory have provided useful feedback. Finally, I thank anonymous reviewer for detailed comments.

[6] Thou shalt not do experiments (only) on Philosophers.

References

1. http://www.fi.muni.cz/~xpelanek/state_spaces.
2. A.L. Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means.* Plume, 2003.
3. 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.
4. 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.
5. 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.
6. I. Černá and R. Pelánek. Distributed explicit fair cycle detection. In *Proc. SPIN workshop*, volume 2648 of *LNCS*, pages 49–73, 2003.
7. 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.
8. 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.
9. 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.
10. D. L. Dill, A. J. Drexler, A. J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proc. Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.
11. P. Erdős and A. Renyi. On random graphs. *Publ. Math.*, 6:290–297, 1959.
12. K. Fisler, R. Fraer, G. Kamhi Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 420–434, 2001.
13. 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.
14. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002.
15. P. Godefroid, G.J. Holzmann, and D. Pirottin. State space caching revisited. *Formal Methods in System Design*, 7(3):227–241, 1995.
16. J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62, 1995.
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. An analysis of bitstate hashing. In *Proc. Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, 1995.
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. Algorithms for automated protocol verification. *AT&T Technical Journal*, 69(2):32–44, 1990.
21. 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.
22. G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, 2003.
23. A. L. Lafuente. Simplified distributed LTL model checking by localizing cycles. Technical Report 176, Institut für Informatik, Universität Freiburg, July 2002.
24. G. D. Penna, B. Intrigila, E. Tronci, and M. V. Zilli. Exploiting transition locality in the disk based Murphi verifier. In *Proc. Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 202–219, 2002.
25. K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Proc. Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *LNCS*, pages 143–160, 2000.
26. Theo C. Ruys. Low-fat recipes for SPIN. In *Proc. SPIN Workshop*, volume 1885 of *LNCS*, pages 287–321, 2000.
27. F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic SCC hull algorithms. In *Proc. Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 88–105, 2002.
28. U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration.* PhD thesis, Technical University of Munich, 1997.
29. 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.
30. 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.
31. 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, pages 259–274, 2001.
32. D. J. Watts. *Six Degrees: The Science of a Connected Age.* W.W. Norton & Company, 2003.

A Models

Tool	Model	Type	Size
Murphi	Peterson's mutual exclusion algorithm	toy	882
Murphi	Parallel sorting	toy	3,000
Murphi	Hardware arbiter	simple	1,778
Murphi	Distributed quering lock	simple	7,597
Murphi	Needham-Schroeder protocol	complex	980
Murphi	Dash protocol	complex	1,694
Murphi	Cache coherence protocol	complex	15,703
Murphi	Scalable coherent interface (SCI)	complex	38,034

Tool	Model	Type	Size
SPIN	Peterson protocol for 3 processes	toy	30,432
SPIN	Dining philosophers	toy	54,049
SPIN	Concurrent sorting	toy	107,728
SPIN	Alternating bit protocol	simple	442
SPIN	Readers, writers	simple	936
SPIN	Token ring	simple	7,744
SPIN	Snooping cache algorithm	simple	14,361
SPIN	Leader election in unidirectional ring	simple	15,791
SPIN	Go-back-N sliding window protocol	simple	35,861
SPIN	Cambridge ring protocol	simple	162,530
SPIN	Model of cell-phone handoff strategy	simple	225,670
SPIN	Bounded retransmition protocol	simple	391,312
SPIN	ITU-T multipoint communication service	complex	5,904
SPIN	Flight guidance system	complex	57,786
SPIN	Flow control layer validation	complex	137,897
SPIN	Needham-Schroeder public key protocol	complex	307,218
CADP	Alternating bit	simple	270
CADP	HAVi leader election protocol	simple	5,107
CADP	INRES protocol	simple	7,887
CADP	Invoicing case study	simple	16,110
CADP	Car overtaking protocol	simple	56,482
CADP	Philips' bounded retransmission protocol	simple	60,381
CADP	Directory-based cache coherency protocol	simple	70,643
CADP	Reliable multicast protocol	simple	113,590
CADP	Cluster file system	complex	11,031
CADP	CO4 protocol for distributed knowledge bases	complex	25,496
CADP	IEEE 1394 high performance serial bus	complex	43,172
μCRL	Chatbox	toy	65,536
μCRL	Onebit sliding window protocol	simple	319,732
μCRL	Modular hef system	complex	15,349
μCRL	Link layer protocol of the IEEE-1394	complex	371,804
μCRL	Distributed lift system	complex	129,849
Divine	Cabbage, goat, wolf puzzle	toy	52
Divine	Dining philosophers	toy	728
Divine	MSMIE protocol	simple	1,241
Divine	Bounded retransmition protocol	simple	6,093
Divine	Alternating bit protocol	simple	11,268
MASO	Aquarium example	toy	6,561
MASO	Token ring	toy	7680
MASO	Alternating bit protocol	toy	11,268
MASO	Adding puzzle	toy	56,561
MASO	Elevator	simple	643,298