
Faculty of Informatics
Masaryk University
Czech Republic

# Experimental Research in Explicit Model Checking

Habilitation Thesis

Radek Pelánek

2010

# Abstract

The thesis consists of an introductory commentary and ten papers. Reported research falls within the general area of formal verification. More specifically the research deals with the explicit model checking technique for finite state systems with focus on experimental aspects of research. It contains both general methodological discussions (e.g., evaluation of use of benchmarking examples) and specific experimental comparisons of algorithms. These results serve as a basis for proposal of new methods and tools.

The keystone of this work is the BEEM project: *BEnchmarks for Explicit Model checkers*. This benchmark set is an important output of the research in itself – in 3 years since it publication, BEEM was used in more than 30 research publications. The benchmark set also serves as a basis for follow-up research reported in the thesis. The set was used to analyze properties of practically used models and their state spaces (as reported in papers *Properties of State Spaces and Their Applications*, *Model Classifications and Automated Verification*, and *Estimating State Space Parameters*) and to experimentaly compare different algorithms for explicit model checking (as reported in papers *Evaluation of State Caching and State Compression Techniques*, *Complementarity of Error Detection Techniques*, and *Fighting State Space Explosion: Review and Evaluation*). Two other experimental studies were carried out without using the BEEM (*Enhancing random walk state space exploration* and *Test input generation for red black trees using abstraction*).

The experiences with practically used models and with experimental comparisons of algorithms showed a large degree of complementarity among different algorithms. This conclusion led to a proposal of a concept of a verification manager (reported in the paper *Verification Manager: Automating the Verification Process*). Verification manager runs several algorithms in parallel and dynamically coordinates individual runs. In this way, the manager is able to harness the complementarity of different algorithms.

The thesis consists of ten papers. Four papers were written solely by the author of the thesis. Six papers were written with one or two co-author; in each case the author has contributed by at least 30%.

# Abstrakt

Předkládáná práce se skládá z úvodního komentáře a deseti příspěvků. Práce spadá do oblasti formální verifikace počítačových systémů, konkrétně se zabývá metodou explicitního ověřování konečně stavových modelů. Zaměřuje se především na experimentální aspekty výzkumu v této oblasti, a to jak na obecné metodické úrovni (např. vyhodnocení dosavadního experimentálního výzkumu v oblasti), tak v oblasti konkrétního experimentálního vyhodnocení algoritmů. Na základě těchto výsledků pak jsou navrhnuty nové postupy verifikace.

Výchozím bodem práce je srovnávací sbírka modelů BEEM – *BEnchmarks for Explicit Model checkers*. Tato sbírka jednak představuje samostatný důležitý výstup, který se setkal s dobrým přijetím mezi výzkumníky zabývajícími se explicitním ověřováním modelů (BEEM byl během 3 let použit ve více než 30 recenzovaných publikacích), a také tvoří základ, na kterém staví další příspěvky obsažené v práci. S využitím této sbírky modelů byly provedeny analýzy vlastností prakticky používaných modelů (viz příspěvky *Properties of State Spaces and Their Applications*, *Model Classifications and Automated Verification* a *Estimating State Space Parameters*) a porovnání různých algoritmů pro explicitní ověřování modelů (viz příspěvky *Evaluation of State Caching and State Compression Techniques*, *Complementarity of Error Detection Techniques* a *Fighting State Space Explosion: Review and Evaluation*). Další experimentální studie pak byly provedeny i mimo rámec sbírky BEEM (viz příspěvky *Enhancing random walk state space exploration* a *Test input generation for red black trees using abstraction*).

Zkušenosti s prakticky používanými modely a s vyhodnocováním různých algoritmů pro ověřování modelu mimo jiné ukázaly vysoký stupeň komplementarity mezi jednotlivými algoritmy. Tento závěr vedl k navržení konceptu verifikačního manažera, který spouští paralelně několik algoritmů a dynamickým způsobem jednotlivé běhy koordinuje, díky čemuž dokáže využít zmíněnou komplementaritu algoritmů (viz příspěvěk *Verification Manager: Automating the Verification Process*).

U čtyř z deseti obsažených příspěvků je uchazeč jediným autorem, u většiny ostatních hrál klíčovou roli při psaní textu příspěvku a celkový obsahový autorský podíl uchazeče je vždy alespoň 30%.

# Acknowledgments

I would like to thank all coauthors of included papers (Luboš Brim, Ivana Černá, Tomáš Hanžl, Pavel Moravec, Corina S. Păsăreanu, Václav Rosecký, Pavel Šimeček, Jaroslav Šeděnka, Willem Visser) for fruitful collaboration, Ivana Černá and Pavel Krčál for many interesting discussions about my research and for feedback on drafts of papers, and developers of the DiVinE tool (particularly Jiří Barnat) for the development of the platform that I used in experimental evaluation. Last but not least, I thank my wife Barbora for support, encouragement, and love.

# Contents

# Chapter 1

# Experimental Research in Explicit Model Checking: Commentary

The presented habilitation thesis consists of a collection of ten papers. This introduction presents the overall motivation, the context of the work, and the state of the art in the area. It also outlines main common themes of the collected papers and gives a brief overview, highlighting relations among individual papers.

## 1.1 Introduction

Let us start by an informal introduction of the "big picture": why is the work important and what is the context of novel contributions?

### 1.1.1 Motivation

Computer systems are pervasive in our lives. At the same time, design and implementation of computer systems is a difficult task during which humans often make errors. This does not mean that we should resign and accept that our lives will be governed by faulty machines. We should try hard to make our computer systems as reliable as possible. In this quest, our effort has to be differentiated. Not all errors are created equal.

Some errors are just annoying, some errors are costly, and some errors are even deadly. If your word processor freezes, it bothers you and you may have to rewrite several sentences. If you operating systems freezes, it makes you upset and you may have to press the reset button and wait for a while. However, if the operating system of your space robot freezes, it may be quite a problem, since it can be complicated to press the reset button when the machine operates on an uninhabited planet. Such an error can be very expensive. Other errors can even kill – for example freezing of a medical machine controller which emulates intensive radiation.

In cases where potential errors are just annoying, it is reasonable to perform the search for errors in an informal and pragmatic way. But when potential errors are very costly or even deadly, we should try hard to find all errors in the system. This is the main motivation of this work: *finding errors in formal way in a safety-critical systems*.

### 1.1.2 Context

Development of computer systems consists of several phases: requirements specification, design, implementation, verification, and maintenance. We focus on just one of these steps – *verification*. Aim of verification is to either verify that the system satisfies given requirements or to find an example which demonstrates an incorrect behavior of the system. There are several different verification methods, e.g., inspections, testing, simulation, and formal verification. None of these methods is superior to others, each of them has its advantages, disadvantages, and domains of application. We focus on *formal verification*. In comparison to the other verification methods, formal verification can give us much higher assurance of system correctness. However, it is a difficult and time-consuming method. Therefore it is not very convenient for detecting errors in ordinary programs, but rather for verification of systems which are safety-critical.

There exist two basic approaches to formal verification: deductive methods and automatic methods. With deductive methods we try to produce a mathematical proof which states that a system satisfies given requirements. Although the construction of a proof can be partially automatized (simple proofs can be constructed algorithmically), deductive methods have to be performed by experts and are very time-consuming. The second approach are *automatic methods*, of which the most commonly used one is *model checking*. Model checking is fully automatic and in case that a system does not satisfy requirements the technique is capable of demonstrating a wrong behavior (counterexample).

Model checking methods can be classified even further. The most often used classifications are explicit versus symbolic and finite state versus infinite state model checking. We focus on *finite state explicit model checking*. This technique is applicable for systems which have a finite number of system states and works by explicitly enumerating all reachable states. Hence the main main idea of finite state explicit model checking is principally very simple: "use brute-force and try to test all possible behaviors of a system and verify that all of them satisfy requirements". Albeit principally simple, this technique is quite powerful and has many applications.

For any practically relevant specification language even the simplest verification problems are in theory algorithmically intractable (to formulate it precisely: PSPACE-complete). However, in practice the approach works and is widely studied. Due to theoretical intractability, all research in this area is principally of a heuristic nature. Therefore, it is very important to perform high-quality experimental evaluation of algorithms and to understand behaviour of algorithms. By this we have finally reached the specific topic of this thesis: *experimental research in explicit model checking*.

## 1.2 State of the Art

The first section provided the motivation and the context for the presented work. This section briefly surveys the state of the art in the area of explicit model checking. We focus only on the research closely relevant to the presented thesis.

### 1.2.1 Research

The main obstacle of model checking is the state space explosion problem – the number of states in the state space can grow exponentially in the worst case. For practical problems we do not observe exponential grow, nevertheless the size of the state space is still very large. Therefore, the main research topic in explicit model checking is the development of techniques for fighting state space explosion, i.e., techniques which aim at reducing the time and memory requirements of the state space search.

A detailed survey of techniques for fighting state space explosion is given in one of the papers included in this thesis ([61]). Here we just briefly summarise the main types of techniques:

- State space reductions: techniques reducing the number of states that need to be explored, e.g., transition merging [15, 46], partial order reduction [25, 39], symmetry reduction [41], live variable reduction [18, 71], cone of influence reduction and slicing [17, 32], and compositional methods [30, 44].
- Storage size reductions: techniques reducing the memory requirements needed for storing states, e.g., state compression [23, 27, 38, 73], state caching [22, 26], selective storing [6, 47], and sweep line method [11, 54].
- Parallel and distributed computation: techniques exploiting additional computation power, e.g., parallel computation on multi-core processor [3, 40] and distributed computation on network of workstations [21, 49, 50].
- Randomized techniques and heuristics: techniques that give up the requirement on completeness and explore only part of the state space, e.g., heuristic search [28, 45, 70], random walk [31, 64], partial search [35, 43, 53], and bitstate hashing [37].

Except for techniques for fighting state space explosion, other active research topics are for example combination of explicit model checking with other techniques (e.g., theorem proving, satisfiability solving, static analysis), transfer of ideas between different domains of application (e.g., between model checking and artificial intelligence), or application of model checking to biological models.

### 1.2.2 Applications and Tools

In general, formal verification is used mainly in the following domains: embedded systems [16], computer network communication protocols [36], traffic supervision (airlines, railways [8]), space flights systems [52], and hardware [5, 24]. Finite state explicit model checking is used particularly during development of protocols (in several of the areas given above). More specifically, explicit model checking is suitable technique for verification of systems that satisfy the following criteria:

- the control part of the system is rather sophisticated,
- data within the system have only restricted influence on the behaviour of the system,

- the system contains parallel components, which may have complicated interleavings.

Typical problems that meet these criteria are[1]:

**Mutual exclusion protocols** The goal of mutual exclusion protocols is to ensure an exclusive access to a resource which is shared by two or more processes. Examples of such protocols are Peterson's algorithm, Fischer's algorithm, or Anderson's queue lock algorithm (for overview see [1]).

**Communication protocols** The goal of communication protocols is to ensure reliable communication via unreliable or shared medium. Examples of such protocols are bounded retransmission protocol [13], sliding window protocol [14], collision avoidance protocol, or layer link protocol of the IEEE-1394 [72].

**Leader election algorithms** The goal of leader election algorithms is to choose a unique leader from a set of nodes. Leader election algorithms are often used as a part of communication protocols. Examples of such algorithms are Firewire (IEEE 1394) tree identification protocol [10], or Lann's leader election algorithm for token ring.

**Controllers of embedded systems** Controllers are algorithms that control behaviour of a distributed algorithm. Examples of controllers are elevator controller, gear controller [51], or audio/video power controller [33].

The proliferation of explicit model checking is also documented by availability of a large number of model checking tools. Let us mention just few illustrative examples:

**CADP** [20] CADP is a toolbox that offers a wide set of functionalities (e.g., simulation, equivalence checking, model checking). The basic specification language is based on a process algebra (LOTOS).

**DiVinE** [4] DiVinE (Distributed Verification Environment) is an environment specifically targeted at distributed verification. As a specification language it uses low-level language based on communicating finite state machines.

**Java PathFinder** [9] Java PathFinder uses as a specification language the general programming language Java. The basic principle of the tool is very similar to other model checkers, but it has to deal with specific problems caused by such high-level specification language.

**mCRL2** [29] mCRL2 is another toolset based on process algebra and offering wide functionality.

**Spin** [7] Spin is probably the most well-known model checker, particularly due to its speed. As a specification language it uses Promela (PROocol MOdeling LAnguage), a high level modeling language with features similar to programming languages.

[1] See web portal of the BEEM project [60] for more problems: `http://anna.fi.muni.cz/models`

**Uppaal** [48] Uppaal uses as a specification language communicating timed automata and is able to perform verification of infinite state real-time systems. However, by virtue of its good graphical interface, it is also often used for finite-state model checking.

### 1.2.3 Experimental Evaluation

Even through some research in the area of explicit model checking is rather theoretically oriented, it is the practical performance of studied techniques that really matters. For example, the development of techniques such as partial order reduction or symmetry reduction involves many interesting theoretical problems, but in the end even these techniques are just heuristics.

In order to asses the performance of heuristical techniques, it is necessary to perform high-quality experiments. Unfortunately, our analysis of research paper in explicit model checking [60, 61] suggests that the experimental standards are rather low[2] and that they are improving only slowly. This is rather disappointing because many powerful tools and interesting case studies are available (as illustrated above).

The need for benchmarking, better experiments, and thorough evaluation of tools and algorithms is well recognized, e.g., experimentation is a key part of Hoare's proposal for a "Grand Challenge of Verified Software" [34]. There is also significant interest in benchmarks in the model checking community (see e.g., [2, 12, 19, 42]). Nevertheless, the progress has been rather slow so far. The main obstacle in developing model checking benchmarks is the absence of a common modeling language – each model checking tool is tailored towards its own modeling language and even verification results over the same example are often incomparable. This makes the situation more difficult than in areas like satisfiability solving or graph algorithms, where different input formats are easily transformable and standard benchmark sets exist.

## 1.3 Main Themes

After introducing the motivation, the context, and the state of the art we can proceed to the discussion of presented contributions to the area of explicit model checking. First, we introduce the main research themes that connect individual papers included in the thesis.

### 1.3.1 Experimental Research

The common theme of all included papers is the focus on experiments. We do not study or develop novel sophisticated techniques. Rather we focus our attention on basic techniques and try to evaluate them and understand their behaviour. It turns out that it is quite challenging to understand properly the behaviour of even very simple techniques like random walk.

Our focus on experimental work forced us to confront the issue of benchmarks. Since we were not able to find a suitable benchmark set for performing our experi-

[2] Note that we have analyzed quality of experiments only with respect to used models, not with respect to other important issues like reproducibility.

ments, we decided to develop our own: BEEM = BEnchmark set for Explicit Model checkers [60]. The development of this benchmark set is one of the important contributions of our work on its own, but it is also an important base step for several of our other works.

Our experimental research has also one distinctive feature: focus on properties of state spaces. Explicit model checking works by traversing the whole state space of the model. The behaviour of explicit model checkers is therefore closely connected with the structure of the underlying state spaces. State space are usually treated as arbitrary directed graphs. However, as our research shows, these state space are definitely not arbitrary and share specific properties. In our experimental work, we have focused not only on simple performance metrics of algorithms, but also on connections between performance metrics and properties of state spaces.

### 1.3.2 Complementarity

Let us imagine a typical experimental setup: we have a problem, several competing algorithms, and a benchmark set. In idealized (and naive) world we perform experiments, analyze results, and conclude: "This is the best algorithm to run, we can throw away all others and use this one." In a real world, things are more complicated. Usually each algorithm works well on different input data, so we cannot choose one universal winner. On the other hand, it is seldom the case that all competing algorithms are of the same value.

In this situation it would be very useful to have an answer to a question "What works when?" Given an input problem, can we determine in advance which technique will work effectively for the problem? It turns out that this is a difficult problem. We have tried to tackle this question in several ways, but we were not very successful.

Nevertheless, we do not need to be bothered to much by our inability to choose a single technique for an input problem. Today, parallelism is widely available and thus we can run several techniques at once. The question of selection, however, did not disappear. Parallelism at our disposal is limited and the number of available techniques is practically unlimited (most techniques are parametrized). A naive approach would be to select $k$ most successful techniques (as evaluated over a benchmark set). With such an approach we would probably use several techniques with very similar functionality and performance – however, it is not very useful to run in parallel two techniques with very similar functionality. What we need is to choose $k$ complementary techniques. Even a technique with rather poor overall performance can be very useful, if it works for problems for which all other techniques fail (e.g., this is the the case of random walk in our experiments [67]).

### 1.3.3 Automating the Verification Process

Given a model and a specification, model checking is supposedly an automated verification technique – it algorithmically checks all possible behaviours of the model and gives us 'yes' or 'no' answer. In practice, however, model checking is quite a laborious process. Typical scenario is the following:

1. We try to run the model checker. After waiting for a nontrivial time, it fails (it runs "out of memory" or "out of patience").

2. We try to run the model checker with several different optimization techniques and parameter values.

3. When the model checker finally works and gives us an output, we find an error in the model, correct it and start the process all over again.

Thus verification itself is automatic, but the verification process is not. In order to manage successfully the verification process, an expert user is usually necessary. One of the important issues is the selection of verification techniques and parameters (due to the complementarity, as discussed above). This issue becomes even more pressing when we want to make use of a parallel environment. Even expert user cannot efficiently manage concurrent verification runs on ten workstations.

Our aim is to automate this verification process and replace the expert user by an automated "verification manager". This goal is fundamentally based on previous two themes: on experimental research, which enables us to get insight into to behaviour of verification techniques, and study of complementarity, which enables us to device strategies for the verification manager.

## 1.4 Contributions

Finally, we discuss individual papers which comprise this thesis[3]. Papers are ordered by topic, i.e., papers with similar topic are grouped together. The first three papers deal with benchmarks and state space properties, the second three papers describe with evaluation of algorithms and with their complementarity, and the last three papers are concerned with automating the verification process.

### 1.4.1 BEEM: Benchmarks for Explicit Model Checkers

In this paper we present BEEM – BEnchmarks for Explicit Model checkers. The benchmark set includes more than 50 parametrized models (together with their correctness properties), which makes it the most comprehensive benchmark set in the area of explicit model checking. Moreover, BEEM is not just a collection of models, it is also accompanied by an comprehensive web portal, which provides detailed information about all models and support for experiments. A specific novel feature of BEEM is the inclusion of information about state space properties.

We use BEEM in several of our subsequent studies and BEEM has also quickly gained popularity in the explicit model checking community. During three years since it was launched, BEEM has been used in more than 30 published papers.

The paper was published in proceedings of SPIN Workshop 2007 [60] (extended version of the paper appeared as a technical report [59]). The author of the thesis is the sole author of the paper.

[3]Descriptions of papers are also repeated just before each paper in the collection.

### 1.4.2 Properties of State Spaces and Their Applications

State spaces are usually treated as directed graphs without any specific features. However, state spaces are generated from a rather small symbolic description (a model). Thus it is conceivable that state spaces are not just arbitrary complicated directed graphs. Motivated by this simple argument, we study the following questions: What are typical properties of state spaces? What do state spaces have in common? Can state spaces be modeled by random graphs? How can we apply properties of state spaces? Can we exploit these typical properties to traverse a state space more efficiently? Are state spaces similar to such an extent that it does not matter which models we choose for benchmarking our algorithms?

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 discuss consequences of these results for model checking experiments and we point out how to exploit typical properties in practical model checking algorithms.

At first we performed this study with the use of state spaces generated by several different model checkers. These results were published in proceedings of SPIN Workshop 2004 [56]. Later we have redone the experiment with models from BEEM[4] and extended the results. As part of this thesis we present the extended version which was was published in the International Journal on Software Tools for Technology Transfer (STTT) in 2008 [63]. The author of the thesis is the sole author of the paper.

### 1.4.3 Estimating State Space Parameters

In this paper we introduce the problem of estimation of state space parameters, argue that it is an interesting and practically relevant problem, and study several simple estimation techniques. Particularly, we focus on estimation of the number of reachable states. Such estimation can be very useful in several ways, e.g., for tuning model checking algorithms, for automating the verification process, as a practical information for users, or as a criterion for selection of models for experiments.

Our techniques are based on our insight of typical properties of state spaces and on the understanding of random walk behaviour (previous two papers). We study techniques based on sampling of the state space and techniques that employ data mining techniques (classification trees, neural networks) over parameters of breadth-first search. We show that even through the studied techniques are not able to produce exact estimates, it is possible to obtain usable information.

This paper was published as a work-in-progress paper at International Workshop on Parallel and Distributed Methods in verifiCation (PDMC) in 2008 and as a technical report [69]. The author of the thesis is one of two coauthors of the paper and has done major part of both experiments and writing.

[4] Note that this work is closely connected to the BEEM project, because the studied properties of state spaces are reported in detail on the BEEM web portal.

### 1.4.4 Enhancing Random Walk State Space Exploration

In this paper we study the behavior of random walk techniques in the context of model checking. It turns out that it is rather difficult to understand the behaviour of even the simple random walk. Using the insight gained by our study of simple random walk, we propose several enhancements, e.g., combination with local exhaustive search, caching, or pseudo-parallel walks.

Thorough this work we focus on important but often neglected experimental issues like length of counterexamples, coverage estimation, and setting of parameters. We also test algorithms on inputs of different types – except for state spaces generated by explicit model checkers, we also use random graphs and regular graphs.

This paper was published in proceedings of Formal Methods for Industrial Critical Systems (FMICS) in 2005 [64]. The author of the thesis is one of four coauthors of the paper and has done major part of both experiments and writing.

### 1.4.5 Evaluation of State Caching and State Compression Techniques

In this paper we employ BEEM to thoroughly evaluate two well-studied techniques in explicit model checking: state caching and state compression techniques. The goal of these techniques is to reduce memory consumed by a model checker at the expense of (hopefully slight) increase in running time. Both of these techniques were repeatedly studied and refined in previous research.

We provide review of the literature, discuss trends in relevant research, and perform extensive experiments over models from BEEM. The conclusion of our review and evaluation is that it is more important to combine several simple techniques in an appropriate way rather than to tune a single sophisticated technique.

This paper was published as a technical report [68]. The author of the thesis is one of three coauthors and has done data analysis and most of the writing.

### 1.4.6 Complementarity of Error Detection Techniques

In this paper we study the performance of techniques for error detection and we focus particularly on the issue of complementarity. Using experimental evidence we argue that it is not important to find the best technique, but to find a set of complementary techniques (as discussed in Section 1.3.2). We choose nine diverse error detection techniques (e.g., depth-first search, directed search, random walk, and bitstate hashing) and perform experiments over the BEEM set.

The topic is closely connected to the research in testing. Therefore, in our evaluation we compare not just a speed of techniques, but also model coverage metrics that are used in the testing domain. The result of our experiments show that the studied techniques are indeed complementarity in several ways.

This paper was published in proceedings of International Workshop on Parallel and Distributed Methods in verifiCation (PDMC) in 2008 [67]. The author of the thesis is one of three coauthors of the paper and has done the analysis of data and most of the writing.

### 1.4.7 Test Input Generation for Java Containers using State Matching

The topic of this paper lies on the border between model checking and testing. We are concerned with test input generation for Java containers and we try to do it with the use of explicit model checker (Java PathFinder). We compare several techniques: exhaustive techniques based on explicit model checking, lossy techniques which are based on explicit model checking but do not visit all states, and also random selection of inputs. The basic metric used for comparison is testing coverage (more specifically, we use a predicate coverage metric).

The first surprising result is that random selection, despite its simplicity, performs surprisingly well. Nevertheless, more sophisticated techniques can beat random selection on complex inputs (e.g., implementation of a red-black tree). The most successful technique seems to be the explicit search with abstract matching of states, but similarly to our other evaluations it is not possible to declare a single universal winner.

This paper was published in proceedings of International Symposium on International Symposium on Software Testing and Analysis (ISSTA) in 2006 [55]. The author of the thesis is one of three coauthors of the paper and has contributed particularly to the experimental design, data analysis, and interpretation of results.

### 1.4.8 Fighting State Space Explosion: Review and Evaluation

This work comprises an important piece in our long term effort. It summarises both the work in the area and our own experiences, and provides a basic argument for our approach to automating the verification process.

In this paper we provide a systematic overview of techniques for fighting state space explosion and we analyse trends in the relevant research. We also report on our own experience with practical performance of techniques – the report is a concise summary of several other papers and technical reports [57, 58, 59, 67, 68]. Main conclusion of the study is a recommendation for both practitioners and researchers: be critical to claims of dramatic improvement brought by a single sophisticated technique, rather use many different simple techniques and combine them.

This paper was published in proceedings of Formal Methods for Industrial Critical Systems (FMICS) in 2008 [61]. The author of the thesis is the sole author of the paper.

### 1.4.9 Model Classifications and Automated Verification

In this paper we discuss the issue of automating the verification process, formulate the verification meta-search problem, and propose the concept of a verification manager. We also discuss general ideas for the realization of the verification manager.

On a specific level the paper is concerned with development of model classifications. Proposed classifications are based both on the syntax of the model (e.g., communication mode, process similarity, application domain) and on properties of state space (e.g., structure of SCC components, shape of the state space, local structure). Classifications were derived from experimental study of models in the BEEM set.

This paper was published in proceedings of Formal Methods for Industrial Critical Systems (FMICS) in 2007 [62]. The author of the thesis is the sole author of the paper.

### 1.4.10 Verification Manager: Automating the Verification Process

In this work we further develop the concept of a verification manager (as outlined in Section 1.3.3). Particularly, we describe a practical realization of this concept for explicit model checking by building a tool EMMA (Explicit Model checking MAnager). The design of the tool is based on our experiences with evaluation of individual techniques (as discussed in other papers in the thesis), i.e., rather then developing few sophisticated techniques, we employ a large number of simple techniques which are executed in parallel.

We also discuss practical experience with the tool. We pay special attention to the problem of selection of problems for experiments. This issue is important (but often neglected) in all experiments, but it becomes crucial in evaluating strategies for the verification manager, which are in principle meta-heuristics.

This paper was published as a technical report [66], short version of the paper was published in proceedings of Model Checking Software (SPIN Workshop) in 2009 [65]. The author of the thesis is one of two coauthors and has done data analysis and most of the writing.

# Bibliography


[1] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distrib. Comput.*, 16(2-3):75–110, 2003.

[2] D. A. Atiya, N. Catano, and G. Lüettgen. Towards a benchmark for model checkers of asynchronous concurrent systems. In *Fifth International Workshop on Automated Verification of Critical Systems: AVOCs*, University of Warwick, United Kingdom, Sept. 12–13 2005.

[3] J. Barnat, L. Brim, and P. Rockai. Scalable multi-core ltl model-checking. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[4] 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`.

[5] J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. Model checking the IBM gigahertz processor: An abstraction algorithm for high-performance netlists. In *Proc. of Computer Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 72–83. Springer, 1999.

[6] G. Behrmann, K. G. Larsen, and R. Pelánek. To store or not to store. In *Proc. of Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*. Springer, 2003.

[7] M. Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

[8] C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, and D. Romano. A formal verification environment for railway signaling system design. *Formal Methods in System Design: An International Journal*, 12(2):139–161, March 1998.

[9] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder–a second generation of a Java model-checker. In *Workshop on Advances in Verification*, pages 130–135, 2000.

[10] M. Calder and A. Miller. Using SPIN to Analyse the Tree Identification Phase of the IEEE 1394 High-Performance Serial Bus (FireWire) Protocol. *Formal Aspects of Computing*, 14(3):247–266, 2003.

[11] 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 2001)*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.

[12] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–180, 1996.

[13] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *LNCS*, pages 416–431. Springer-Verlag, 1997.

[14] Y. Dong, X. Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, and D.S. Warren. Fighting livelock in the i-protocol: A comparative study of verification tools. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems*, pages 74–88. Springer, 1999.

[15] Y. Dong and C. R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proc. of Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 241–256. Kluwer, B.V., 1999.

[16] B. Dutertre and V. Stavridou. Formal requirements analysis of an avionics control system. *Software Engineering*, 23(5):267–278, 1997.

[17] M B. Dwyer, J. Hatcliff, M. Hoosier, V. P. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 73–89, 2006.

[18] J. C. Fernandez, M. Bozga, and L. Ghirvu. State space reduction based on live variables analysis. *Journal of Science of Computer Programming (SCP)*, 47(2-3):203–220, 2003.

[19] 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.

[20] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, 2002.

[21] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. SPIN Workshop*, volume 2057 of *LNCS*, pages 217–234. Springer, 2001.

[22] J. Geldenhuys. State caching reconsidered. In *SPIN*, volume 2989 of *LNCS*, pages 23–38. Springer, 2004.

[23] J. Geldenhuys and P. J. A. de Villiers. Runtime efficient state compaction in SPIN. In *Proc. of SPIN Workshop*, volume 1680 of *LNCS*, pages 12–21. Springer, 1999.

[24] R. Gerth. Model checking if your life depends on it: A view from Intel's trenches. In *Proc. SPIN workshop*, volume 2057 of *LNCS*. Springer, 2001.

[25] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *LNCS*. Springer, 1996.

[26] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. of Computer Aided Verification (CAV 1992)*, volume 663 of *LNCS*, pages 178–191. Springer, 1992.

[27] J. Gregoire. State space compression in spin with GETSs. In *Proc. Second SPIN Workshop*. Rutgers University, New Brunswick, New Jersey, 1996.

[28] A. Groce and W. Visser. Heuristics for model checking java programs. *Software Tools for Technology Transfer (STTT)*, 6(4):260–276, 2004.

[29] J.F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In *Methods for Modelling Software Systems (MMOSS)*, volume 6351 of *Dagstuhl Seminar Proceedings*, 2007.

[30] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[31] P. Haslum. Model checking by random walk. In *Proc. of ECSEL Workshop*, 1999.

[32] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher Order Symbol. Comput.*, 13(4):315–353, 2000.

[33] K. Havelund, K. G. Larsen, and A. Skou. Formal verification of a power controller using the real-time model checker uppaal. In *ARTS '99: Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, pages 277–298, London, UK, 1999. Springer-Verlag.

[34] T. Hoare. The ideal of verified software. In *Proc. of Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 5–16. Springer, 2006.

[35] G. J. Holzmann. Algorithms for automated protocol verification. *AT&T Technical Journal*, 69(2):32–44, 1990.

[36] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[37] G. J. Holzmann. An analysis of bitstate hashing. In *Proc. of Protocol Specification, Testing, and Verification*, pages 301–314. Chapman & Hall, 1995.

[38] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. of Protocol Specification, Testing, and Verification*, 1992.

[39] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. of Formal Description Techniques VII*, pages 197–211. Chapman & Hall, Ltd., 1995.

[40] G.J. Holzmann and D. Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[41] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.

[42] M. Jones, E. Mercer, T. Bao, R. Kumar, and P. Lamborn. Benchmarking explicit state parallel model checkers. In *Proc. of Workshop on Parallel and Distributed Model Checking (PDMC'03)*, volume 89 of *ENTCS*. Elsevier, 2003.

[43] M. D. Jones and J. Sorber. Parallel search for LTL violations. *Software Tools for Technology Transfer (STTT)*, 7(1):31–42, 2005.

[44] J.P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1997)*, volume 1217 of *LNCS*, pages 239–258. Springer, 1997.

[45] 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.

[46] R. P . Kurshan, V. Levin, and H. Yenigün. Compressing transitions for model checking. In *Proc. of Computer Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 569–581. Springer, 2002.

[47] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proc. of Real-Time Systems Symposium (RTSS'97)*, pages 14–24. IEEE Computer Society Press, 1997.

[48] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[49] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of SPIN workshop*, volume 1680 of *LNCS*. Springer, 1999.

[50] 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.

[51] M. Lindahl, P. Pettersson, and W. Yi. Formal Design and Analysis of a Gearbox Controller. *Springer International Journal of Software Tools for Technology Transfer (STTT)*, 3(3):353–368, 2001.

[52] M. R. Lowry. Software construction and analysis tools for future space missions. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 1–19. Springer, 2002.

[53] 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. Springer, 1994.

[54] T. Mailund and W. Westergaard. Obtaining memory-efficient reachability graph representations using the sweep-line method. In *TACAS*, volume 2988 of *LNCS*, pages 177–191. Springer, 2004.

[55] C. Pasareanu, R. Pelánek, and W. Visser. Test input generation for java containers using state matching. In *Proc. of International Symposium on International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 37–48. ACM, 2006.

[56] R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[57] R. Pelánek. Evaluation of on-the-fly state space reductions. In *Proc. of Mathematical and Engineering Methods in Computer Science (MEMICS'05)*, pages 121–127, 2005.

[58] R. Pelánek. On-the-fly state space reductions. Technical Report FIMU-RS-2005-03, Masaryk University Brno, 2005.

[59] R. Pelánek. Web portal for benchmarking explicit model checkers. Technical Report FIMU-RS-2006-03, Masaryk University Brno, 2006.

[60] R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[61] R. Pelánek. Fighting state space explosion: Review and evaluation. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'08)*, 2008. To appear.

[62] R. Pelánek. Model classifications and automated verification. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'07)*, volume 4916 of *LNCS*, pages 149–163. Springer, 2008.

[63] R. Pelánek. Properties of state spaces and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(5):443–454, 2008.

[64] 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.

[65] R. Pelánek and V. Rosecký. Emma: Explicit model checking manager (tool presentation). In *Proc. of Model Checking Software (SPIN'09)*, volume 5578 of *LNCS*, pages 169–173. Springer, 2009.

[66] R. Pelánek and V. Rosecký. Verification manager: Automating the verification process. Technical Report FIMU-RS-2009-02, Masaryk University Brno, 2009.

[67] R. Pelánek, V. Rosecký, and P. Moravec. Complementarity of error detection techniques. *ENTCS*, 220(2):51–65, 2008.

[68] R. Pelánek, V. Rosecký, and J. Šeděnka. Evaluation of state caching and state compression techniques. Technical Report FIMU-RS-2008-02, Masaryk University Brno, 2008.

[69] R. Pelánek and P. Šimeček. Estimating state space parameters. Technical Report FIMU-RS-2008-01, Masaryk University Brno, 2008.

[70] 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)*, volume 2988 of *LNCS*, pages 487–511, 2004.

[71] J. P. Self and E. G. Mercer. On-the-fly dynamic dead variable analysis. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 113–130. Springer, 2007.

[72] M. Sighireanu and R. Mateescu. Verification of the link layer protocol of the ieee-1394 serial bus (firewire). *Software Tools for Technology Transfer (STTT)*, 2(1):68–88, November 1998.

[73] W. Visser. Memory efficient state storage in SPIN. In *Proc. of SPIN Workshop*, pages 21–35, 1996.

# Chapter 2

# BEEM: Benchmarks for explicit model checkers

In this paper we present BEEM – BEnchmarks for Explicit Model checkers. The benchmark set includes more than 50 parametrized models (together with their correctness properties), which makes it the most comprehensive benchmark set in the area of explicit model checking. Moreover, BEEM is not just a collection of models, it is also accompanied by an comprehensive web portal, which provides detailed information about all models and support for experiments. A specific novel feature of BEEM is the inclusion of information about state space properties.

We use BEEM in several of our subsequent studies and BEEM has also quickly gained popularity in the explicit model checking community. During three years since it was launched, BEEM has been used in more than 30 published papers.

The paper was published in proceedings of SPIN Workshop 2007, extended version of the paper appeared as a technical report:

- R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
- R. Pelánek. Web Portal for Benchmarking Explicit Model Checkers. FIMU-RS-2006-03, 39 pages, 2006.

# BEEM: Benchmarks for Explicit Model Checkers

Radek Pelánek*

Department of Information Technologies, Faculty of Informatics
Masaryk University Brno, Czech Republic
{xpelanek}@fi.muni.cz

**Abstract.** We present Beem — BEnchmarks for Explicit Model checkers. This benchmark set includes more than 50 parametrized models (300 concrete instances) together with their correctness properties (both safety and liveness). The benchmark set is accompanied by an comprehensive web portal, which provides detailed information about all models. The web portal also includes information about state spaces and facilities for selection of models for experiments.

The address of the web portal is `http://anna.fi.muni.cz/models`.

## 1 Introduction

The model checking field underwent a rapid development during last years. Several new, sophisticated techniques have been developed, e.g., symbolic methods, bounded model checking, or automatic abstraction refinement. However, for several important application domains we cannot do much better than the basic explicit model checking approach — brute force exhaustive state space search. This technique is used by several of the most well-known model checkers (e.g., Spin, Murphi). The application scope of the explicit technique has been extended significantly by progress in computer speed and algorithmic improvements and many realistic case studies showed practical usability of the method. Even some of the software model checkers (e.g., Java PathFinder, Zing) are based on the explicit search.

There is also a significant body of research work devoted to the improvement of explicit model checking. Unfortunately, many papers fail to convincingly demonstrate the usefulness of newly presented techniques. In order to perform high quality experimental evaluation, researchers need to have access to:

- tool in which they can implement model checking techniques,
- benchmark set of models which can be used for comparisons.

At the moment, there is a large number of model checking tools (see [4]), but the availability of benchmark sets is rather poor. The aim of this work is to contribute to the progress in this direction. We present Beem — a new benchmark set with a web portal.

* Partially supported by GA ČR grant no. 201/07/P035.

D. Bošnački and S. Edelkamp (Eds.): SPIN 2007, LNCS 4595, pp. 263–267, 2007.

This short paper presents the main rationale and design choices behind Beem. Detailed documentation is given in a technical report [10], which presents description of the modeling language and used models, functionality and realization of the web portal, and an example of an experimental application over the set.

## 2 Experimental Work in Model Checking

In order to support the need for benchmarks, we present an evaluation of experiments in model checking papers. We have used a sample of model checking publications; experiments in each of these publications were classified into one of the following five categories:

**Q1** Random inputs or few toy models.
**Q2** Several toy models (possibly parametrized) or few simple models.
**Q3** Several simple models (possibly parametrized) or one large case study.
**Q4** An exhaustive study of parametrized simple models or several case studies.
**Q5** An exhaustive study with the use of several case studies.

Table 1. presents the quality of experiments in papers from our sample (detailed description of the classification and list of all used papers and their classification is given in [10]). Although the classification is slightly subjective, it is clear from Table 1. that there is nearly no progress in time towards higher quality of used models. This is rather disappointing, because more and more case studies are available. Low experimental standards make it hard to assess newly proposed techniques; the practical impact of many techniques can be quite different from claims made in publications. This obstructs the progress of the research in the field. Clearly, a good benchmark set is missing.

The need for benchmarking, better experiments, and thorough evaluation of tools and algorithms is well recognized, e.g., experimentation is a key part of Hoare's proposal for a "Grand Challenge of Verified Software" [6]. There is also significant interest in benchmarks in the model checking community (see e.g., Corbett [3], Avrunin et al. [5], Atiya et al. [1], Jones et al. [8]). Nevertheless, the progress up to date has been rather slow. The main obstacle in developing

**Table 1.** Quality of experiments reported in model checking papers. We have used a sample of 80 publications which are concerned with explicit model checking and contain an experimental section (for details see [10]). For each quality category, we report number of published papers in years 1994-2006.

| | 1994 | 1995 | 1996 | 1997 | 1998 | 1999 | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Q1 | - | - | 1 | 1 | 1 | 1 | 1 | 3 | 2 | 4 | 2 | 1 | 1 |
| Q2 | - | - | 3 | 3 | 2 | 3 | 3 | 1 | 2 | 2 | 2 | 1 | - |
| Q3 | - | 2 | 1 | 3 | 1 | 2 | 2 | 1 | 3 | 2 | 2 | 4 | 1 |
| Q4 | 1 | - | - | - | 1 | - | 1 | 4 | 1 | 1 | 2 | - | 2 |
| Q5 | - | - | - | 1 | - | - | - | - | 1 | - | 1 | - | - |

model checking benchmarks is the absence of a common modeling language — each model checking tool is tailored towards its own modeling language and even verification results over the same example are often incomparable.

Although the development of benchmarks is difficult and the model checking community will probably never have a universal benchmark set, we should try to build benchmarks as applicable as possible and steadily improve our experimental analysis. This is the goal of this work.

## 3 BEEM

**Modeling Language.** Models are implemented in a *low-level modeling language* based on communicating extended finite state machines (DVE language, see [10] for syntax and semantics). The adoption of a low-level language makes the manual specification of models hard, but it has several advantages. The language has a simple and straightforward semantics; it is not difficult to write own parser and state generator. Models can be automatically translated into other modeling languages — at the moment, the benchmark set includes also *Promela* models which were automatically generated from DVE sources.

**Models and Properties.** 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). In order to make the set organized, models are *classified* into different types and categories. The benchmark set is *large* and still growing (at the moment it contains 57 parametrized models with 300 specified instances). *Source codes* of all models are publicly available. Models are briefly described and include *pointers to sources* (e.g., paper describing the case study), i.e., BEEMalso serves as an information portal.

The benchmark set includes also *correctness properties* of models. Safety properties are expressed as reachability of a predicate, liveness properties are expressed in Linear temporal logic. Since an important part of model checking is error detection, the benchmark set includes also *models with errors* (presence of an error is a parameter of a model).

**Tool Support.** The modeling language is supported by an *extensible model checking environment* — The Distributed Verification Environment (DiVinE) [2]. DiVinE is both a model checking tool and a open and extensible library for a development of model checking algorithms. Researchers can use this extensible environment to implement their own algorithms, easily perform experiments over the benchmark set, and directly compare with other algorithms in DiVinE. Promela models can be used for comparison with the well-known model checker Spin [7].

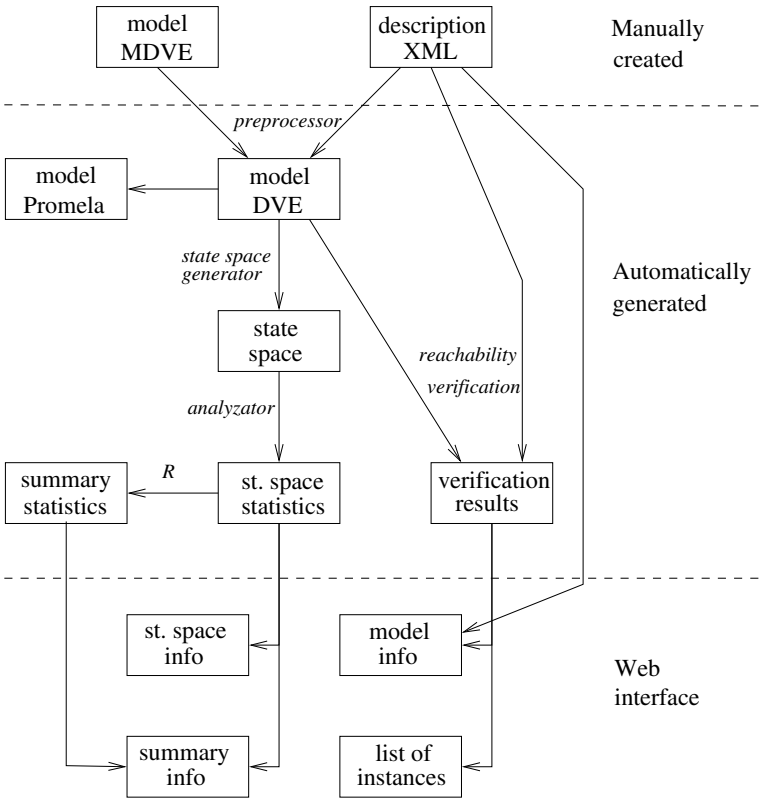



**Fig. 1.** Overview of the realization of the web portal. The user provides two files: parametrized model and its description. All other information is automatically generated.

**Web Portal.** The benchmark set is accompanied by an comprehensive web portal, accessible at `http://anna.fi.muni.cz/models`, which *facilitates the experimental work*. The web provides (see Fig 1. for overview of realization):

- presentation of all information about models, their parameters, and correctness properties,
- detailed information about properties of state spaces of models [9] including summary information,
- verification results,
- web form for selection of suitable model instances according to a given criteria,
- instance generator, which can generate both DVE models and Promela models for given parameter values.

All data can be downloaded. Since model descriptions are systematic (XML file), it is easy to write own scripts for manipulation with models and automation of experiments.

## 4 Summary

The aim of this paper is not to present "the ultimate benchmark set" but rather:

- to provide a ready-made set for those who want to compare different model checking techniques and to facilitate experimental research,
- to encourage higher standards in model checking experiments,
- to stimulate the discussion about benchmarks in the model checking community.

Detailed description of the benchmarks set, example of an experimental application, and direction for the future work can be found in the technical report [10].

**Acknowledgement.** I thank Pavel Krčál and to members of the DiVinE group, particularly Ivana Černá, Pavel Šimeček and Jiří Barnat, for collaboration, discussions, and feedback.

## References

1. Atiya, D.A., Catano, N., Lüettgen, G.: Towards a benchmark for model checkers of asynchronous concurrent systems. In: Fifth International Workshop on Automated Verification of Critical Systems: AVOCs, University of Warwick, United Kingdom (September 12–13, 2005)
2. Barnat, J., Brim, L., Černá, I., Moravec, P., Rockai, P., Šimeček, P.: Divine - a tool for distributed verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006), `http://anna.fi.muni.cz/divine`
3. Corbett, J.C.: Evaluating deadlock detection methods for concurrent software. IEEE Trans. Softw. Eng. 22(3), 161–180 (1996)
4. Crhová, J., Krčál, P., Strejček, J., Šafránek, D., Šimeček, P.: Yahoda: the database of verification tools. In: Proc. of TOOLSDAY affiliated to CONCUR 2002, FI MU report series (2002) Accessible at `http://anna.fi.muni.cz/yahoda/`
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Benchmarking finite-state verifiers. International Journal on Software Tools for Technology Transfer (STTT) 2(4), 317–320 (2000)
6. Hoare, T.: The ideal of verified software. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 5–16. Springer, Heidelberg (2006)
7. Holzmann, G.J.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading, Massachusetts (2003)
8. Jones, M., Mercer, E., Bao, T., Kumar, R., Lamborn, P.: Benchmarking explicit state parallel model checkers. In: Proc. of Workshop on Parallel and Distributed Model Checking (PDMC'03). ENTCS, vol. 89, Elsevier, Amsterdam (2003)
9. Pelánek, R.: Typical structural properties of state spaces. In: Graf, S., Mounier, L. (eds.) Proc. of SPIN Workshop. LNCS, vol. 2989, pp. 5–22. Springer, Heidelberg (2004)
10. Pelánek, R.: Web portal for benchmarking explicit model checkers. Technical Report FIMU-RS-2006-03, Masaryk University Brno (2006)

# Chapter 3

# Properties of state spaces and their applications

State spaces are usually treated as directed graphs without any specific features. However, state spaces are generated from a rather small symbolic description (a model). Thus it is conceivable that state spaces are not just arbitrary complicated directed graphs. Motivated by this simple argument, we study the following questions: What are typical properties of state spaces? What do state spaces have in common? Can state spaces be modeled by random graphs? How can we apply properties of state spaces? Can we exploit these typical properties to traverse a state space more efficiently? Are state spaces similar to such an extent that it does not matter which models we choose for benchmarking our algorithms?

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 discuss consequences of these results for model checking experiments and we point out how to exploit typical properties in practical model checking algorithms.

This paper was published in the International Journal on Software Tools for Technology Transfer (STTT) in 2008, preliminary version of this research was reported in SPIN Workshop in 2004:

- R. Pelánek. Properties of state spaces and their applications. *International Journal on Software Tools for Technology Transfer*, 10(5):443-454, 2008.
- R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

Int J Softw Tools Technol Transfer (2008) 10:443–454
DOI 10.1007/s10009-008-0070-5

REGULAR CONTRIBUTION

# Properties of state spaces and their applications

**Radek Pelánek**

Published online: 6 June 2008


**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

R. Pelánek was partially supported by GA ČR grant no. 201/07/P035.

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

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 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 [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 Sect. 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 Sect. 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.

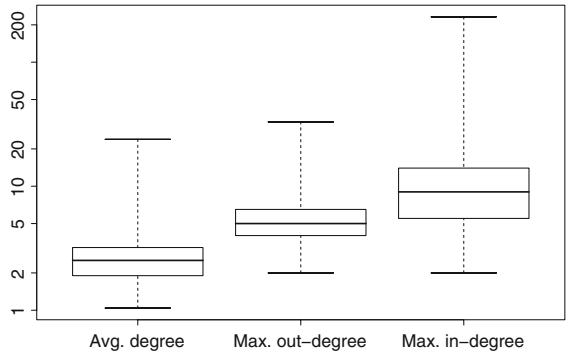



**Fig. 1** Degree statistics. Values are displayed with the boxplot method. The *upper and lower* lines are maximum and minimum values, the middle line is a median, the other two are quartiles. Note the logarithmic $y$-axis

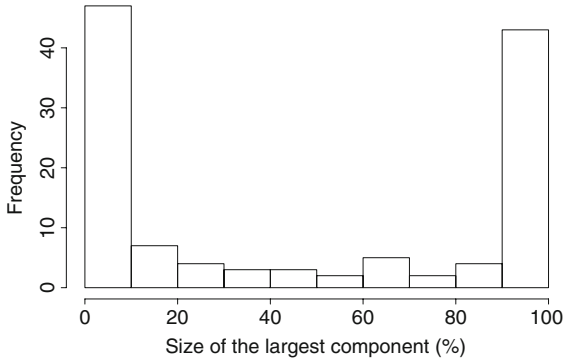



**Fig. 2** Histogram of sizes of the largest SCC component in a state space

### 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 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 1–4 places. Distribution of these 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' [27,28]. 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.

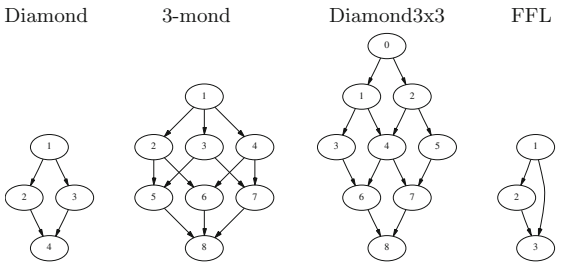



**Fig. 3** Illustrations of motifs

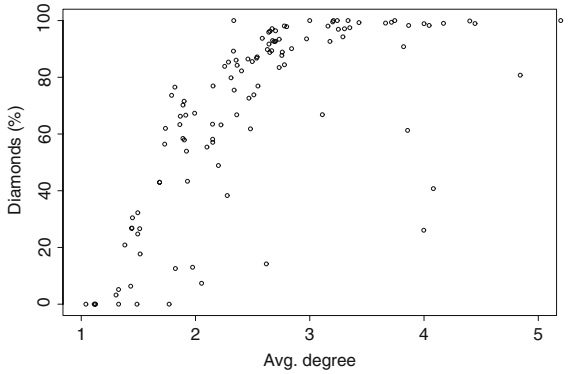



**Fig. 4** Relationship between occurrence of diamonds and the average degree. The occurrence of diamonds is reported as a ratio of the number of states which are roots of some diamond to all states

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

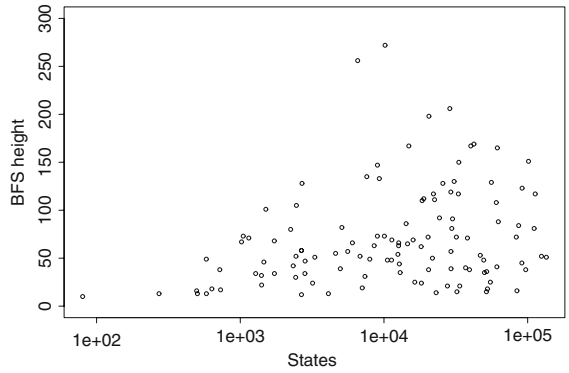



**Fig. 5** The BFS height plotted against the size of the state space. Note the logarithmic $x$-axis. Three examples have BFS height larger than 300

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.

### 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*. Figure 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. Figure 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 Sect. 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 high and it rapidly

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

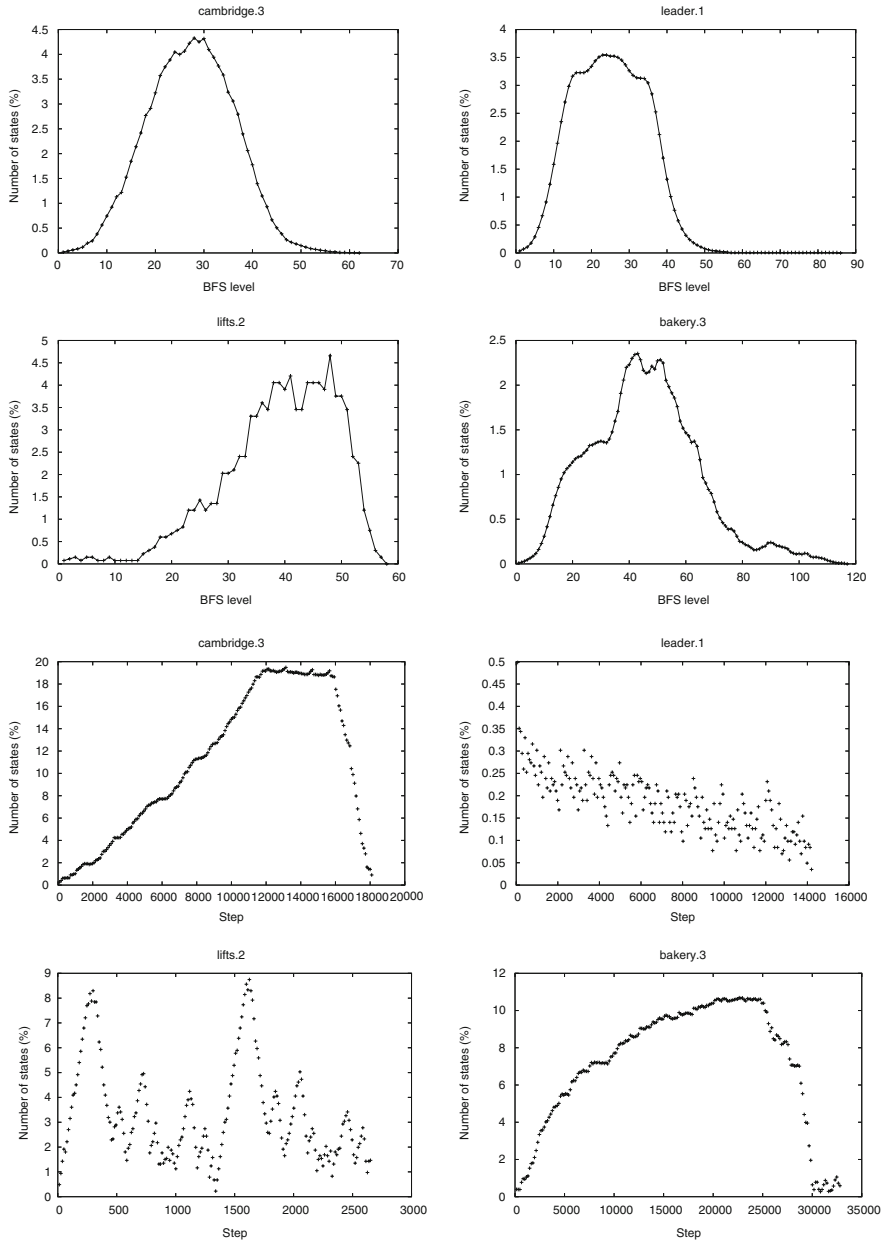



**Fig. 6** BFS level graphs (*first four*) and stack graphs (*second four*)

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. Figure 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

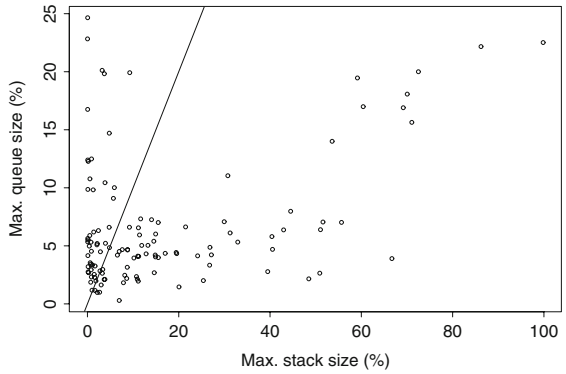



**Fig. 7** A comparison of maximal queue and stack sizes expressed as percentages of the state space size

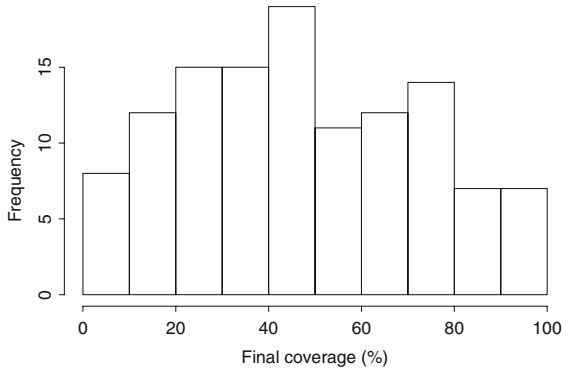



**Fig. 8** Histogram of random walk coverage after number of steps equal to 10 times the size of the state space. Frequency means the number of state spaces for which the final coverage was in the given interval

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.

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.

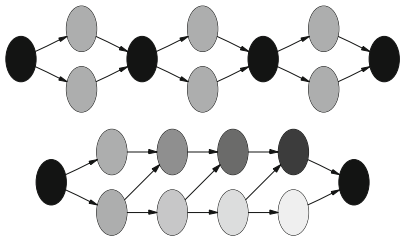


**Fig. 9** Graphs with similar properties but different random walk coverage. The color correspond to probability of a visit by a random walk; darker vertices have higher change of a visit.

Finally, we provide a specific example which demonstrates how the use of random graphs can obfuscate experimental analysis. Figure 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].

- 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 Sect. 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.

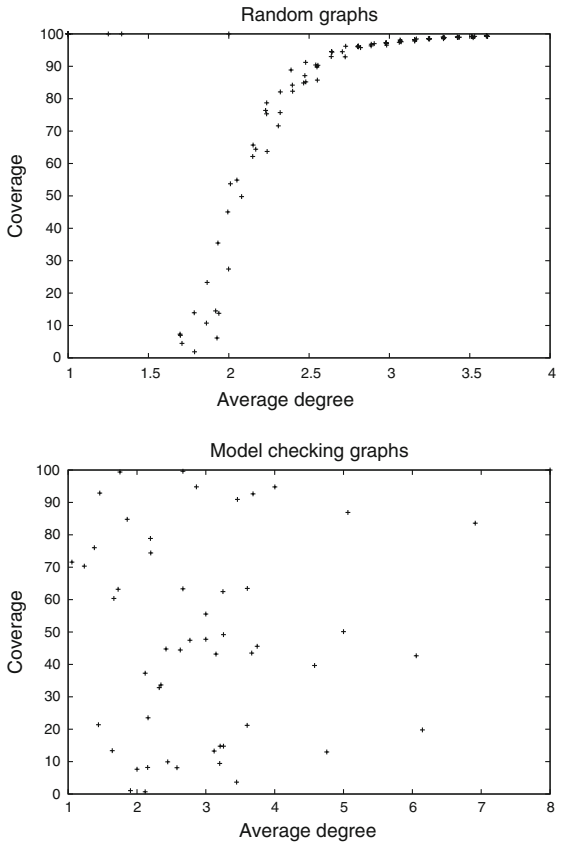



**Fig. 10** Correlation between the average degree and random walk coverage for random graphs and model checking graphs

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

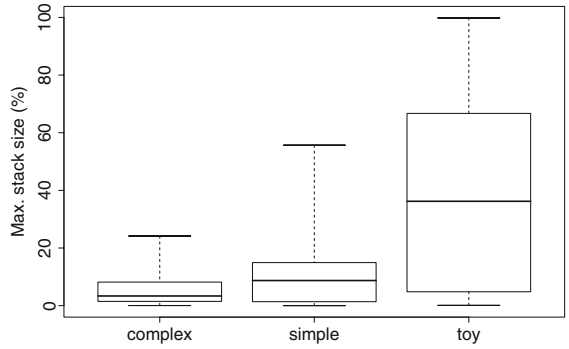



**Fig. 11** The maximal stack size (given in percents of the state space size) during DFS. Results are displayed with the boxplot method (see Fig. 1 for explanation)

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 Sect. 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].

Breadth-first search 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, pp. 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 Sect. 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 Sect. 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 Sect. 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 Sect. 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.

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

30. Pelánek, R.: Reduction and Abstraction Techniques for Model Checking. PhD thesis, Faculty of Informatics, Masaryk University, Brno (2006)
31. Pelánek, R.: Beem: Benchmarks for explicit model checkers. In: Proceeding of SPIN Workshop, vol. 4595 of *LNCS*, pp. 263–267. Springer, Heidelberg (2007)
32. Pelánek, R.: Model classifications and automated verification. In: Proceeding of Formal Methods for Industrial Critical Systems (FMICS'07) (2007). (To appear)
33. Pelánek, R., Hanžl, T., Černá, I., Brim, L.: Enhancing random walk state space exploration. In: Proceeding of Formal Methods for Industrial Critical Systems (FMICS'05), pp. 98–105. ACM Press, New York (2005)
34. Pelánek, R., Šimeček, P.: Estimating state space parameters. Technical Report FIMU-RS-2008-01, Masaryk University Brno (2008)
35. Seppi, K., Jones, M., Lamborn, P.: Guided model checking with a bayesian meta-heuristic. In: Proceeding of Application of Concurrency to System Design (ACSD'04), p. 217. IEEE Computer Society, New York (2004)
36. Sivaraj, H., Gopalakrishnan, G.: Random walk based heuristic algorithms for distributed memory model checking. In: Proceeding of Parallel and Distributed Model Checking (PDMC'03), vol. 89 of *ENTCS* (2003)
37. Stern, U.: Algorithmic Techniques in Verification by Explicit State Enumeration. PhD thesis, Technical University of Munich (1997)
38. Stern, U., Dill, D.L.: Using magnatic disk instead of main memory in the Murphi verifier. In: Proceeding Computer Aided Verification (CAV 1998), vol. 1427 of *LNCS*, pp. 172–183 (1998)
39. Tronci, E., Penna, G.D., Intrigila, B., Venturini, M.: A probabilistic approach to automatic verification of concurrent systems. In: Proceeding Asia-Pacific Software Engineering Conference (APSEC 2001), pp. 317–324. IEEE Computer Society, New York (2001)
40. Tronci, E., Penna, G.D., Intrigila, B., Zilli, M.V.: Exploiting transition locality in automatic verification. In: Proceeding Correct Hardware Design and Verification Methods (CHARME 2001), vol. 2144 of *LNCS*, pp. 259–274 (2001)
41. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Kozen D. (eds.) Proceeding of Logic in Computer Science (LICS '86), pp. 332–344. IEEE Computer Society Press, New York (1986)

# Chapter 4

# Estimating State Space Parameters

In this paper we introduce the problem of estimation of state space parameters, argue that it is an interesting and practically relevant problem, and study several simple estimation techniques. Particularly, we focus on estimation of the number of reachable states. Such estimation can be very useful in several ways, e.g., for tuning model checking algorithms, for automating the verification process, as a practical information for users, or as a criterion for selection of models for experiments.

Our techniques are based on our insight of typical properties of state spaces and on the understanding of random walk behaviour (previous two papers). We study techniques based on sampling of the state space and techniques that employ data mining techniques (classification trees, neural networks) over parameters of breadth-first search. We show that even through the studied techniques are not able to produce exact estimates, it is possible to obtain usable information.

This paper was published as a work-in-progress paper at International Workshop on Parallel and Distributed Methods in verifiCation (PDMC) in 2008 and as a technical report [69]. The author of the thesis is one of two coauthors of the paper and has done major part of both experiments and writing.

# Estimating State Space Parameters

Radek Pelánek[1] Pavel Šimeček [2]

*Faculty of Informatics*
*Masaryk University Brno, Czech Republic*

**Abstract**

We introduce the problem of estimation of state space parameters, argue that it is an interesting and practically relevant problem, and study several simple estimation techniques. Particularly, we focus on estimation of the number of reachable states. We study techniques based on sampling of the state space and techniques that employ data mining techniques (classification trees, neural networks) over parameters of breadth-first search. We show that even through the studied techniques are not able to produce exact estimates, it is possible to obtain useful information about a state space by sampling and to use this information to automate the verification process.

*Keywords:* explicit model checking, state space parameters, state space size, data mining

## 1 Introduction

Explicit model checking is a state-of-the-art technique for verification of asynchronous concurrent systems. This technique is based on construction of a reachable part of a model state space. In this work we are concerned with techniques for estimation of state space parameters, particularly with estimation of the number of reachable states. Estimation of state space parameters is not a typical problem in verification. Nevertheless, we argue that it has a good motivation — there are even several independent reasons to study this problem.

*Tuning of model checking algorithms.* Estimation of state space parameters can be useful for tuning model checking algorithms, for example to select a suitable size of hash table or cache [10], [7] or to choose a proper I/O efficient graph exploration algorithm [11,4].

Parameter estimations are particularly useful in the distributed environment. If we do not use enough workstations, then the computation runs out of memory. If we use too many workstations, then the performance deteriorates due to unnecessary communication overhead [14]. Estimate of the number of reachable states

[1] Partially supported by GA ČR grant no. 201/07/P035.
[2] Partially supported by GA ČR grant no. 201/06/1338.

*This paper is electronically published in*
*Electronic Notes in Theoretical Computer Science*
*URL:* `www.elsevier.nl/locate/entcs`

can help us choose the suitable number of workstations for a given model. Parameter estimations can also be used to select an appropriate reduction technique or to choose an algorithm from several competing ones [18]. Another application in a distributed environment is the selection of a suitable cycle detection algorithm. There are many distributed cycle detection algorithms which can be used for LTL verification of large models (see [2] for an overview); each of these algorithms is suitable for a certain class of graphs.

*Information for users.* The run-time of a model checker is often long. It would be very convenient for the user to have the estimation of the remaining time. Such an estimation is not just of practical interest, it has also psychological advantages — if users are informed about the remaining time, they are more willing to wait [13].

*Selection of models for experiments.* Experimentalists need a lot of models in order to convincingly show advantages of novel techniques and to properly evaluate known techniques. It is often desirable to have models with a specific state space size. Consider, for example, experiments with distributed algorithms and algorithms which are concerned with memory consumption (state space caching, external memory algorithms). For such experiments, it is needed to have models with a state space size around the limits of a single machine. It is laborious and computationally intensive to find models with specific state space size. Techniques for estimating the reachable state space size can significantly speed up this effort and thus lead to an improvement in experimental standards.

Our aims and contributions in this work are the following:

(i) Study of simple techniques for estimation of state space parameters. Particularly, we study techniques based on sampling of the state space and techniques that employ data mining techniques (classification trees, neural networks) over parameters of breadth-first search.

(ii) Evaluation and comparison of techniques. We evaluate these techniques over a large benchmarks set and compare their performance.

(iii) Summary of techniques potential. We formulate what is reasonable to expect from studied techniques.

Note that most of the techniques presented in this work could be further improved by careful tuning and optimisation, but that is not the aim of the presented work.

**Related Work** The problem of estimating state space parameters did not receive much attention so far. Most of the relevant work deals only with techniques based on analysis of model structure and was not very successful. Watson and Desrochers [24] proposed a technique for estimation of the size of Petri nets based on a static analysis of the net. Estimation is quite accurate, but it is limited to a specific type of models. Chamillard et al. [6] proposed technique based on the linear regression measures given by static metrics of the model. However, the technique does not work very well. Peng and Makki [20] described a technique for comparison of two state space sizes; their technique is independent of the specification language. Sahoo et al. [22] use sampling of the state space to decide which BDD based reachability technique is the best for a given model.

## 2 Background

In this section we discuss the state space parameters that we are interested in. We also describe the experimental data and the sampling techniques that we have used for evaluation.

### *2.1 State Space Parameters*

We 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 $v_0$. Vertices are states of the model, edges represent valid transitions between states. For our purposes we ignore any labeling of states or edges. We are concerned only with the reachable part of the state space.

An *average degree* of $G$ is the ratio $|E|/|V|$. 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. Let us consider the breadth-first search (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 $u$.

### *2.2 Experimental Environment*

Reported techniques are implemented in the DiVinE environment [3]. In order to evaluate estimation techniques, we perform experiments over the BEEM benchmark set [17]. The web portal of the benchmark set contains all the used models and it also presents state space properties of models.

For the evaluation we use 160 models from the BEEM set. The state space size of used models is between 20,000 and 20,000,000 states (note that for the evaluation we use only models for which we are able generate the full state space). Used models are divided randomly into a training set and a testing set, each of them contains 80 models. Reported estimation techniques are trained (calibrated) on the training set and their success is measured over the testing set.

### *2.3 Sampling Techniques*

All our estimation techniques are based on the following approach: we take a sample of the state space and according to the information collected by this sample we do the estimation. We use mainly the classical search techniques to make samples:

- breadth-first search (BFS) sample: first $s$ states of BFS,
- depth-first search (DFS) sample: first $s$ states of DFS (the size of stack was limited to 10,000 states during the search),
- random walk (RW) sample of size $s$: we run random walks through the state space until we find $s$ states.

Beside these classical techniques, we also introduce a special kind of random walk. The aim of this technique is to traverse a small, but representative portion of

the state space. The technique uses a hash function to decide which states should be stored and explored, therefore we denote it as *Hash-RW*. Unlike memoryless random walk, it uses a state repository to recognize visited states. To increase the probability that the walk visits states lying on various paths in various distances from the initial state, the search traverses through up to $C$ states in a parallel fashion, where $C$ is a given constant number.

In essence, this random walk works in a similar way as BFS. BFS works by levels: after generation of level $i$, level $i+1$ is generated using the successor function applied to states in level $i$. Hash-RW works in the same way, but it tries to traverse only a subset of states of size $C$ from each level. The subset of states to traverse is determined by a special decision function. The function computes a hash of a given state and if it is smaller than a certain limit, it decides to store the state to the repository and the exploration queue. The limit is updated after computation of all successors of the last level to keep number of states in the next level close to $C$. The aim of this decision function is to reduce the number of traversed back level edges. In some cases the basic version of Hash-RW explores large portion of a state space. Therefore, we use additional finishing conditions; these conditions are described together with a specific usage of Hash-RW in Section 4.

## 3 Estimation of the Number of Reachable States

In this section we discuss techniques for estimating the number of reachable states of a given model. Our preliminary experiments showed that simple techniques are not able to produce accurate absolute estimates. Therefore, we classify models in three classes and try to estimate these classes. In the following we introduce the classification, discuss two approaches for the estimation of the classification (one based on sampling and one based on BFS parameters), and then we combine techniques and compare them.

### *3.1 Classification*

Using simple techniques, it seems impossible to estimate the number of reachable states with a high accuracy. However, for practical applications this is not necessary. It is often sufficient to have an 'order of magnitude' estimate. Therefore, we introduce three classes, which produce the order of magnitude estimate and study techniques for estimating the classification. This approach also simplifies evaluation and comparison of estimation techniques.

The classes are not defined in absolute terms (by number of states), but rather relatively: we suppose that a sample of a state space is generated and we define the classes with respect to how many times the total number of reachable states is larger than the sample. We have two reasons to adopt this approach. Firstly, it enables us to do meaningful experiments with estimation techniques on state spaces of different sizes. Secondly, the speed of state generation significantly differs for different models [17], i.e., the relative estimate is more useful for estimating the model checker run time.

For our experiments, we have used three classes which we believe have practical substantiation. Let $R$ be the ratio of the total number of reachable states to the

number of the taken sample:

- Class 1, $1 \leq R < 4$. Models in this class can be verified. It should be sufficient to just wait or to slightly tune the model checkers parameters (e.g., use a more appropriate hash table size, turn on a reduction technique).
- Class 2, $4 \leq R < 32$. To check a model in this class, it is necessary to use an aggressive reduction technique and/or distributed computation. It should be possible to verify the model as it is, but it may be a bit complicated.
- Class 3, $32 \leq R$. Models in this class seem out of reach (if the sample is large). It is probably necessary to apply abstraction to these models.

For calibration and evaluation of class estimation techniques we use training and test data of the following form: input = model + sample size, output = class. We have used several sample sizes for each model; both training and testing set contained approximately 320 inputs. Note that we work only with models for which we know the size of the state space, i.e., we know the correct class.

For each estimation technique we report: *success rate* — the ratio of inputs correctly classified, *major mistakes* — the ratio of inputs classified totally incorrectly (i.e., class 1 classified as class 3 or vice versa).

### *3.2 Techniques Based on Sampling*

A straightforward approach for estimating the number of reachable states is the following:

(i) Take two independent random samples of size $s$ of reachable states.

(ii) Let the number of states which occur in both samples be $x$.

(iii) Estimated number of reachable states is $s/x$ (in other words, the ratio $x/s$ is expected to be close to $s/n$, where $n$ is the number of reachable states).

However, this straightforward approach cannot be realized — without actually generating all reachable states, there is no way to obtain two independent random samples of reachable states. The straightforward way to obtain a random reachable state is to use random walk from the initial state. However, the chance of picking a state by this method is far from uniform [19], i.e., this method can not be used to obtain a random sample.

Nevertheless, the outlined method can be used even if samples are not completely independent and random. We use the BFS, DFS, and RW samples (as described in Section 2). Results in Fig. 1. show the relation between the ratio $x/s$ and the correct estimate $s/n$. Results are show for all three combinations of the three sampling methods, these results are obtained on the training set. Based on the data from the training set we identify decision values which are used for the classification of the testing set (more specifically, the decision values were identified automatically with the use of R software [21]). For example, for the DFS x RW sampling method we use the following values:

- If $x/s >= 0.401$ then output 'C1'.
- If $x/s < 0.401 \wedge x/s >= 0.096$ then output 'C2'.

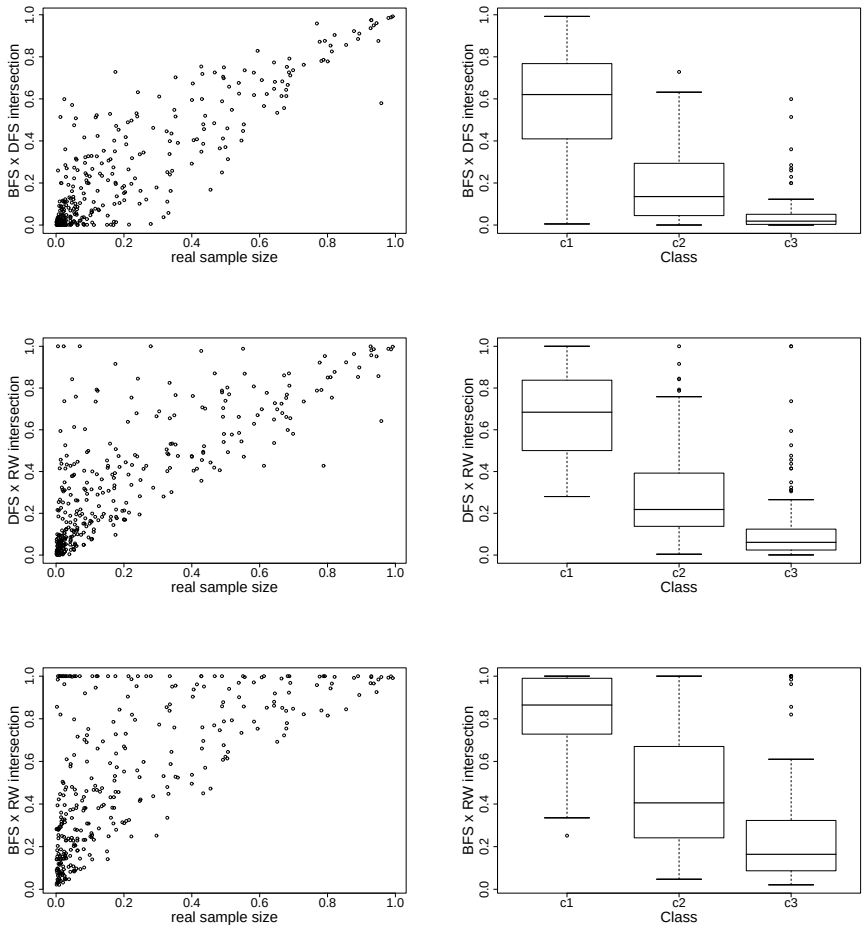



Fig. 1. Figures on the left show scatter plot of the ratio of sample size to number of reachable states and the relative size of intersection of two samples. Figures on the right show relative sizes of intersections for each class using the boxplot method (the upper and lower lines are maximum and minimum values, the middle line is a median, the other two are quartiles; circles mark outliers).

- If $x/s < 0.096$ then output 'C3'.

Using this classification method, we classify the data in the testing set. Table 1. gives results. It shows that the best results are obtained using the intersection of DFS and RW samples — success rate is 72%.

### *3.3 Techniques Based on BFS Parameters*

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. Usually this graph has a 'bell-like' shape [16] (see Fig. 2. for several such graphs; more BFS level graphs are on the BEEM web page [17]). Our goal is to use the knowledge of this typical behaviour of breadth-first search and to estimate the size of the state space based on the first $k$ BFS levels ($k$ is determined by the size of a sample).

| | BFS x DFS | | | | DFS x RW | | | | BFS x RW | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | E1 | E2 | E3 | | E1 | E2 | E3 | | E1 | E2 | E3 |
| C1 | 31% | 6% | 0% | C1 | 34% | 2% | 0% | C1 | 25% | 11% | 0% |
| C2 | 8% | 19% | 12% | C2 | 8% | 29% | 3% | C2 | 4% | 31% | 4% |
| C3 | 3% | 9% | 12% | C3 | 3% | 12% | 9% | C3 | 3% | 10% | 11% |
| success rate 62% | | | | success rate 72% | | | | success rate 67% | | | |
| major mistakes 3% | | | | major mistakes 3% | | | | major mistakes 3% | | | |

Table 1
Results of estimation techniques based on sampling. Rows (C1, C2, C3) are correct classifications, columns (E1, E2, E3) are estimated classifications. Results are given as percents, numbers are rounded.

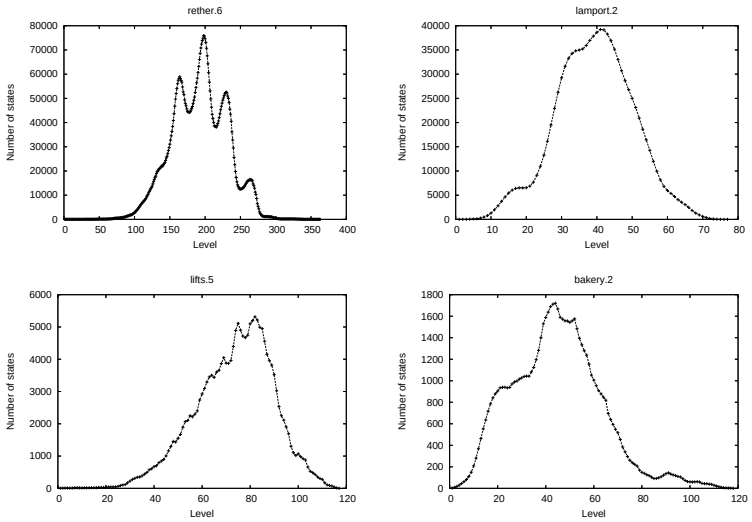



Fig. 2. BFS level graphs. For simple models the curve is smooth and bell-like, for more complex models the graph is a bit ragged.

### *3.3.1 Estimation by Human*

As the first step, we have performed experiments with classification by human (one of the authors). The human is shown a graph of first $k$ levels (see Fig. 3. for examples) and based on this information estimates the classification. Results in Table 2. suggest that a human can perform classification reasonably well. This is an interesting observation — it suggests that the BFS level graph may be a useful output of a model checker during the state space generation. Note that this is a rather cheap operation both in terms of computation overhead and implementation effort.

Based on the experience with human estimation, we choose the following parameters as inputs for automatic classification methods (let $k$ be the number of BFS levels in the sample):

- LA — ratio of size of the last explored BFS level to the average size of the first

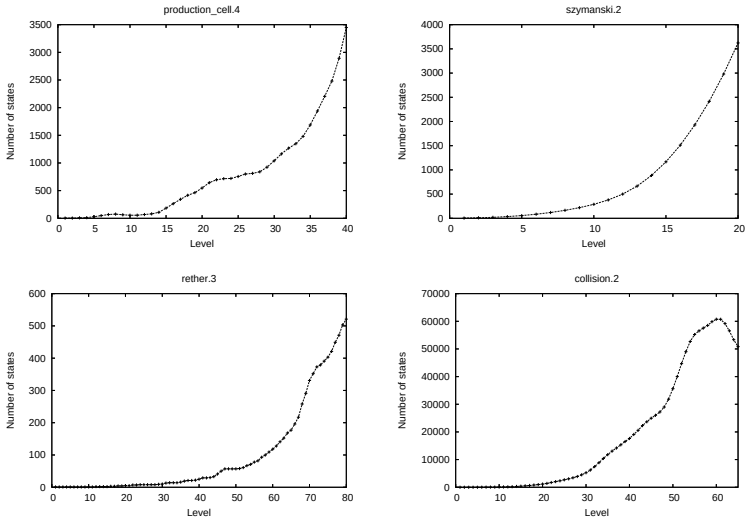



Fig. 3. Examples of partial BFS level graphs as used for human classification.

$k$ levels;

- LM — ratio of size of the last level to the maximal size of a level in the first $k$ levels;
- INF (inflexion) — boolean value, zero means that the difference of sizes of consecutive levels is increasing (up to $k$), i.e., there is no 'inflection point';
- HE (height estimate) — the ratio of $k$ and an estimated BFS height; as an estimate of the BFS height we use the median height from the training set of models of the corresponding type (see Section 4.2).

### *3.3.2 Classification Tree*

Classification tree [5] is a data mining technique used to predict membership of cases in the classes of a categorical dependent variable from their measurements on predictor variables. Classification tree is built through a binary recursive partitioning. Splitting conditions are determined automatically by an analysis of the training data.

Fig. 4. shows a classification tree constructed on our training data (the tree was constructed using R software [21]). Prediction results for the test data are given in Table 2.

### *3.3.3 Neural Network*

Neural network is a machine-learning technique that simulates a network of communicating nerve cells. The network is a weighted graph, the learning is accomplished by modification of weights of edges. This process can be automated using a suitable learning algorithm.

We use a neural network with:

- 4 input neurons (parameters LA, LM, INF, and HE),

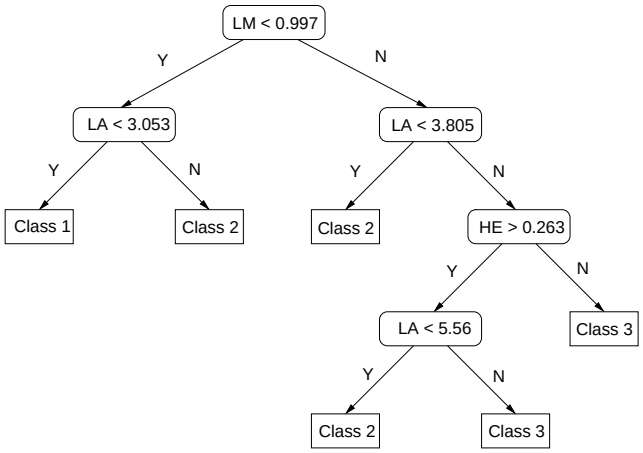



Fig. 4. Classification tree based on BFS parameters.

| | human | | | | classification tree | | | | neural net | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | E1 | E2 | E3 | | E1 | E2 | E3 | | E1 | E2 | E3 |
| C1 | 21% | 18% | 2% | C1 | 16% | 20% | 1% | C1 | 23% | 13% | 1% |
| C2 | 2% | 18% | 13% | C2 | 2% | 24% | 13% | C2 | 7% | 25% | 7% |
| C3 | 3% | 6% | 17% | C3 | 0% | 10% | 14% | C3 | 2% | 14% | 7% |
| success rate 56% | | | | success rate 54% | | | | success rate 55% | | | |
| major mistakes 5% | | | | major mistakes 1% | | | | major mistakes 3% | | | |

Table 2
Results of estimation techniques based on BFS parameters on on the test data. Rows (C1, C2, C3) are correct classifications, columns (E1, E2, E3) are estimated classifications. Results are given as percents, numbers are rounded.

- 4 hidden neurons,
- 3 output neurons (one for each class).

For implementation we use FANN library [15]. The network is trained on the training data; we use the default learning algorithm of the FANN library. Results for the test data are in Table 2.

### *3.4 Combinations and Comparison*

Finally, we combine estimates produced by the five above presented automated techniques (three sampling techniques, classification tree, and neural network). The combined estimate is obtained in the following way:

- If four techniques agree on the estimate, we output the given class.
- If the estimates are divided between two neighbouring classes, we output 'undecided estimate' (E12, E23).
- Otherwise, we output 'don't know' (DN).

| | E1 | E12 | E2 | E23 | E3 | DN |
|---|---|---|---|---|---|---|
| C1 | 18% | 15% | 2% | 0% | 0% | 1% |
| C2 | 1% | 4% | 19% | 7% | 0% | 8% |
| C3 | 0% | 1% | 4% | 9% | 5% | 5% |

Table 3
Combined estimations from the five presented automated techniques. 'Exy' means that the estimate is between classes x and y, DN means don't know.

Table 3. presents results obtained in this way. We see that there are no major mistakes, the number of misses is low (7%), and at the same time the number of undecided and don't know results is reasonable.

Let us compare the studied techniques. Better results can be achieved using the sampling approach: 70% success rate compared to 55% success rate of the BFS based techniques. However, the sampling approach is less practical: it requires at least two samples of the state space and it cannot be easily used on-the-fly (during the state space generation). Techniques based on BFS parameters are less precise, but they can be used very easily during the BFS traversal of the state space — after the traversal of each BFS level we just plug the data into the prefabricated classification tree or neural network. Moreover, we suppose that it should be possible to further improve BFS based techniques by considering more parameters for decision and by incorporating domain specific information (i.e., by training techniques on similar data as they will be applied to).

## 4 Estimation of Other Parameters

In this section we study techniques for estimation of some other state space parameters: the average degree, the BFS height, the number of back level edges, and the size of the largest SCC.

### *4.1 Average Degree*

The average degree usually corresponds to the amount of non-determinism in the system, which substantially influences usefulness of partial order reduction [9]. Average degree can also be understood as a quotient of the number of edges and the number of states. Since a complexity of many graph algorithms depends on a number of edges, estimation of graph edges amount is crucial.

We found out that the vertex degree is almost evenly spread among all vertices of the graph. Therefore, it should be possible to estimate the average degree from quite a small sample of the state space. Fig. 5. shows estimation of the average vertex degree gained by exploration of 5% of the state space. The figure shows results computed by three sampling techniques: BFS, Hash-RW (with $C = 500$) and a simple random walk. For each technique we moreover compute ratios of an estimate and a real value of an average degree and study their distribution. The best results are provided by Hash-RW (standard deviation of the ratio is 0.17). BFS also provides good results (standard deviation is 0.23). Estimates produced by the random walk are poor (standard deviation is 0.44).

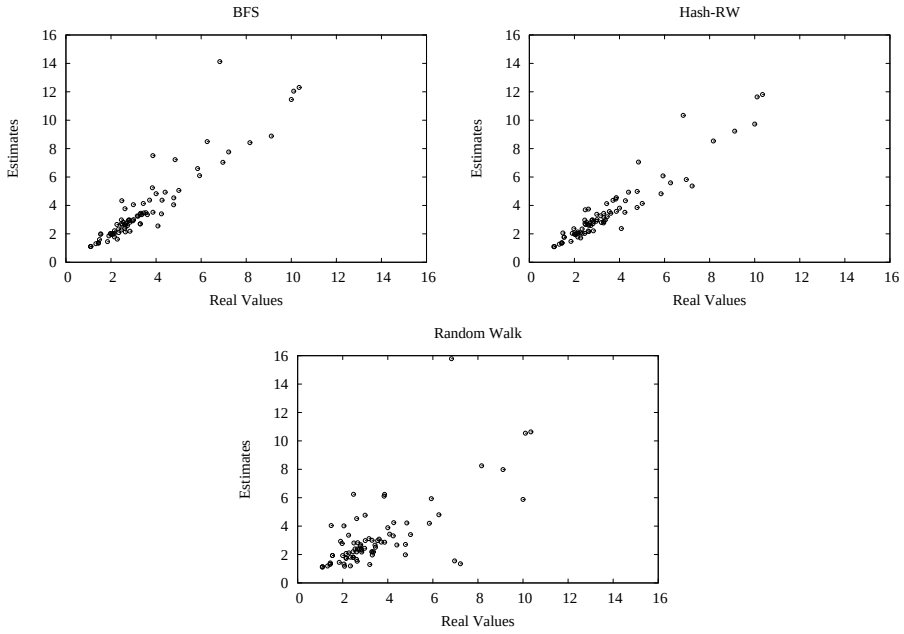



Fig. 5. Graphs show an average degree from the whole state space (Real value) and an average degree from a given sample (Estimate).

We conclude that average degree can be estimated quite easily, nevertheless it matters which technique is used for sampling.

### *4.2 BFS Height*

Estimation of BFS height can be used for estimation of state space size (see Section 3.3). It can also be used to tune several verification techniques: setting depth limit for random walk search [19] and explicit bounded search [12] ; or setting parameters for techniques using stratified caching [8].

Models in our benchmark have BFS heights mostly between 20 and 600. This interval can be further specified if we restrict to a certain type of models. Fig. 6. shows BFS heights of models according to their type (in Section 3.3 we use median values as BFS height estimates).

We provide also an estimation technique based on sampling. First, we reduce BFS height estimation to the estimation of the largest BFS level index. Since we expect bell-like shape of the BFS level graph (see Section 3.3), the largest level has an index equal approximately to the half of the BFS height of a state space. Hence, we can estimate the index of the largest level and multiply it by two.

To identify the index of the largest level, we use Hash-RW with a special finishing condition. The basic idea of the finishing condition exploits estimation of BFS level widths from Hash-RW level widths and numbers of Hash-RW levels successors. While absolute values of BFS level sizes estimates are quite inaccurate, their relative sizes are preserved — if real BFS level sizes are growing with higher index, estimated sizes are usually also growing and if real sizes are falling, estimated sizes are also

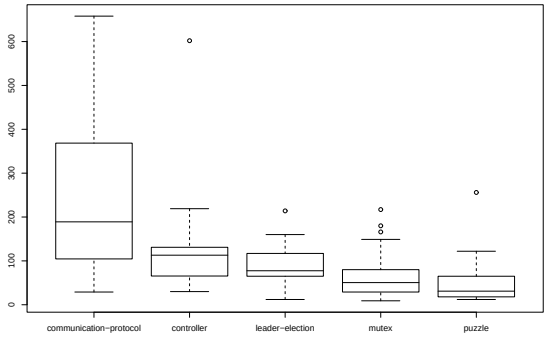



Fig. 6. BFS heights sorted by the type of the model. Results displayed using the boxplot method (see Fig. 1. for description).

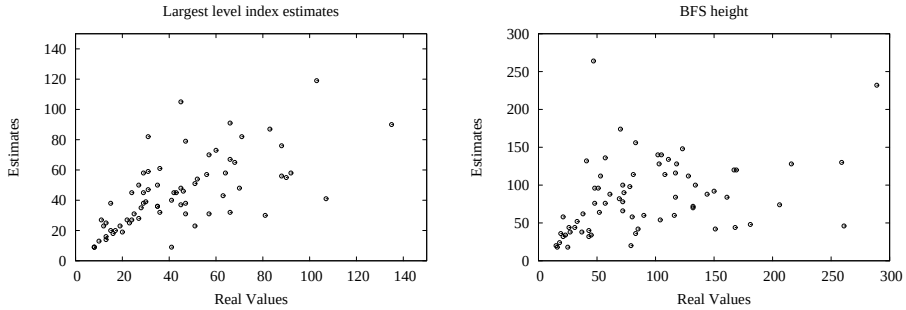



Fig. 7. Estimation of the index of the largest level and BFS height (8 dots are outside the displayed area).

falling. Due to these estimates we are able to recognize local maximums in the graph of BFS levels using Hash-RW. We ignore small local maximums (specified by a constant which is derived from the training set — the local maximum is small if it is up to 20% larger than the last BFS level size estimate) and the search is stopped when a big local maximum is identified. During the reported experiments Hash-RW explored 6.2% of the state space on average.

Fig. 7. shows estimates of the index of the largest BFS level and the BFS height. It is apparent that computing the BFS height as a double of the index of the largest BFS level brings an additional inaccuracy, but estimates are still reasonable. To evaluate the estimates we again compute ratios of estimates and correct values and their standard deviations. For the index of the largest level the standard deviation is 0.72; for the BFS height the standard deviation is 0.84. This means that BFS height estimates usually do not exceed double of the real height and they are rarely lower than one fifth of the real height.

### *4.3 Back Level Edges*

Some algorithms work more efficiently on models with no or only few back level edges [1], [25]. Therefore, it can be useful to know a ratio of back level edges from all edges in the graph.

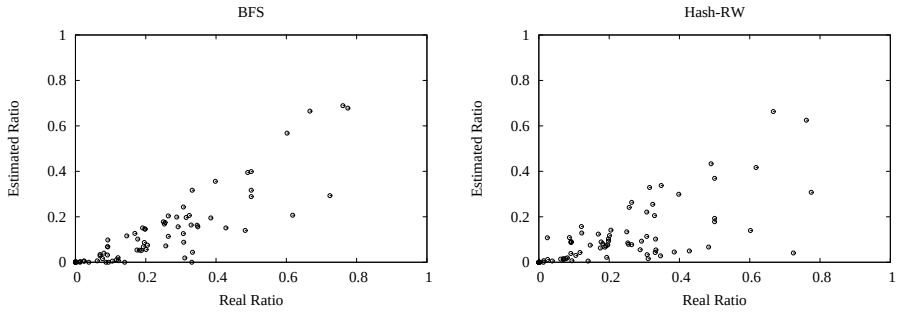



Fig. 8. Estimation of back level edges ratio.

Back level edge is a notion closely connected to BFS. There is a straightforward method to determine how much back level edges is among all transitions — to compute the number of all edges $A$ and back level edges $B$ in a sample given by BFS limited to 5% of the state space. Then we estimate a ratio of back level edges as $B/A$. Since the number of back level edges is naturally growing as more of the state space is explored, our estimates are practically always below the real values (see Fig. 8.) and the standard deviation of the ratio of estimated and real values is 0.42.

Hash-RW can be used as well. The standard deviation of the ratio is worse (0.58), but the technique can find back level edges unreachable by limited BFS, since back level edges are often hidden in high BFS levels. Consequently, Hash-RW is more successful in deciding, whether the given state space has any back level edges or not (90% success rate for BFS, 99% success rate for Hash-RW).

### *4.4 Size of the Largest SCC*

State spaces have a specific structure of strongly connected components [16], particularly with respect to the size of the largest SCC. In [18], we identify three main classes of state spaces: acyclic state spaces, state spaces with small SCCs, and state spaces with one large strongly connected component (more than 50% states). This classification is relevant for example for selection of a distributed cycle detection algorithm [2] or an SCC detection algorithm.

For estimating this classification we use simple random walk exploration [19]. We run 100 independent random walks through the state space. Each random walk starts at the initial state and is limited to at most 2000 steps. During the walk we store visited states, i.e., path through the state space. If a state is revisited then a cycle is detected and its length can be easily computed. At the end of the procedure, we return the length of the longest detected cycle.

Using training data we identified the following bounds for estimation:

- If the longest detected cycle is zero (i.e., no cycle is detected) then we estimate that the state space is acyclic.
- If the longest detected cycle is shorter then 69 then we estimate that the state space contains only small components.
- Otherwise, we estimate that the state space contains one large component.

| | estimated acyclic | estimated small components | estimated large component |
|---|---|---|---|
| acyclic | 28% | 0% | 0% |
| small components | 0% | 16% | 4% |
| large component | 0% | 18% | 34% |

Table 4
Results for SCC structure estimation technique.

Results of this estimation technique over testing data are in Table 4. The method can safely distinguish between acyclic and cyclic state spaces, models with large component are sometimes wrongly classified as models with small components.

## 5 Conclusions and Future Work

In this work we study simple techniques for estimation of state space parameters. Particularly, we focus on techniques based on sampling of the state space. We employ breadth-first search sample, depth-first search sample, random walk, and a novel hash-RW technique. The main messages of our work are:

- Estimation of state space parameters is an interesting problem with applications particularly in the distributed environment.
- Some parameters are easy to estimate (e.g., the average degree), other parameters are rather difficult to estimate (e.g., the number of states, the number of back level edges).
- Selection of a sampling technique matters. Each sampling technique is suitable for estimation of different parameters.
- It seems not reasonable to expect accurate estimates of the number of reachable states. However, when we restrict to three estimate classes and combine several methods, we can get reasonable and useful results. Particularly, it is possible to safely distinguish between huge state spaces and state spaces only slightly larger than a taken sample.

There are several directions for the future work. In this work we restricted our attention to simple techniques. It should be possible to get better estimates by optimizing presented techniques, by parameter tuning, and by incorporating domain specific information.

As an output for a user of a model checker, it would be useful to have on-the-fly estimates of the number of states. Such estimates would be updated regularly during the search (e.g., after the traversal of each BFS level). It would be interesting to have an on-the-fly estimate as an absolute number and to study whether (how fast) the estimate converges to the correct value.

Finally, our long term goal is to use parameter estimates for automation of the verification process [18], i.e., for selection of verification techniques, algorithms, and parameters values.

# References


[1] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 106–115. IEEE Computer Society, 2003.

[2] J. Barnat, L. Brim, and I. Černá. Cluster-Based LTL Model Checking of Large Systems. In *FMCO'05*, volume 4111 of *LNCS*, pages 259–279. Springer, 2006.

[3] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Rockai, and P. Šimeček. DiVinE - A Tool for Distributed Verification. In *CAV'06*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006. The tool is available at `http://anna.fi.muni.cz/divine`.

[4] Jiri Barnat, Lubos Brim, and Pavel Simecek. I/O Efficient Accepting Cycle Detection. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 281–293. Springer, 2007.

[5] L. Breiman. *Classification and Regression Trees*. CRC Press, 1984.

[6] A. Chamillard. An Empirical Comparison of Static Concurrency Analysis Techniques, 1996.

[7] P. C. Dillinger and P. Manolios. Enhanced Probabilistic Verification with 3Spin and 3Murphi. In *Proc. of SPIN Workshop*, volume 3639 of *LNCS*, pages 272–276. Springer, 2005.

[8] J. Geldenhuys. State Caching Reconsidered. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 23–39. Springer, 2004.

[9] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *LNCS*. Springer, 1996.

[10] M. Hammer and M. Weber. "To Store or not to Store" Reloaded: Reclaiming Memory on Demand. In *Formal Methods for Industrial Critical Systems (FMICS'06)*, 2006. To appear.

[11] Irit Katriel and Ulrich Meyer. Elementary Graph Algorithms in External Memory. In *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 62–84, Berlin, Germany, 2003. Springer.

[12] 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.

[13] 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.

[14] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. *SIGARCH Comput. Archit. News*, 25(2):85–97, 1997.

[15] S. Nissen. Implementation of a fast artificial neural network library. Technical report, Department of Computer Science, University of Copenhagen, 2003.

[16] R. Pelánek. Typical Structural Properties of State Spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[17] R. Pelánek. Web Portal for Benchmarking Explicit Model Checkers. Technical Report FIMU-RS-2006-03, Masaryk University Brno, 2006. `http://anna.fi.muni.cz/models`.

[18] R. Pelánek. Model Classifications and Automated Verification. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'07)*, 2007. To appear.

[19] 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.

[20] W. Peng and K. Makki. On Reachability Analysis of Communicating Finite State Machines. In *Proc. of International Conference on Computer Communications and Networks (ICCCN '95)*, page 58, Washington, DC, USA, 1995. IEEE Computer Society.

[21] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[22] D. Sahoo, J. Jain, S. K. Iyer, D. Dill, and E. A. Emerson. Predictive Reachability Using a Sample-Based Approach. In *Proc. of Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 388–392. Springer, 2005.

[23] Ulrich Stern and David L. Dill. Parallelizing the Mur$\phi$ Verifier. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–278. Springer, 1997.

[24] J. F. Watson and A. A. Desrochers. A Bottom-Up Algorithm for State-Space Size Estimation of Petri Nets. In *Proc. of International Conference Robotics and Automation (ICRA'93)*, volume 1, pages 592–597. IEEE Computer Society Press, 1993.

[25] Rong Zhou and Eric A. Hansen. Breadth-First Heuristic Search. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 92–100. AAAI, 2004.

# Chapter 5

# Enhancing Random Walk State Space Exploration

In this paper we study the behavior of random walk techniques in the context of model checking. It turns out that it is rather difficult to understand the behaviour of even the simple random walk. Using the insight gained by our study of simple random walk, we propose several enhancements, e.g., combination with local exhaustive search, caching, or pseudo-parallel walks.

Thorough this work we focus on important but often neglected experimental issues like length of counterexamples, coverage estimation, and setting of parameters. We also test algorithms on inputs of different types – except for state spaces generated by explicit model checkers, we also use random graphs and regular graphs.

This paper was published in proceedings of Formal Methods for Industrial Critical Systems (FMICS) in 2005:

- 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.

The author of the thesis is one of four coauthors of the paper and has done major part of both experiments and writing.

# Enhancing Random Walk State Space Exploration*

Radek Pelánek Tomáš Hanžl Ivana Černá
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 behavior 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 behavior 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.

## Categories and Subject Descriptors

D.2.2 [**Software Engeneering**]: Software/Program Verification—*model checking*

## General Terms

Algorithms, Verification

## Keywords

Random walk, State space exploration, Formal verification

## 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 behavior of random walk on practical examples? Can it be theoretically explained and predicted?
- 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 an 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 behavior 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 has been 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). Unfortunately, most of the results concern undirected graphs whereas the state spaces encountered in model checking are

*Research supported by the Grant Agency of Czech Republic grant No. 201/03/0509 and by the Academy of Sciences of Czech Republic grant No. 1ET408050503


*FMICS'05,* September 5–6, 2005, Lisbon, Portugal.

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 the 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 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 behavior 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 behavior 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 behavior 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 behavior 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 behavior of algorithms. In our experiments we have used manually constructed regular graphs for testing (and usually falsifying) hypothesizes about the behavior 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 behavior 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 given in advance limit on the number of steps is exceeded. Similarly to other randomized algorithms, we always run the random walk several times to obtain expected behavior.

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 practice, 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 summarize 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 summarizes 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 and 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 behavior 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 of these types occur naturally in model checking.

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.

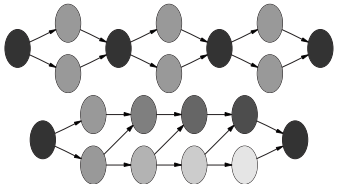


**Figure 1: Graphs with similar properties but different random walk coverage.**

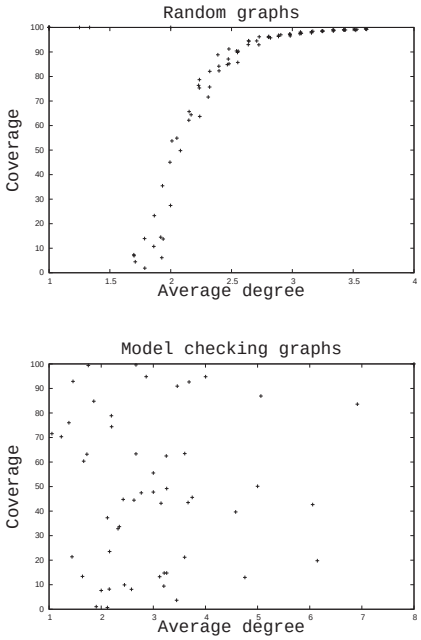



**Figure 2: Correlation between the average degree and coverage for random graphs and model checking graphs.**

### 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.

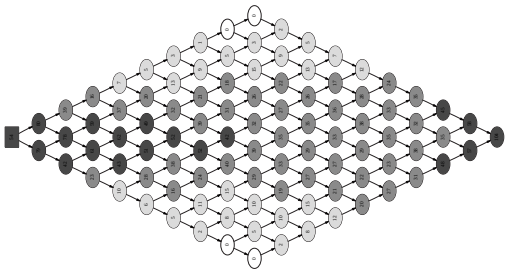


**Figure 3: Behavior of the random walk on a diamond-like structure; darker vertices are visited more often**

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 of visiting the middle states is high. 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 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-initialization*

Re-initialization 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-initialized from) the initial state. The question is how to choose the number of computation steps after which the random walk should be re-initialized. 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-initialization 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-initialized. The set can be for example computed as a frontier of a partial breadth-first search. After re-initialization 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-initialized from the state where the local search has been started, or from a state saved in the respective data structure. The latter 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 randomized 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?

Parallelization 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] use 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-initialization 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 model checking 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 prioritize 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 whose 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 maximize/minimize 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 analyzing components of the model [4, 14] or it can be approximated from the state space of a more abstract model of the system [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

**Table 1: Resulting coverage after number of steps $10\times$ 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 | | | | | |

using enhanced random walk and 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 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 model checking state spaces are typically small [19], there 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 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 $\epsilon$ guarantees that the probability that an error will be found by further random walks is smaller than $\epsilon$. But this does not mean that the probability of existence of an error is smaller than $\epsilon$. 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 metrics can supply a useful information in practice.

## 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 behavior. 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 behavior 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 behavior 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 an 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 2 summarizes the appropriateness of variants relative to the ratio of the available memory to the size of the searched state space.

**Future work** aims at suggesting mechanisms for automatic selection of appropriate methods and/or their parameters for a given application. To this end even broader and

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

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

more extensive case studies can be at hand. Yet another area deserving a deep insight 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).

## 7. 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.

# Chapter 6

# Evaluation of State Caching and State Compression Techniques

In this paper we employ BEEM to thoroughly evaluate two well-studied techniques in explicit model checking: state caching and state compression techniques. The goal of these techniques is to reduce memory consumed by a model checker at the expense of (hopefully slight) increase in running time. Both of these techniques were repeatedly studied and refined in previous research.

We provide review of the literature, discuss trends in relevant research, and perform extensive experiments over models from BEEM. The conclusion of our review and evaluation is that it is more important to combine several simple techniques in an appropriate way rather than to tune a single sophisticated technique.

This paper was published as a technical report [68]. The author of the thesis is one of three coauthors and has done data analysis and most of the writing.

# Evaluation of State Caching and State Compression Techniques*

Radek Pelánek, Václav Rosecký, and Jaroslav Šeděnka

Department of Information Technology, Faculty of Informatics
Masaryk University Brno, Czech Republic

**Abstract.** We study two techniques for reducing memory consumption of explicit model checking — state caching and state compression. In order to evaluate these techniques we review the literature, discuss trends in relevant research, and perform experiments over a large benchmark set (more than 100 models). The conclusion of our evaluation is that it is more important to combine several simple techniques in an appropriate way rather than to tune a single sophisticated technique.

## 1 Introduction

In this work we are concerned with explicit model checking and verification of safety properties. This approach is principally very simple — it is based on the straightforward construction of the whole state space and on a simple reachability analysis of this state space. The technique is successful for verification of asynchronous systems, particularly protocols.

The main problem of explicit model checking is the state space explosion problem and hence large memory requirements of the technique. Researchers proposed during the last 15 years many techniques aimed at reduction of the memory consumption of explicit model checking. At this moment, there is a large number of reduction technique proposals. However, from a practitioner point of view, the situation is not satisfying:

- Research papers usually include only few experiments on selected models on which the techniques bring (non-trivial) improvement. Our evaluation [22] of on-the-fly reduction techniques shows that the improvement is often more humble than claimed in research papers.
- Most reduction techniques involve some kind of trade-off (usually time-memory) and they bring improvement on only some models. The trade-off and the dependence of performance on model properties is not well understood.
- Research papers usually compare a newly proposed technique only to the standard algorithm and not to other reduction technique. It is also not clear whether the impact of different reduction techniques combine.

* Partially supported by GA ČR grant no. 201/07/P035.

### 1.1 Contribution

In this paper we focus on two techniques for reducing the memory consumption of the explicit model checking: state caching and state compression. State caching saves memory by deleting some visited states from memory; state compression saves memory by compressing individual states in memory. We evaluate these techniques in the following way:

- We review the literature and discuss trends in the relevant research.
- We perform experiments with caching and compression techniques over a large benchmark set (BEEM [23]) and we present results of these experiments and their interpretation.
- We analyze how the performance of the techniques depends on model and state space parameters.

### 1.2 Context

This work is a part of our long term effort to make the verification process more automatic, i.e., to automate the selection of reduction techniques and parameters [24]. Our other works which contribute to this goal are the following:

- a general overview and evaluation of on-the-fly reduction techniques [22],
- evaluation of techniques for error detection [25],
- analysis of properties of state spaces [21],
- description and evaluation of techniques for estimating state space size and other parameters [26].

## 2 Review of Literature

Before the discussion of related work we clarify the terminology that we use to discuss the effect of techniques for reduction of memory consumption. We use the notion 'reduction ratio' to denote the ratio between the memory consumption of the reachability search with a memory reduction technique and the memory consumption of the standard reachability. Some authors report 'reduced by' factor, i.e., if we report 'reduction ratio' 80%, it means that the memory consumption was 'reduced by' 20%.

State caching and compression techniques have been studied rather extensively. The main works are the following:

- Holzmann [10,11] was the first to propose the state caching technique, but he did not perform realistic evaluation of the technique.
- Godefroid et al. [8] focus on the relation of state caching and partial order reduction. Their experiments are, however, limited to only few models.
- Geldenhuys [5] performs more extensive evaluation and compares many different caching strategies. However, this evaluation is done partly on random graphs, which can be misleading (for argumentation against the use of random graph see [21]).

```
proc Explore(M)
  add s_0 to Wait
  while Wait ≠ ∅ do
        remove s from Wait
        explore (s)
        foreach s → s' do
          if s' ∉ Visited then add s' to Wait fi od
  od
end
```

**Fig. 1.** The basic algorithm for exploring the state space.

- Similar approaches to state caching are selective storing of states during the search [2] and the sweep line technique [3], which deletes only states that are guaranteed to not be revisited.
- Holzmann [12] describes the state compression algorithm with training runs in Spin model checker. Each part of the state space is compressed according to a local table.
- Visser [28] describes a similar compression techniques as Holzmann [12] and combines the compression technique with OBDD storage.

Table 1 gives the overview of these and several other most relevant papers. From this table we can see the following general trends:

- There is a steady flow of publications about the topic (1-2 every year).
- At first, techniques were implemented in SPIN (and its predecessors), from 2000 onwards the scope of used tools is rather diverse.
- We expected that the (reported) reduction ratio would increase in time, however there is no clear trend with time. The reported reduction ratio is usually in the interval 5% to 80%.
- The quality of experiments (number and quality of used model and performed experiments) is nearly constant, despite the improving availability of realistic models.

## 3 Techniques

In this section we describe formally the context of our work and the state caching and state compression techniques.

### 3.1 State Space Exploration

Figure 1 gives the basic state space exploration algorithms. It is just a simple graph traversal algorithm, which uses two important data structures:

**Table 1.** Review of literature. Experiments on random graphs are not taken into account. The 'reported reduction ratio' is only an approximate due to the use of different metrics to measure memory consumption; the reduction ratio is given only if enough experiments are reported in the paper. (*) The small reduction ratio is due to combination with partial order reduction method.

| paper | year | technique | tool | number of models | reported reduction ratio |
|---|---|---|---|---|---|
| [10] | 1985 | caching | Trace | 1 | |
| [11] | 1987 | caching | Argos | 2 | |
| [16] | 1991 | caching | unknown | 0 | |
| [13] | 1992 | compression | SPIN | 5 | $\sim$ 80% |
| [8] | 1992 | caching | SPIN | 4 | 1-3% (*) |
| [28] | 1996 | compression | SPIN | 5 | 5-25% |
| [9] | 1996 | compression | SPIN | 3 | $\sim$ 80% |
| [19] | 1997 | caching-like | ARC | 11 | 15-50% |
| [12] | 1997 | compression | SPIN | 17 | 15-50% |
| [17] | 1997 | compression, caching-like | Uppaal | 10 | 5-25% |
| [20] | 1998 | compression | SPIN | ? | 20-60% |
| [14] | 1999 | compression | SPIN | 14 | $\sim$ 15% |
| [6] | 1999 | compression | SPIN | 7 | 40-60% |
| [3] | 2001 | caching-like | Design/CPN | 3 | 5-80% |
| [27] | 2001 | caching-like | Murphi | 20 | $\sim$ 60% |
| [18] | 2001 | compression | JPF | 2 | $\sim$ 5% |
| [2] | 2003 | caching-like | Uppaal | 9 | 5-80% |
| [7] | 2003 | compression | TVT | 4 | 15-50% |
| [5] | 2004 | caching | SPIN | 18 | 5-50% |
| [4] | 2005 | compression | Helena | 8 | 30-60% |

– *Visited* is a set of states that were visited during the exploration. Since we need to perform a test of membership in this data structure, the set is usually represented by hash table.
– *Wait* is a set of states that need to be explored. The implementation of this data structure determines the search order of the algorithm — usually either breadth-first search (BFS) or depth-first search (DFS).

States are represented as vectors of bytes, which code the current location of individual processes and values of variables.

### 3.2 State Caching

The basic idea of state caching is simple: if we run out of memory then we remove some states from the data structure *Visited*. This, of course, has the consequence of revisits of states and thus time increase. If we use depth-first search, the method is still guaranteed to terminate. With breadth-first search order we do not have such a guarantee in general.

Note that the name of the approach is slightly misleading, since it is not caching in the usual sense of the word — states are *not* moved lower in the memory hierarchy, they are simply deleted. However, in model checking the technique is traditionally called this way [8,5].

**Caching Strategies** The main issue in application of state caching is to determine states for removal from the cache. We call an algorithm for selection of states a *caching strategy*. In our experiments we consider the following strategies (see also [5]):

**O** (out-degree) States are removed according to the number of successors (out-degree) of the state. The intuition behind the strategy is that states with higher number of successors have higher probability that some of its successor was forgotten.

**I** (in-degree) States are removed according to the number of visits (actual in-degree) of the state. The intuition behind the strategy is that often visited states will also be more visited in the future.

**OI** (out-degree, in-degree) The strategy takes into account both the out-degree and the actual in-degree.

**RAND** (random) States are removed randomly.

**SC** (stratified caching) We assign to every stored state the depth on which it was discovered (the length of the path from the initial state). States lying on the same depth form stratas. We erase state $s$ when the following predicate is true ($d$ is depth of a state $s$, $k$ is a constant, initially $k = 2$):

$$d \bmod k \neq k - 1$$

When no such state exists, the constant $k$ is doubled, so there are additional available stratas with states for erasing.

**Table 2.** Hufmann code used for the compression — part of the static code.

| | |
|---|---|
| 0 00 | ... ... |
| 1 01 | 248 111111111010 |
| 2 1000 | 249 111111111011 |
| 3 1001 | 250 1111111111000 |
| 4 1010 | 251 1111111111001 |
| 5 1011 | 252 111111111101 |
| 6 11001 | 253 111111111110 |
| 7 11000 | 254 111111111111 |
| 8 1101 | 255 11100 |

### 3.3 State Compression

State vectors contain significant redundancy. Researchers proposed several techniques that try to reduce the size of state vectors (see Section 2). Previous studies suggest that these techniques achieve similar results, therefor we focus on the most typical compression techniques — huffman coding.

Huffman coding [15] is a general loss-less compression method that is proven to be memory-optimal when exact probabilities of each value usage are known. The Huffman code can be constructed by the well-known algorithm [15].

**Static Code** In order to construct the Huffman code we need to know the frequencies of individual values. Since prior to the reachability analysis we do not have the knowledge of the frequencies, we cannot construct the optimal Huffman code. Nevertheless, we can construct at least some Huffman code using approximated frequencies.

A straightforward approach to compute this estimated frequencies is to take a representative set of small models, compute frequencies of values in these models and then compute a "static" Hufmann code. Part of this static Huffman code (that we use in the experiments) is shown in Table 2.

**Training Runs** A more precise way of gathering value frequencies is to go through a small part of the state space, i.e., to perform a short reachability. We call this preliminary reachability a *training run*, as it is not supposed to walk through the whole state space. After such training run we create Huffman trees based on inaccurate probabilities.

There are several possibilities how to perform the training run. We consider two basic approaches to sampling a state space during the training run: DFS and BFS, each with a specified limit on a number of states to be searched.

## 4 Evaluation

Reported techniques are implemented in the DiVinE environment [1]. Experiments were performed on 2GHz Intel Xeon Linux workstation with 16 GB RAM.

Evaluation was performed over the BEEM benchmark set [23]. The web portal of the benchmark set contains all the used models and it also presents detailed information about models. For our experiments we used 120 models, which fall into following (BEEM) categories: 29 communication protocols, 18 controllers, 6 leader election algorithms, 26 mutual exclusion algorithms, 22 scheduling problems, 19 other; 26 complex case studies, 70 simple models, 24 toy examples.

In this section we report only summaries of results and their interpretation; all results can be found on the following web page:

`http://anna.fi.muni.cz/~xrosecky/mem_reduction/`

We measured (computed) both the real memory consumption and the (theoretical) consumption of the storage data structures. The real memory consumption is higher due to the additional implementation overheads. The distinction between these two metrics is important because the used techniques can reduce only the memory consumed by storage data structures. Each of these metrics have its (dis)advantages — theoretical consumption of storage structures is less implementation dependent, however real memory consumption is, at the end, what really counts.

Nevertheless, our experiments show that these two metrics are rather well correlated and that the main conclusions of our experiments are not dependent on the used metric. In the following we use the real memory consumption. More specifically, we report relative memory consumption with respect to standard reachability.

### 4.1 Caching

For state caching there is a clear trade-off between memory consumption and runtime. Figure 2 gives several examples of this trade-off. Since our focus is on memory consumption, we use the following approach. We use a relative time limit — 6 times the runtime of standard reachability. We run reachability with caching with different cache sizes, cache sizes are also set relatively to the full reachability ($i \cdot 10\%$ of state space size). We consider the run successful if it finishes within the given time limit.

**Comparison of strategies** We run the experiment for all caching strategies. Since the performance of caching is dependent on the search order (BFS or DFS), we also combined each strategy with both BFS and DFS order. We use the following notation: BFS-SC means reachability search with BFS order and SC caching strategy. Here we report only on 7 of these combined strategies, the results are representative of the overall results:

- The overall results of different strategies are similar, nevertheless there is a difference between different strategies (Fig. 3).
- There is no universally best strategy, strategies are to a certain degree complementary; Figure 4 shows which techniques have similar behaviour.

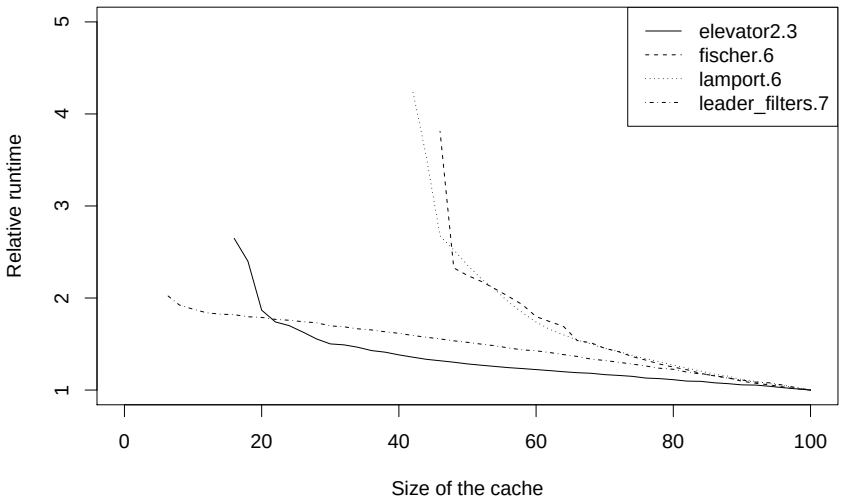



**Fig. 2.** Trade-off between size of the cache (reduction of memory consumption) and runtime increase.

- The most successful strategies are the following (numbers indicate how many times a technique achieves the best result): BFS-SC (56), DFS-SC (37), BFS-OI (14), DFS-OI (7).
- In 71% of cases the best strategy is successful with cache size 20% of less of the full state space.

**Dependence on a model** The performance of caching depends on the model — for some models we can use small cache, for other models we need to store nearly all states. Which characteristics of the model influence the performance of caching?

- Type of a model: for controllers, leader election algorithms, and communication protocols caching works well (10% cache is sufficient for more than a half of these models), whereas for mutual exclusion algorithms caching works poorly (at least 40% cache is necessary for more than a half of these models).
- Average degree of the state space (correlation coefficient is 0.53): caching works better for sparse state spaces (see Figure 5).
- BFS height of the state space (correlation coefficient is -0.43 with respect to logarithm of the height): caching works better for state spaces with many BFS levels (see Figure 5).

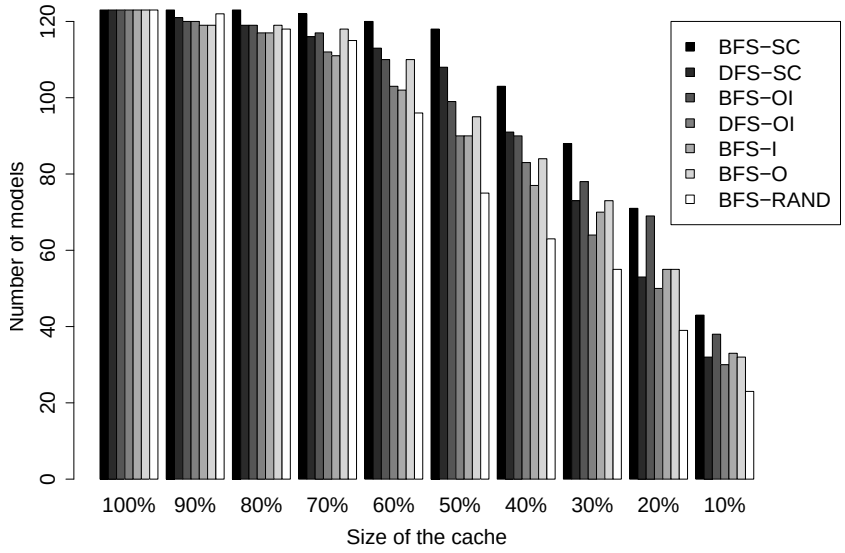



**Fig. 3.** Comparison of caching strategies. The graph shows the number of successes for different strategies and cache sizes.

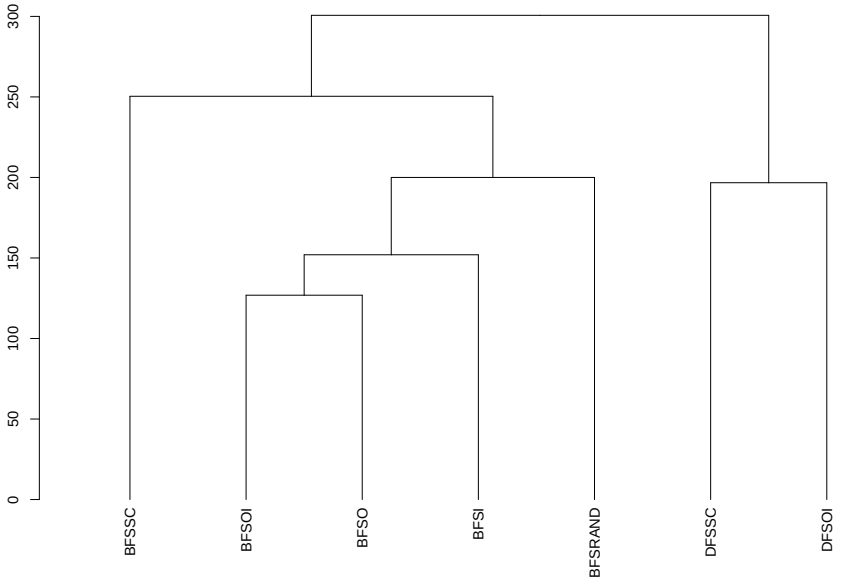



**Fig. 4.** Clustering tree which shows similarity between caching techniques.

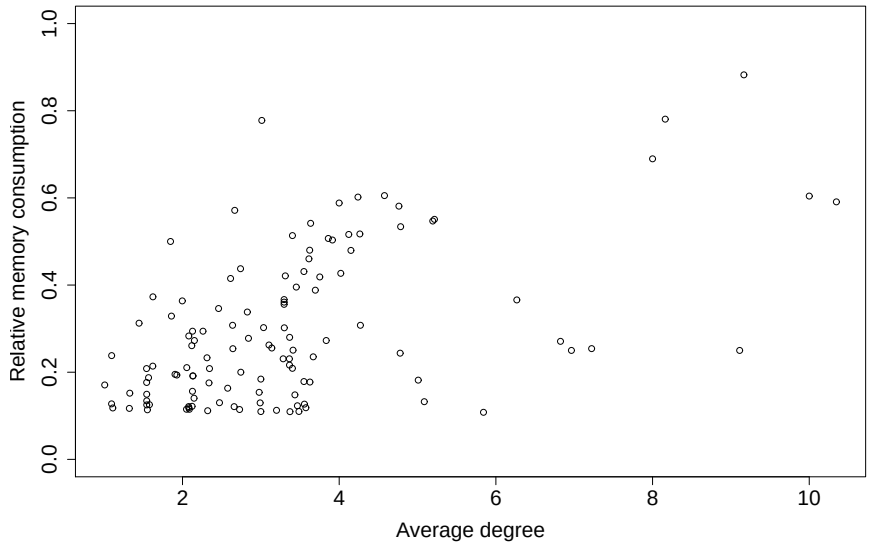



**Fig. 5.** Correlation between best results of caching techniques and average degree of a state space.

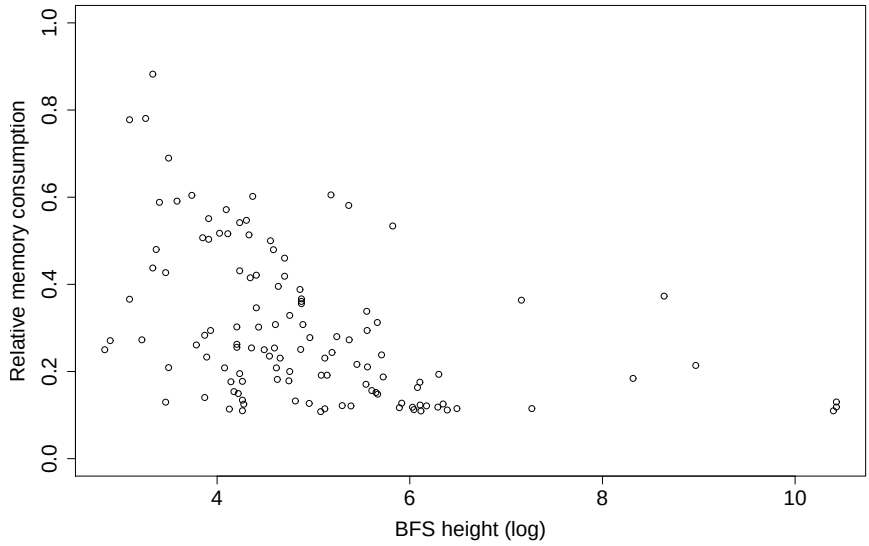



**Fig. 6.** Correlation between best results of caching techniques and BFS height of a state space.

### 4.2 Compression

The performance of state compression technique is not dependent on the search order (we use BFS). For each model we run the compression technique with static code and with codes obtained by analysis of 4 different training runs: BFS (respectively DFS) with limit of 2% (respectively 15 %) of the state space. Results of these experiments can be summarized as follows:

- The reduction ratio is in range 40% to 90%, typically 60% (see Figure 7).
- The reduction ratio is slightly better with the huffman code obtained from training runs than with the static code (approximately 10% better) (see Figure 7).
- Neither the type of the training run (BFS, DFS) nor the size of the training run (2% or 15% of the state space) are important — the reduction ratio is very similar (see Figure 7).
- The performance of state compression technique depends on state size — there is very good linear correlation with logarithm of state size (see Figure 8, correlation coefficient is -0.84).
- There is nearly no relation between the runtime increase and reduction of memory consumption (cf. caching techniques).
- The performance of state compression technique is not related to the type of model (cf. caching techniques).

### 4.3 Combination

Finally, we implemented and evaluated the combination of caching and compression techniques. For the evaluation we used fixed strategies: BFS search order with stratified caching and compression with dictionary computed by a BFS training run (15% of the state space). The results are following (see also Figure 9):

- Since the techniques are rather orthogonal, they combine well.
- The achieved reduction ratio is between 5% and 70% with median value 25%.
- Better results are achieved for complex case studies, particularly for leader election and communication protocols, worse results are for mutual exclusion algorithms.

## 5 Conclusions

In this paper we focus on two techniques for reducing memory consumption of model checking algorithms: state caching and state compression.

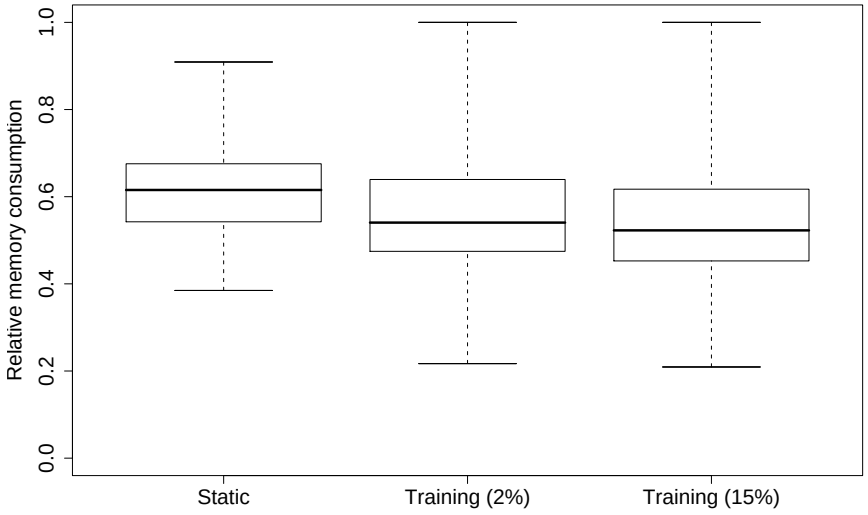



**Fig. 7.** Compression: static dictionary, training runs

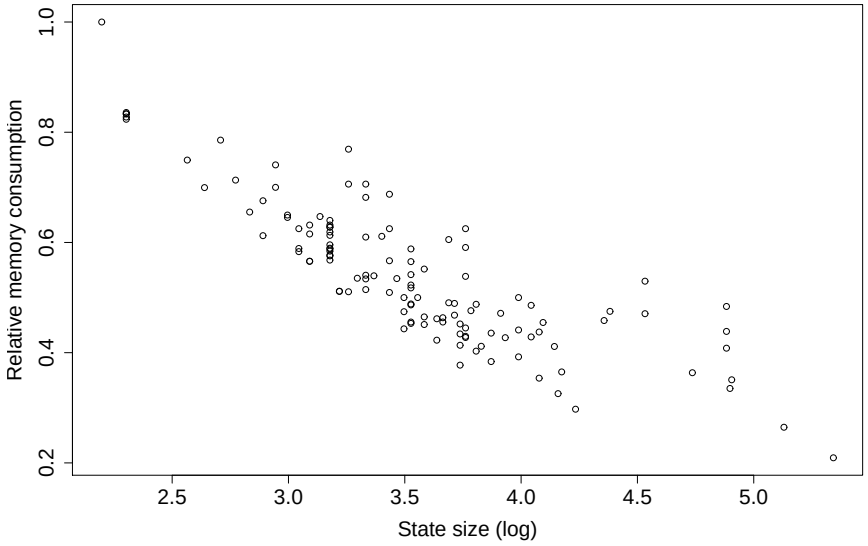



**Fig. 8.** Correlation between state size and memory consumption

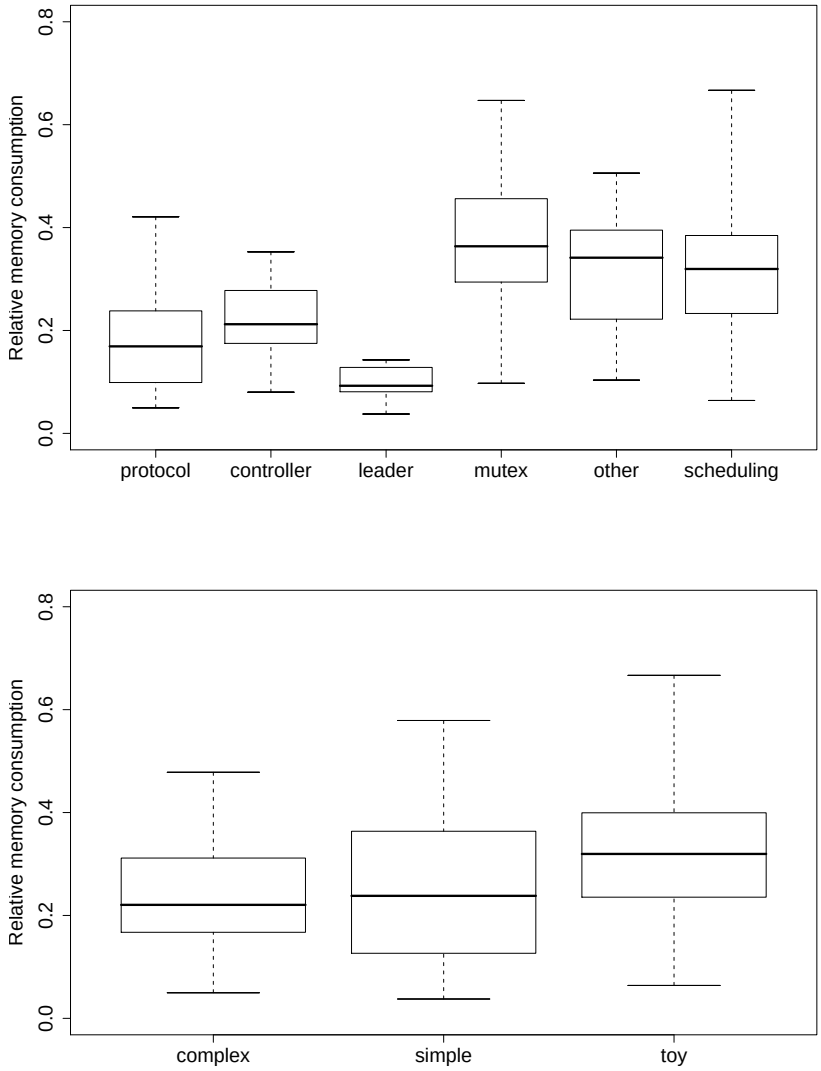



**Fig. 9.** Results for combination of caching and compression

### 5.1 Evaluation of Effectiveness

We evaluate these techniques over a large set of models and reach the following conclusions about their effectiveness:

- Caching strategies are to a certain degree complementary. Using an appropriate state caching strategy, the memory consumption can be in most cases reduced to 10% to 30%. Using a fixed strategy (stratified caching with BFS order), cache of the size 30% of the state space is sufficient in 3/4 of cases.
- Using state compression the memory consumption can be usually reduced to approximately 60%. Using training runs (a short preliminary reachability analysis) the performance of compression can be improved, but only slightly.
- The two techniques combine well.
- Effectiveness of state caching is related to average degree, height of the BFS tree and the type of a model. Effectiveness of state caching differs for toy models and real case studies.
- Effectiveness of state compression is very well correlated with state size, it does not depend on other parameters.

### 5.2 Comparison with Related Work

Let us also put our work in the context of the numerous related work. Rather then tuning a single implementation (as is the case of most of the related work), we try several simple strategies and parameter values. We also use significantly larger number of models than other studies and compare the performance on different types of models. In this context, the results of our study are the following:

- Our review of the literature as well as our experimental results show that the reduction ratio obtained by state caching and state compression techniques is in most cases in the interval 10-80%, i.e., with the use of these techniques it may be possible to traverse up to 10 times larger state space than by standard search.
- Using simple and easy-to-implement techniques, we are able to achieve very similar results as reported in other works which use more sophisticated approaches.
- The performance depends on used models — the choice of models does not change the overall results fundamentally (smaller number of models may be sufficient to get the basic insight), but with respect to comparison of techniques the choice of models can be important. Caching and compression techniques work better on realistic models than on academic toy examples.

### 5.3 Outlook

In the case of state caching and state compression techniques it seems better to implement several simple technique than a single sophisticated one. Rather than tuning a single technique, we should focus on methods for selecting an appropriate simple techniques and choosing suitable parameter values.

## References


1. 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`.
2. G. Behrmann, K. G. Larsen, and R. Pelánek. To store or not to store. In *Proc. of Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*. Springer, 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 2001)*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.
4. S. Evangelista and J.-F. Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *Proc. of Model Checking Software (SPIN)*, volume 3639 of *LNCS*, pages 43–57. Springer, 2005.
5. J. Geldenhuys. State caching reconsidered. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 23–39. Springer, 2004.
6. J. Geldenhuys and P. J. A. de Villiers. Runtime efficient state compaction in SPIN. In *Proc. of SPIN Workshop*, volume 1680 of *LNCS*, pages 12–21. Springer, 1999.
7. J. Geldenhuys and A. Valmari. A nearly memory-optimal data structure for sets and mappings. In *Proc. of Model Checking Software (SPIN)*, volume 2648 of *LNCS*, pages 136–150. Springer, 2003.
8. P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. of Computer Aided Verification (CAV 1992)*, volume 663 of *LNCS*, pages 178–191. Springer, 1992.
9. J. Gregoire. State space compression in spin with GETSs. In *Proc. Second SPIN Workshop*. Rutgers University, New Brunswick, New Jersey, 1996.
10. G. J. Holzmann. Tracing protocols. *Bell System Technical Journal*, 64(2413-2434):336, 1985.
11. G. J. Holzmann. Automated protocol validation in argos: Assertion proving and scatter searching. *IEEE Trans. Softw. Eng.*, 13(6):683–696, 1987.
12. G. J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proc. of SPIN Workshop*, 1997.
13. G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. of Protocol Specification, Testing, and Verification*, 1992.
14. G. J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer (STTT)*, 3(1):270–278, 1998.
15. D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. of the Institute of Radio Engineers*, 40(9):1098–1101, Sep 1952.
16. C. Jard and T. Jéron. Bounded-memory algorithms for verification on-the-fly. In *Proc. Computer Aided Verification (CAV'(91)*, volume 575 of *LNCS*, pages 192–202. Springer, 1992.
17. K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Proc. of IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society, 1997.
18. 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.
19. A. N. Parashkevov and J. Yantchev. Space efficient reachability analysis through use of pseudo-root states. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *LNCS*, pages 50–64. Springer, 1997.

20. B. Parreaux. Difference compression in spin. In *Proc. of Workshop on automata theoric verification with the SPIN model checker (SPIN'98)*, 1998.
21. R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.
22. R. Pelánek. Evaluation of on-the-fly state space reductions. In *Proc. of Mathematical and Engineering Methods in Computer Science (MEMICS'05)*, pages 121–127, 2005.
23. R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, LNCS. Springer, 2007. To appear.
24. R. Pelánek. Model classifications and automated verification. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'07)*, 2007. To appear.
25. R. Pelánek, V. Rosecký, and P. Moravec. Complementarity of error detection techniques. In *Proc. of Parallel and Distributed Methods in verifiCation (PDMC)*, 2008. To appear.
26. R. Pelánek and P. Šimeček. Estimating state space parameters. Technical Report FIMU-RS-2008-01, Masaryk University Brno, 2008.
27. G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Exploiting transition locality in automatic verification of finite state concurrent systems. *Software Tools for Technology Transfer (STTT)*, 6(4):320–341, 2004.
28. W. Visser. Memory efficient state storage in SPIN. In *Proc. of SPIN Workshop*, pages 21–35, 1996.

# Chapter 7

# Complementarity of Error Detection Techniques

In this paper we study the performance of techniques for error detection and we focus particularly on the issue of complementarity. Using experimental evidence we argue that it is not important to find the best technique, but to find a set of complementary techniques (as discussed in Section 1.3.2). We choose nine diverse error detection techniques (e.g., depth-first search, directed search, random walk, and bitstate hashing) and perform experiments over the BEEM set.

The topic is closely connected to the research in testing. Therefore, in our evaluation we compare not just a speed of techniques, but also model coverage metrics that are used in the testing domain. The result of our experiments show that the studied techniques are indeed complementarity in several ways.

This paper was published in proceedings of International Workshop on Parallel and Distributed Methods in verifiCation (PDMC) in 2008:

- R. Pelánek, V. Rosecký, and P. Moravec. Complementarity of error detection techniques. In *Proc. of Parallel and Distributed Methods in verifiCation (PDMC)*, volume of 220 ENTCS, 2008.

The author of the thesis is one of three coauthors of the paper and has done the analysis of data and most of the writing.

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 220 (2008) 51–65
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

# Complementarity of Error Detection Techniques

Radek Pelánek,[1] Václav Rosecký[1] and Pavel Moravec[2]

*Faculty of Informatics*
*Masaryk University Brno, Czech Republic*

**Abstract**

We study explicit techniques for detection of safety errors, e.g., depth-first search, directed search, random walk, and bitstate hashing. We argue that it is not important to find *the best* technique, but to find a set of *complementary* techniques. To this end, we choose nine diverse error detection techniques and perform experiments over a large set of models. We compare speed of techniques, lengths of reported counterexamples, and also achieved model coverage. The results show that the studied set of techniques is indeed complementary in several ways.

*Keywords:* explicit model checking, experimental evaluation, parallel execution

## 1 Introduction

There are many methods for checking correctness of computer systems, e.g., testing, model checking, static analysis, theorem proving. Currently, these techniques are being successfully combined and the border between them is more and more blurred. Although the research community focuses mainly on verification, industry is concerned with falsification: "Falsification comes before verification! Maximise the number of found bugs per hour per spend euro." [13]. In this work we study a spectrum of falsification techniques between model checking and testing.

Nowadays there is a significant trend which should change the way we study and evaluate verification and falsification techniques — a trend towards cheap and widely available parallelism. Consequently, it is possible to run several techniques in parallel, either on a single multi-core machine or on a network of workstations. This has two important consequences:

[1] Partially supported by GA ČR grant no. 201/07/P035.
[2] Partially supported by The Academy of Sciences of the ČR grant no. 1ET408050503.


doi:10.1016/j.entcs.2008.11.013

(i) Rather than focusing on "universal" techniques (suitable for both verification and falsification), it is better to develop specialised techniques and run them in parallel (either on two different machines or as independent threads on a multi-core machine).

(ii) Rather than focusing on the search of "the best" technique, it is better to look for a set of complementary techniques, such that each of these techniques works well on different kind of models. Such a set of techniques can be again run in parallel. Note that it is not necessary to know which technique works well for which models.

Our goal in this work is to find a set of complementary techniques for error detection of safety properties (plain reachability analysis). To this end, we study techniques which lie on the spectrum between explicit model checking (systematic traversal of a state space) and testing (exploration of sample paths through a state space), e.g., breadth-first search, randomized depth-first search, bitstate hashing, directed search, random walk, and under-approximations based on partial order reduction. Our specific contributions are the following:

- We give an overview of explicit error detection techniques. We show that there are several basic building blocks which are nearly orthogonal and which can be combined in many ways. Previous studies usually focused on a specific technique.
- We choose nine diverse techniques, implement them in a single setting, and experimentally evaluate them over a large benchmark set. This is the first study that compares a large number of different techniques. Previous studies compared only two techniques or several variants of the same technique.
- We study the impact of model selection on results of experiments. Such analysis has not been done in previous studies in this domain.
- We study the ability to detect specified errors, the length of counterexamples and also the model coverage (as measured by coverage metrics). We focus on complementarity with respect to these different aims. Previous studies focused only on one of the described aspects.

*Related work*

One line of related work deals with study of a single error detection technique, e.g., bitstate hashing [11], directed search (also called guided search) [14,7], state-less search [9], or random walk [22]. These works only compare a proposed technique to a standard search technique (BFS, DFS).

Interesting line of recent research has focused on randomized techniques [6,5,24,23]. These papers show an interesting point: sometimes the effect of randomization can overshadow the effect of sophisticated optimization techniques. These works, however, usually compare only two techniques (e.g., random walk versus random DFS [24], directed search versus randomized directed search [23]).

Another line of research is concerned with coverage metrics and test case generation, particularly with test input generation for Java containers [2,15,17,18]. These

works are, however, often specific to a particular application domain (containers) and consider only coverage metrics, not an error detection.

In this work we advocate the use of parallel computation. There is already a large amount of research work devoted to application of parallel and distributed computation in verification. Most of this work, however, focuses on parallelization of one procedure. Such approach incurs significant communication overhead. We advocate a different application of parallelism — several different procedures which run completely independently.

Most of the related work share several deficiencies of the experimental work: poor experimental data (only simple models or a small number of models is used), lack of transparency and reproducibility (used models are not available, verified properties are not stated, implementation details are veiled), comparisons are unfair (compared techniques are programmed using different sets of programming primitives). In our work we try to overcome these deficiencies.

*Organization of the paper*

Section 2 presents general building blocks of error detection techniques and describes the specific techniques that we compare. Section 3 describes the experimental methodology that we use (implementation details, used models, performance measures). Section 4 presents main results of our experiments. Main points are summarised in Section 5 and future directions are outlined in Section 6.

## 2 Overview of Techniques

Error detection techniques are based on several basic building blocks. Although these building blocks are not completely independent, there is a large degree of orthogonality and thus these building blocks can be combined in many ways. In this section we give an overview of building blocks and specify which specific techniques we use for the experimental evaluation. We also describe an artificial example which illustrates complementarity of techniques.

### *2.1 Building Blocks of Error Detection Techniques*

Fig. 1. gives a simplified general pseudocode of an error detection technique. A data structure *Wait* holds states that are to be visited by the search. A data structure *Visited* holds information about states that have already been visited. A technique inserts the initial state to the *Wait* structure, then it repeatedly extracts *some* state from the *Wait* structure, checks whether the state violates the property, takes *some* successors in *some* order and if these successors are not *matched* within the *Visited* structure then it adds these states to the *Wait* structure and *updates* the *Visited* structure. However, there are many ways how to implement these general operations (marked by italics in this paragraph).

#### 2.1.1 Selection of states

We need to specify the way how successors of the current state are selected (line 6):

- complete search: all successors are selected,
- incomplete search: only some successors are selected, e.g.,
  - random selection of one state,
  - selection of several states according to a heuristic function.

#### 2.1.2 Search order

We need to specify in what order states are extracted from the *Wait* structure (line 4):

- breadth-first search order (*Wait* implemented as a queue),
- depth-first search order (*Wait* implemented as a stack [3]),
- order given by a heuristic function (e.g., best first search).

#### 2.1.3 State storage and matching

We need to specify what information to store in the *Visited* structure and how to use this information (lines 7, 9). The most common approaches are the following:

- *Visited* is implemented as a standard set of states (usually implemented as hash table which holds states in a collision list).
- *Visited* is a hash table which stores just one bit for each row in a table (i.e., no collision detection), this technique is usually called bitstate hashing [11]. A more general technique is based on bloom filters [4].
- *Visited* stores and performs matching on abstract states computed by a given abstraction function [19].
- *Visited* data structure is not used at all (random walk or state-less search [9]).

[3] We note that in order to get the exact depth-first search order, it is not sufficient to use the pseudocode in Fig. 1. with *Wait* implemented as stack; it is necessary to slightly modify the code.

```
1  proc ErrorDetection(M, φ)
2    insert initial state to Wait
3    while not finished do
4          get s from Wait
5          if s violates φ then  return path to s fi
6          foreach s' ∈ selected successors of s do
7            if s' not matched in Visited
8              then insert s' to Wait
9                   update Visited with information about s' fi
10         od od
11 end
```

Fig. 1. Basic pseudocode for error detection techniques.

#### 2.1.4 Repetition

For some techniques it is meaningful to do a repetition or refinement of the search. Such techniques may be terminated (e.g., after reaching a time limit or filling a hash table) and then be called again.

- No repetition or refinement. If a technique is deterministic and does not have any parameters, then there is no point in repeating the search.
- Repetition with different seed. This is meaningful for techniques that use randomization (e.g., random walk [22] or randomized DFS [3,24]).
- Repetition with changes of parameters (refinement of the search). This is meaningful for techniques that have parameters which influence the above stated building blocks (e.g., size of hashing table for bitstate hashing [11], predicate abstraction function for matching based on abstraction [19]).

### 2.2 Techniques Used for Evaluation

The described building blocks can be combined in many ways. For our experimental evaluation we select the following techniques and parameter values (Section 3.4 describes the methodology used for the selection of techniques and parameters):

**BFS** Breadth-first search.

**DFS** Depth-first search. Successors of a state are taken in a fixed order given by the state generator.

**RDFS** Randomized depth-first search. Successors of a state are taken in a random order [3,24].

**RW** Random walk. Classical random walk with a fixed length (500 states) and repetition.

**ERW** Enhanced random walk. Combination of random walk with local exhaustive BFS [22]. The technique is parametrized by the probability that the local BFS is started (0.004), the number of states explored by the BFS (5000 states), and the number of random walk steps before reinitialization (500 states).

**BITH** Bitstate hashing with repetition [11,12]. The search uses DFS, in the first iteration the size of a hash table is small (8000 bits) and in each repetition we enlarge the size of hash table (multiply by 4).

**DIRS** Directed search with structural heuristic. The overall score for state is defined as a sum of ranks for transitions, which lead to that state. Rank for a transition is defined as a sum of count of read variables in its guard and count of modified variables in its effect, plus one in a case of a communication. The heuristic is inspired by [10].

**DIRG** Search directed by heuristic function given by the goal. We try to estimate remaining length to reach some goal state. The heuristic function $h_g(s)$ for state $s$ and goal $g$ is obtained by direct transformation of the goal (we use the same transformation as [7,14]).

**UPOR** Under-approximation refinement based on partial order reduction [16]. The

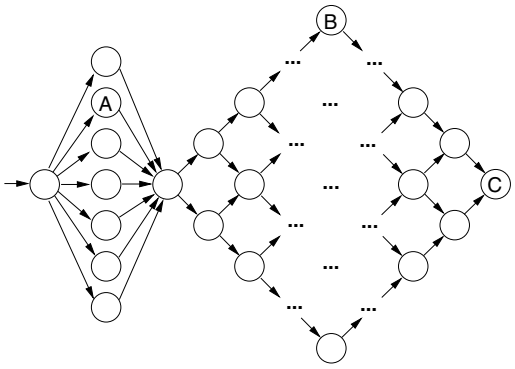



Fig. 2. An artificial example, the right part of the graph is comprised of a 'big diamond'.

technique is based on an under-approximation of conditions of a correct partial order reduction. Gradual refinement of such approximations generates a sequence of subspaces of the original model such that the subspaces have increasing set of behaviours. Approximations are based on BFS.

### *2.3 Illustration on an Artificial Example*

Fig. 2. shows an example, which demonstrates typical features of state spaces of realistic models although it is artificially constructed. Let us discuss the behaviour of the three most classical techniques on this example:

- BFS quickly finds state A, whereas state C is found as the last one.
- DFS can quickly find state C, the detection of states A, B depends on the search order, but with high probability the state A will be one of the last ones to be found.
- RW does quickly find state C, state A will be also find reasonably quickly, but the state B is difficult to reach by RW.

This example illustrates our main point: different techniques are complementary — technique A may work well in case X but not in case Y while technique B may work well in case Y but not in case X. The example also illustrates another often neglected fact: the performance of error detection techniques depends not only on models, but also on errors (goals) of interest.

## 3 Experimental Methodology

We try hard to make our experiments fair, transparent and reproducible. All the experimental data are available at:

`http://www.fi.muni.cz/~xrosecky/error_detection`

The webpage contains implementation source codes, list of all models and their source codes, verified properties, and all results. For the implementation we use the Distributed Verification Environment (DiVinE) [1], which is also publicly available. We have tried to make the comparison fair by paying special attention to implement all techniques in similar and comparable way.

### 3.1 Models and Errors

Models are specified as finite state machines extended with integer variables and communication. We use models from the BEEM set (BEnchmarks for Explicit Model checkers) [20]; some models have been slightly modified, particularly we have seeded additional errors to models. The used models span several application domains, particularly mutual exclusion algorithms, communication protocols, and controllers. We use 54 models; all the used models have large state spaces — in most cases the whole reachable state space does not fit into memory (explicitly represented) and even the fastest error detection technique needs to visit thousands of states before the error is detected.

Except for models, our techniques take as an input a goal for error detection (a boolean expression). We search not only for real errors in erroneous versions of models, but also for interesting states in correct versions of models. This is also useful — for example the user may be interested how the protocol can get to a certain configuration. However, in order to make the explanation simpler, in this paper we always use the term "error detection".

### 3.2 Performance Measures

As a main measure of technique's performance we use the number of states processed by the technique before a goal state is found. Other studies usually use time as a main performance measure. However, time depends on a specific machine and implementation and is not reproducible. We note that using number of processed states as a performance metric is not completely fair, because techniques differ in their speed of exploration (e.g., random walk is faster than search which stores and matches states). However, this effect is not very significant and it does not distort the main message of our results.

As a second measure we use the length of counterexample returned by a technique. The length is measured as a number of states in the counterexample.

As a third measure we consider coverage metrics. Coverage metrics are used particularly in the testing community (see e.g. [2,15,17,18]). Coverage metrics measure the coverage of a model's behaviour by a technique. We have selected four different coverage metrics:

**Statement coverage** Counts the number of visited statements (positions of program counters of every process in the model).

**Branch coverage** Counts the number of visited branches (transitions of every process in the model).

**Condition coverage** Counts the number of combinations of truth values of expressions in conditions within each visited state.

**Multiple condition coverage** Counts within each visited state the number of different truth values of all atomic expressions that occur in conditions.

In case of coverage metrics, we fix the number of states (50,000) and measure a coverage achieved after this limit.

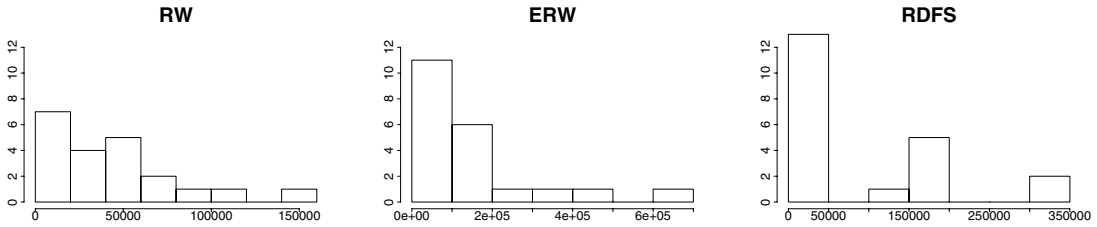



Fig. 3. Histograms showing the number of processed states for 21 runs of three randomized techniques over the model firewire_link.x1 (x-axis: states, y-axis: number of cases).

### *3.3 Randomized Techniques*

Several of the studied techniques use randomization (RW, ERW, RDFS). For these techniques we run 21 runs. Note that the results of these runs do not have a normal distribution. For RW the results are usually more like Poisson distribution, for RDFS the results can fall into several distant regions, i.e., the technique can be either very fast or very slow, but nothing in between (see Fig 3.). For this reason, we do not report mean value and standard deviation, as is usual. Rather we use the median value, because we consider it to be more meaningful (note that the median can be used meaningfully even if there are several runs which do not terminate). For a comparison of counterexamples we use a counterexample returned by the run with median number of processed states.

### *3.4 Selection of Techniques and Parameters*

In our experiments we use nine techniques, which are described in Section 2.2. To select these nine techniques, we specified a large set of techniques that covered many combinations of building blocks (see Section 2.1), particularly we tried different values of parameters and different types of search order. Then we run a preliminary version of experiments with all these techniques. Then we analyzed the correlations among technique's performance (in the same way as shown in Fig. 5) and we found that techniques which differ only slightly (e.g., by parameter values) have very similar performance. From each group of similar techniques we selected one with good results (note that since we use several performance measures, we cannot say which is "the best").

## 4 Experiments

In this section we report the results of experiments. The results show complementarity of techniques in two aspects. At first, each technique works well on different models. At second, the performance differs with respect to the number of visited states, the length of reported counterexample, and the model coverage. We also discuss the impact of selection of models on results.

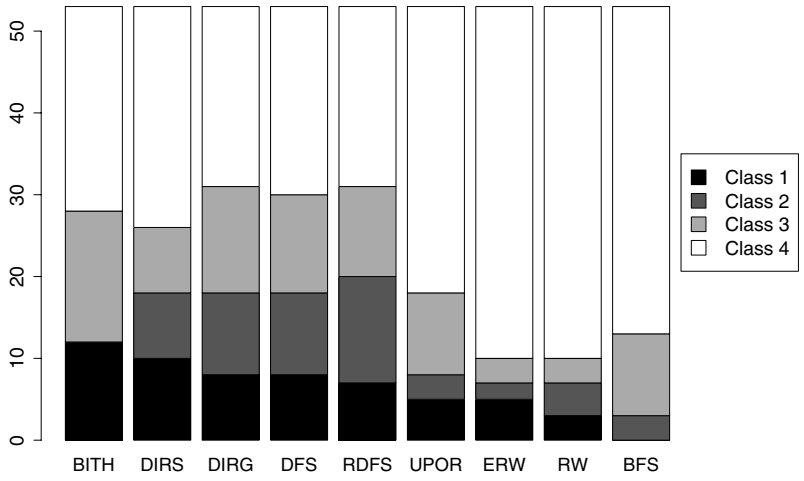



Fig. 4. Comparison of techniques for error detection.

### *4.1 Error Detection*

In order to make the results easier to understand, we normalize the performance of each technique for each verification problem (i.e., model and target goal) relatively to the best technique for the verification problem. More specifically, we classify the technique's performance in one of 4 classes. Let $N_T$ be the number of states processed by a technique $T$ and $N_B$ be the number of states processed by the technique which is the best for a given verification problem. Then the performance of $T$ over the problem is classified as follows:

$$\begin{aligned} &\text{Class 1} && N_B = N_T \\ &\text{Class 2} && N_B < N_T \leq 2 \cdot N_B \\ &\text{Class 3} && 2 \cdot N_B < N_T \leq 10 \cdot N_B \\ &\text{Class 4} && 10 \cdot N_B < N_T \end{aligned}$$

Fig. 4. gives a summary of our experiments. For each technique we report the number of cases in which it was classified to each class. There are significant differences among techniques — BITH and DIRG are clearly more successful than BFS. However, there is no dominant technique and for each technique there are cases where it works well.

Fig. 5 illustrates the complementarity of studied techniques. In order to compute correlations among techniques we consider normalized numbers of states processed by each technique ($N_B/N_T$). For each pair of techniques we compute the correlation coefficient and visualise it by colors. The figure shows that there is a low degree of correlation among studied techniques. To a certain extent, this result is caused by our selection process (see Section 3.4).

Note that sometimes even similar techniques can yield different results. This is particularly the case of DFS and RDFS. These techniques achieve very similar

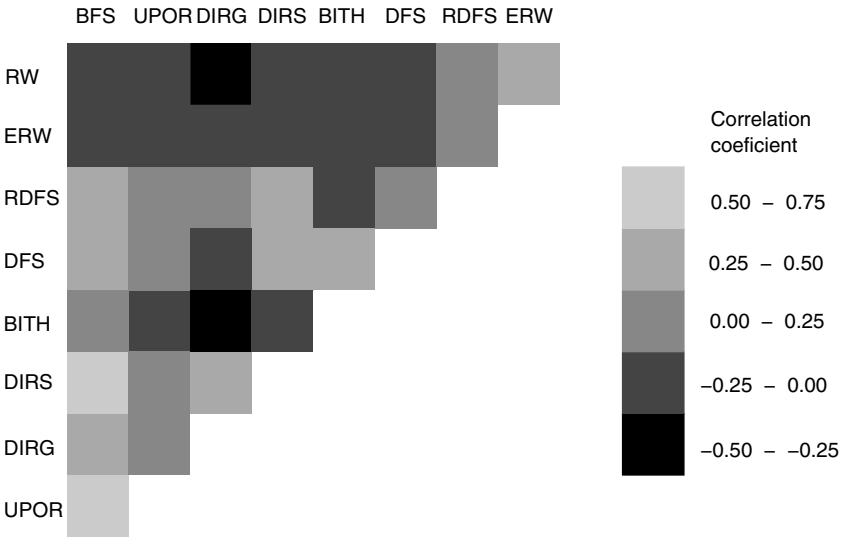



Fig. 5. Correlations among techniques. A light color means high correlation (techniques work/fail on same verification problems), a dark color means low correlation (techniques work/fail on different verification problems).

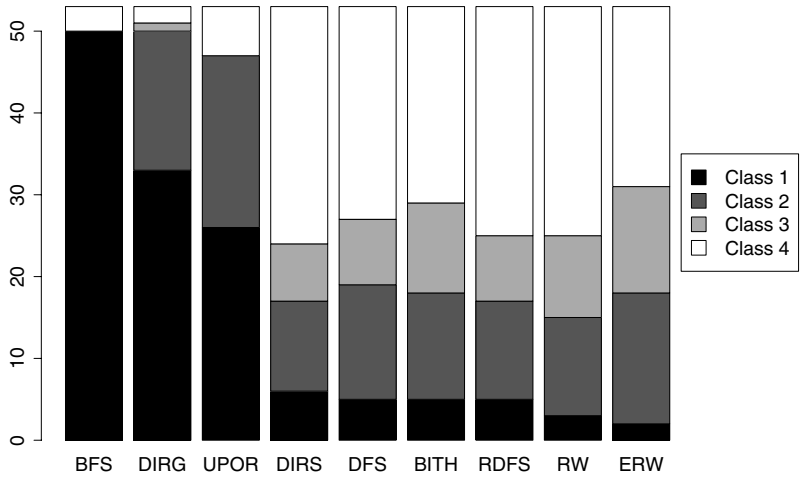



Fig. 6. Lengths of counterexamples produced by techniques. The graph uses the same type of classification as for number of visited states.

overall results (see Fig. 4), but their performance is only loosely correlated, i.e., each of them works well on different models (see Fig 5).

### *4.2 Length of Counterexample*

In applications, we are also concerned with the length of the counterexample. Shorter counterexamples are easier to analyse and understand, therefore it is important whether a technique returns short counterexamples. Fig. 6. gives a comparison of techniques with respect to the length of produced counterexample (the same type of classification as before is used). If BFS terminates, than it produces the shortest possible counterexample (by definition of the technique). Two other techniques,

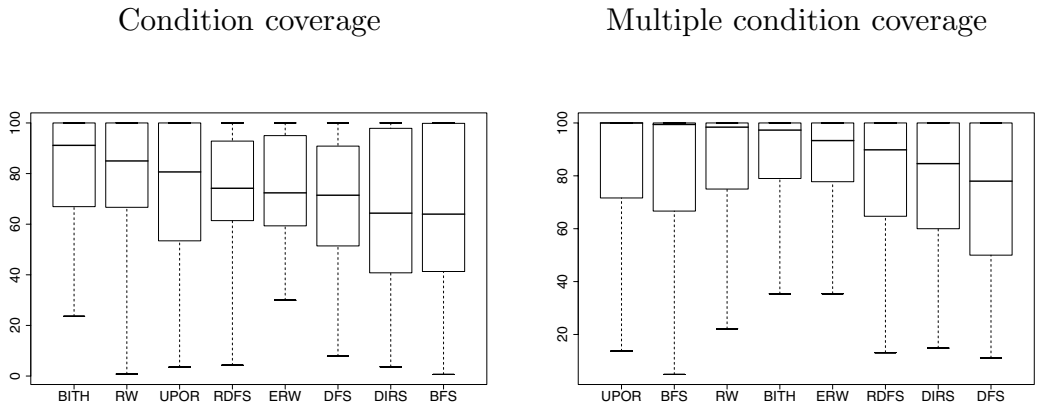



Fig. 7. Comparison of techniques for two coverage metrics; the results are normalized by the best technique (best = 100); results are shown using the boxplot method (minimum, 1st quartile, median, 3rd quartile, maximum) and sorted by the median.

DIRG and UPOR (both are based on BFS), produce short counterexamples most of the time. Other techniques are significantly worse. Note that DIRG is the only one which produces short counterexamples and at the same time it is often successful, i.e., there is a certain trade-off between the performance of a technique and a length of computed counterexamples.

### *4.3 Coverage*

All techniques can usually achieve the optimal coverage for statement coverage and branch coverage, i.e., these metrics are not suitable for distinguishing the performance of techniques. Results for condition coverage and multiple condition coverage, however, show more variability, see Fig. 7. Note that in this case we do not use the DIRG technique, because there is no goal to guide the directed search.

Again, we see that there is no dominant technique, each technique works in some cases and fails in others. The successfulness of techniques differs for the two coverage metrics and it is different from successfulness for error detection. For example, the UPOR and RW techniques, which do not work very well for error detection, are quite good for achieving coverage. Only BITH technique has consistently good results.

### *4.4 The Impact of Model Selection*

How much does the selection of models influence results of our experimental study? We study this question from several perspectives.

Some errors can be easily found by several techniques, whereas others can be tackled efficiently only by one technique. This is illustrated in Fig. 8., which shows that from the 54 cases, in 14 cases the selection of a technique is crucial (1 technique works well, other do not) and in another in 7 cases the selection is still very important (only 2 techniques work well). On the other hand, in 7 cases the selection of a technique is not very important because 4 from 9 techniques are in Class 1 or Class 2.

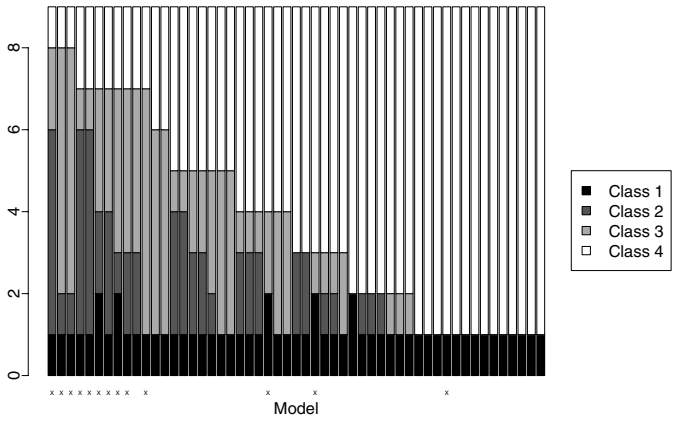



Fig. 8. The performance of techniques over individual models. Each column corresponds to one model and shows the number of techniques in each class. The marked columns correspond to toy models.

Table 1
Performance for different types of models. For each model type and technique we report the average classification (1 to 4 values).

| model type | DIRG | DIRS | RDFS | DFS | BITH | UPOR | ERW | RW | BFS |
|---|---|---|---|---|---|---|---|---|---|
| all | **2.9** | **2.9** | **2.9** | **2.9** | 3.0 | 3.4 | 3.5 | 3.6 | 3.6 |
| random half | 2.6 | 2.9 | **2.5** | 3.0 | 3.2 | 3.6 | 3.5 | 3.6 | 3.7 |
| mutex | 3.2 | 3.6 | **2.4** | 3.5 | 3.0 | 4.0 | 3.3 | 3.5 | 4.0 |
| protocols | **2.8** | 3.2 | 3.4 | 3.0 | 2.9 | 3.3 | 3.7 | 3.5 | 3.8 |
| toy | 2.6 | 2.2 | 2.3 | **2.1** | 2.6 | 2.6 | 3.5 | 4.0 | 3.0 |
| complex | 3.2 | 2.8 | 3.4 | 3.0 | **2.6** | 3.0 | 3.7 | 3.4 | 3.9 |

Table 1. shows what happens when we restrict our attention to a specific type of models. The table shows the average classification for each technique, compare the first line in Table 1. with Fig. 4. The first row in the table gives the overall results. The second row in the table gives the result for a randomly selected half of the models. The results are similar to overall results; this supports our conviction that our set of models is sufficiently large so that results are not influenced by the selection of models.

BEEM [20] provides a classification according to application domain and according to complexity of the model (as a toy, simple, and complex models). These classifications are used in the rest of Table 1. When we consider only models from a particular application domain we see that some techniques do not work at all (e.g., BFS and UPOR for mutual exclusion algorithms). Nevertheless, even in this case there is no clear winner.

The restriction to toy/complex models shows how results of experiments can be distorted by the usage of (only) toy models. For toy models, the average classifications are very low (compared to other model types). The low average classification means that usually many techniques are successful on each model, i.e., that the selection of a technique does not matter (see also Fig. 8). If we order techniques by average classification, we get quite different order for toy and complex models.

# 5 Summary

*There is no single best technique. Never mind, it does not matter.* Some techniques work well for error detection (directed search, randomized DFS), other techniques can achieve good coverage (bitstate hashing with refinement, random walk, UPOR) or produce short counterexamples (BFS). However, given the accessibility of hardware for parallel computation (multi-core processors, networks of workstations, clusters), we can run many techniques in parallel (independently, with no communication overhead) and thus combine strength of different techniques.

*It is important to focus also on complementarity of techniques, not just on their perfectness.* Tuning of parameters of a single technique, in order to make it as fast as possible, is not a good way forward. Our experiences suggest that tuning of parameters can improve the performance slightly, but it does not change whether the technique works well on a model. A good example of advantageous complementarity are the techniques BITH and DIRG, both of these techniques work quite well and they complement each other (their performance is inversely correlated).

*It is important to compare a new technique with a large number of previously known techniques.* Usual experimental approach is to compare a new technique with one or two classical techniques; the focus is on showing an improvement over some benchmark set. However, there may be different techniques which works significantly better for many problems in the benchmark set than used classical techniques and thus the reported improvement may be rather irrelevant. Therefore, it is important to do the comparison with a nontrivial set of complementary techniques. Our work suggests that for error detection the following set of well-known and easy-to-implement techniques is reasonably complementary and should be used in subsequent experiments: BFS, DFS, randomized DFS, directed search, bitstate hashing with refinement, and random walk.

*Both models and goals have significant and hard to predict impact on the performance of techniques.* One of our original goals was to predict the performance of techniques on a given model by parsing the syntax of the model and/or taking a small sample of the state space. Now we consider this goal to be unrealistic, mainly due to the impact of the goal on technique's performance. Note that the importance of a goal selection is neglected in previous studies (the goal is usually not even stated).

Verification problems also differ in the number of techniques which work well over them. For some problems it is not very significant which technique do we use, all techniques need similar time to find the error. For some verification problems, however, one technique can defeat other techniques utterly.

# 6 Future Work

In this work we analyse and evaluate only techniques for detection of safety errors. Similar analysis, with the main goal of finding a set of good complementary

techniques, should be done at least for the following, practically important cases [4]:

- verification of safety properties,
- falsification of liveness properties,
- verification of liveness properties.

In this work we advocate the use of parallel independent runs of several techniques. The independence of individual runs means that there is no communication overhead. However, it may be advantageous to collect and share some information among different techniques, see [21,8] for general proposals of such a setting. It may be interesting to implement a globally controlled parallel run using the techniques discussed in this paper.

# References

[1] 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`.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proc. of International symposium on Software testing and analysis (ISSTA '02)*, pages 123–133. ACM Press, 2002.

[3] L. Brim, I. Černá, and M. Nečesal. Randomization helps in LTL model checking. In *Proc. of PAPM-PROBMIV Workshop*, number 2165 in LNCS, pages 105–119. Springer, 2001.

[4] P. C. Dillinger, P., and Manolios. Bloom Filters in Probabilistic Verification. *Formal Methods in Computer-Aided Design (FMCAD)*, 3312:367–381, 2004.

[5] M B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proc. of International Conference on Software Engineering (ICSE '07)*, pages 3–12. IEEE Computer Society, 2007.

[6] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. of Foundations of software engineering (SIGSOFT '06/FSE-14)*, pages 92–104. ACM Press, 2006.

[7] 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. Springer, 2001.

[8] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs. *Concurrency and Computation: Practice and Experience*, 19(3):267–279, 2007.

[9] P. Godefroid. Model checking for programming languages using verisoft. In *Proc. of Principles of programming languages (POPL '97)*, pages 174–186. ACM Press, 1997.

[10] A. Groce and W. Visser. Heuristics for model checking Java programs. *Software Tools for Technology Transfer (STTT)*, 6(4):260–276, 2004.

[11] G. J. Holzmann. An analysis of bitstate hashing. In *Proc. of Protocol Specification, Testing, and Verification*, pages 301–314. Chapman & Hall, 1995.

[12] G.J. Holzmann and M.H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

[13] T. Kropf. Software bugs seen from an industrial perspective or can formal methods help an automotive software development?, 2007. Invited talk on CAV'07.

[14] A. L. Lafuente. *Directed Search for the Verification of Communication Protocols*. PhD thesis, Albert-Ludwigs-Universitat Freinburg, 2003.

[4] Note that there exist techniques which can cope with all of these cases. However, as we argue in Introduction, such "universal" techniques are not very important from the practical point of view.

[15] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report LCS-TR-921, MIT Computer Science and Artificial Intelligence Laboratory, September 2003.

[16] P. Moravec. Approximations of state spaces reduced by partial order reduction. Submitted to SOFSEM'08.

[17] C. Pasareanu, R. Pelánek, and W. Visser. Test input generation for red black trees using abstraction. In *Proc. of Automated Software Engineering (ASE'05)*, pages 414–417. ACM, 2005.

[18] C. Pasareanu, R. Pelánek, and W. Visser. Test input generation for java containers using state matching. In *Proc. of International Symposium on International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 37–48. ACM, 2006.

[19] C. Pasareanu, R. Pelánek, and W. Visser. Predicate abstraction with under-approximation refinement. *Logical Methods in Computer Science*, 3(1), 2007.

[20] R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007. Available at `http://anna.fi.muni.cz/models`.

[21] R. Pelánek. Model classifications and automated verification. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'07)*, 2007. To appear.

[22] 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.

[23] N. Rungta and E. G. Mercer. Generating counter-examples through randomized guided search. In *Model Checking Software*, volume 4595 of *LNCS*, pages 39–57. Springer, 2007.

[24] N. Rungta and E. G. Mercer. Hardness for explicit state software model checking benchmarks. In *Proc. of Software Engineering and Formal Methods (SEFM'07)*. IEEE Computer Society, 2007.

# Chapter 8

# Test Input Generation for Java Containers using State Matching

The topic of this paper lies on the border between model checking and testing. We are concerned with test input generation for Java containers and we try to do it with the use of explicit model checker (Java PathFinder). We compare several techniques: exhaustive techniques based on explicit model checking, lossy techniques which are based on explicit model checking but do not visit all states, and also random selection of inputs. The basic metric used for comparison is testing coverage (more specifically, we use a predicate coverage metric).

The first surprising result is that random selection, despite its simplicity, performs surprisingly well. Nevertheless, more sophisticated techniques can beat random selection on complex inputs (e.g., implementation of a red-black tree). The most successful technique seems to be the explicit search with abstract matching of states, but similarly to our other evaluations it is not possible to declare a single universal winner.

This paper was published in proceedings of International Symposium on International Symposium on Software Testing and Analysis (ISSTA) in 2006 [55], preliminary version of the research was reported as a short paper in Automated Software Engineering in 2005:

- C. Pasareanu, R. Pelánek, and W. Visser. Test input generation for java containers using state matching. In *Proc. of International Symposium on International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 37–48. ACM, 2006.
- C. Pasareanu, R. Pelánek, and W. Visser. Test input generation for red black trees using abstraction. In *Proc. of Automated Software Engineering (ASE'05)*, pages 414–417. ACM, 2005.

The author of the thesis is one of three coauthors of the paper and has contributed particularly to the experimental design, data analysis, and interpretation of results.

# Test Input Generation for Java Containers using State Matching

Willem Visser
RIACS/NASA Ames
Moffett Field, CA 94035, USA
wvisser@email.arc.nasa.gov

Corina S. Păsăreanu
QSS/NASA Ames
Moffett Field, CA 94035, USA
pcorina@email.arc.nasa.gov

Radek Pelánek
Masaryk University
Brno, Czech Republic
xpelanek@fi.muni.cz

## ABSTRACT

The popularity of object-oriented programming has led to the wide use of container libraries. It is important for the reliability of these containers that they are tested adequately. We describe techniques for automated test input generation of Java container classes. Test inputs are sequences of method calls from the container interface. The techniques rely on state matching to avoid generation of redundant tests. *Exhaustive techniques* use model checking with explicit or symbolic execution to explore *all* the possible test sequences up to predefined input sizes. *Lossy techniques* rely on abstraction mappings to compute and store abstract versions of the concrete states; they explore *under-approximations* of all the possible test sequences.

We have implemented the techniques on top of the Java PathFinder model checker and we evaluate them using four Java container classes. We compare state matching based techniques and random selection for generating test inputs, in terms of testing coverage. We consider basic block coverage and a form of predicate coverage - that measures whether all combinations of a predetermined set of predicates are covered at each basic block. The exhaustive techniques can easily obtain basic block coverage, but cannot obtain good predicate coverage before running out of memory. On the other hand, abstract matching turns out to be a powerful approach for generating test inputs to obtain high predicate coverage. Random selection performed well except on the examples that contained complex input spaces, where the lossy abstraction techniques performed better.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

## General Terms

Reliability, Experimentation, Verification


*ISSTA'06* July 17–20, 2006, Portland, Maine, USA.


## Keywords

Software Model Checking, Unit Testing, Symbolic Execution, Abstraction, Random Testing

## 1. INTRODUCTION

Object oriented programming is fast becoming the paradigm of choice in everything from web applications to safety critical flight control software in the next generation of NASA manned missions. Modern object oriented languages typically come with libraries of container classes that are heavily reused without much concern given to the correctness of these container implementations. To ensure the reliability of systems built with such containers, they must be tested adequately. The large number of test inputs for thorough testing makes *automated* test input generation imperative.

This paper presents techniques for automated test input generation of container classes that use *state matching* to avoid generation of redundant tests. Test inputs are sequences of method calls from the container interface, that cover the relevant structural and behavioral aspects of the code. We use a model checker to *exhaustively* try all combinations of method calls and parameters to these calls up to a specified limit, but after each call the state of the container is examined to see if it can be "matched" with a previously stored state; if so, that sequence is discarded, if not the search continues with the next call.

During this search the testing coverage is measured and whenever new coverage is obtained the sequence of calls to achieve that coverage is recorded. We consider basic block coverage, as a representative example of simple structural coverage, and a form of predicate coverage [3] which measures the coverage of all the combinations of program predicates; predicate coverage is more difficult to achieve than basic block coverage.

The large amount of input data necessary to test the containers made us investigate an alternative technique, which uses symbolic, rather than explicit, execution, i.e. instead of concrete parameters to interface method calls it uses symbolic parameters. Symbolic execution [22] manipulates symbolic states, representing *sets* of concrete states, and *partitions* the input domain of the interface methods into non-overlapping subdomains, according to different paths that are taken during symbolic execution. Therefore, this technique has the potential to yield significant improvement over the "explicit" exhaustive technique. We describe a method for examining when a symbolic state is *subsumed* by another symbolic state. This is used for state matching to determine whether a test specific sequence can be discarded by the

model checker. Furthermore, we show that this approach scales better than the exhaustive explicit technique.

Even with state matching, the number of test sequences that needs to be explored with the explicit or symbolic techniques quickly becomes intractable – due to the state space explosion problem. We therefore define abstraction mappings to be used for state matching, to further reduce the state space explored by the model checker. More precisely, for each explored state, the model checker computes and stores an abstract version of the state, as specified by the abstraction. State matching is then used to determine if an abstract state is being re-visited. This technique is *lossy*, since parts of the feasible input sequences can be discarded due to abstraction. We introduce here a simple but powerful abstraction that records only the structure or *shape* of the container, while it discards the data stored in the container. We show that this lossy technique is the most effective with respect to coverage achieved.

We have implemented the techniques in a unified framework and we evaluate them on several container classes. The framework is built on top of the Java PathFinder [19, 28] model checker that already supports symbolic execution. Our framework also incorporates a technique based on random selection – and we use it as a point of comparison with the other techniques. As mentioned, we evaluate the techniques in terms of basic block and predicate coverage achieved by the generated test sequences. Predicate coverage is motivated by the observation that certain subtle program errors (that go undetected in the face of 100% basic block coverage) may be due to complex correlations between program predicates [3]. Therefore predicate coverage is a good addition to basic block coverage for evaluating test input generation strategies.

Although there is a lot of related work (presented in Section 6), we are not aware of a framework or a study that implements and compares explicit and symbolic techniques for test input generation with random selection in terms of structural coverage, let alone in terms of predicate coverage, as we do here.

The contributions of the paper are the following:

- Framework for test input generation for Java container classes. The framework incorporates explicit and symbolic techniques and uses state matching to avoid generation of redundant tests.
- Automated support for *shape abstraction* to be used during state matching. The abstraction can be used with both explicit and symbolic techniques. The abstraction is shown to be the most effective, as compared to all the other techniques.
- Evaluation of test generation approaches on four non-trivial Java container classes measuring the performance in achieving both a simple structural coverage and a form of predicate coverage. The evaluated approaches range from exhaustive testing, partition testing using symbolic execution and random testing.

# 2. BACKGROUND

We describe here the Java PathFinder (JPF) tool [19, 28] and its extension with a symbolic execution capability. Section 4 shows how to use JPF for test input generation.

## 2.1 Java PathFinder

JPF is an explicit-state model checker for Java programs that is built on top of a custom-made Java Virtual Machine (JVM). JPF handles all the Java language features and it also supports program annotations that are added to the programs through method calls to a special class `Verify`. We used the following methods from `Verify`:

`beginAtomic() ... endAtomic()` specify that the execution of the enclosed block should proceed atomically.

`random(n)` returns values $[0, n]$ nondeterministically.

`ignoreIf(cond)` forces the model checker to backtrack when `cond` evaluates to true.

By default, JPF stores all the explored states and it backtracks when it visits a previously explored state. Alternatively, the user can customize the search (by forcing the search to backtrack on user-specified conditions) and it can specify what part of the state (if any) to be stored and used for matching. We used these features to implement our test generation techniques that use model checking with abstract matching and random search. JPF also supports various search heuristics [13], that can be used to guide the model checker's search.

## 2.2 Symbolic Execution

Symbolic execution [22] allows one to analyze programs with un-initialized inputs. The main idea is to use *symbolic values*, instead of actual data, as input values and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs.

The *state* of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* (`PC`) and a program counter. The path condition is a (quantifier free) boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path.

In previous work [21], we have extended JPF to perform symbolic execution for Java programs. The approach handles dynamically allocated data, arrays, and multi-threading. Programs are instrumented to enable JPF to perform symbolic execution; concrete types are replaced with corresponding symbolic types and concrete operations are replaced with calls to methods that implement corresponding operations on symbolic expressions. A Java implementation of the Omega library [25] is used to check satisfiability of numeric path conditions (for linear integer constraints).

# 3. TEST INPUT GENERATION

In this section we present our framework for generating test inputs for Java container classes. We illustrate our approach on a Java implementation of a binary search tree (see Figure 1). Each tree has a `root` node. Each node has an integer `elem` field and `left` and `right` children. Values are added and removed from the tree using the `add` and `remove` methods respectively.

A test input for `BinTree` consists of a sequence of method calls in the class interface (e.g. `add` and `remove`), with corresponding method arguments, that builds relevant object states and exercise the code in some desired fashion. Here is an example of a test input for `BinTree`:

```
class Node { ...
  public int elem;
  public Node left, right;
}
public class BinTree {
  private Node root;
  ...
  public void add(int x) { ... }
  public boolean remove(int x) { ... }
}
```

**Figure 1: Java declaration of a binary tree**

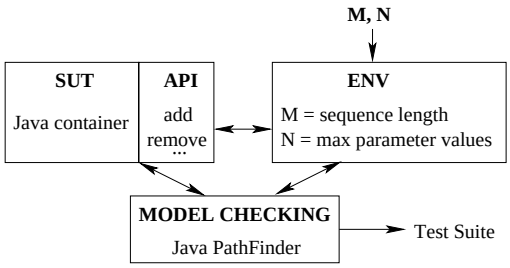



**Figure 2: Test Generation Framework**

```
BinTree t = new BinTree();
t.add(1); t.add(2); t.remove(1);
```

## 3.1 Framework

The framework for test input generation is illustrated in Figure 2. For each container (the system under test `SUT`) we built a *nondeterministic* environment `ENV`, i.e. a test driver that executes *all* sequences of API method calls up to a user-specified size `M`. JPF is used to enumerate all these sequences. The model checker's state matching capability avoids the exploration (and generation) of redundant tests.

The framework implements the following techniques (they are described in detail in the next section):

- exhaustive explicit execution with state matching
- explicit execution with abstract matching
- symbolic execution with subsumption checking
- symbolic execution with abstract matching
- random selection (no state matching).

For the techniques that use explicit execution or random selection, the user also needs to specify the range of values for the method parameters `[0, N-1]`. `N` is not needed when performing symbolic execution, since in this case the methods are executed with symbolic parameters.

## 3.2 Testing Coverage

The model checker analyzes the nondeterministic environment and it generates method sequences that achieve the desired testing coverage. We use basic block coverage, as a representative example of a widely used structural coverage measure. At each basic block, we also measure the coverage of all the combinations of a set of predicates chosen

```
static int M; /* sequence length */
static int N; /* parameter values */

static BinTree t = new BinTree();
public static void main(String[] args) {...
1:  for (int i=0;i<M;i++) {
2:    Verify.beginAtomic();
3:    int v = Verify.random(N-1);
4:    switch (Verify.random(1)) {
5:      case 0: t.add(v); break;
6:      case 1: t.remove(v); break;
      }
7:    Verify.endAtomic();
8: /* Verify.ignoreIf(store(abstractMap(t))); */
    }  }
```

**Figure 3: Environment for explicit search**

from conditions in the source code. We refer to the latter as *predicate coverage*, although it is strictly speaking only a simplified version of the predicate coverage as defined in [3] – where all predicates in the code are used rather than a subset as done here. For our purposes this is sufficient, since we are not measuring the adequacy of a test suite, but rather use it as a measure to compare test generation strategies.

The code of the methods is instrumented to record the coverage, e.g. basic block or predicate coverage. Whenever the model checker executes an uncovered block (or a new predicate combination), it outputs the current test sequence.

We should note that basic block coverage is a simple structural coverage whereas predicate coverage requires more behaviors (paths) to be followed through the code to obtain good coverage – this is in part due to the predicates coming from different portions of the code.

Also note that we have no way of computing the optimal predicate coverage for each SUT (hence we will refer to the highest observed coverage as the optimal). We use the rate of increase in the number of predicate combinations covered to evaluate whether a particular testing technique is helpful in covering paths not previously executed.

## 3.3 Testing Oracles

Method post-conditions can be used as test oracles to check the correctness of container methods. JPF also supports partial correctness properties given as assertions in the code and temporal logic specifications. In the experiments reported in this paper, we used JPF to check just absence of run-time errors, e.g. absence of uncaught exceptions.

# 4. TEST GENERATION TECHNIQUES

In this section we describe the techniques that are implemented in the test input generation framework, namely exhaustive explicit execution, explicit execution with abstract matching and symbolic execution (with subsumption checking and with abstract matching). We also describe how to use random selection for test input generation and we discuss the search order used in the model checker.

## 4.1 "Classical" Exhaustive Explicit Execution

We illustrate this technique using the `BinTree` example introduced in the previous section. The testing environment

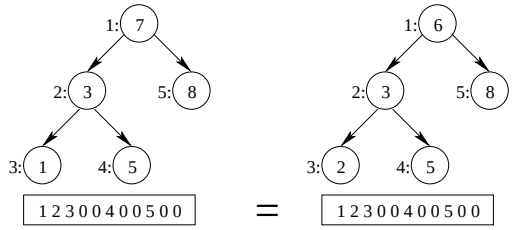



**Figure 4: Abstraction recording shapes**

is illustrated in Figure 3. The environment executes atomically *all* the sequences of `add` and `remove` methods up to the pre-specified sequence size `M`. The input values are chosen nondeterministically from range `[0, N-1]`. Line 8 is discussed in Section 4.2. As discussed, "classic" explicit state model checking is then used to search the state space of the program defined in Figure 3. The model checker's *default* state matching capability is used to avoid exploration and generation of redundant test sequences.

This straightforward approach does not scale well for large values of `M` and `N` – the number of possible test sequences becomes quickly intractable (the state space explosion problem). One way to address this problem is to use heuristic search; JPF supports several heuristics (guided search, beam search). Another solution is to perform explicit execution with abstract matching as described below.

## 4.2 Explicit Execution with Abstract Matching

The idea is to use the model checker to perform the explicit execution of all the possible method sequences (as above) but to store *abstract* versions of the explored program states, and use these abstract states to perform state matching (and to backtrack if an abstract state has been visited before). This effectively explores an under-approximation of the space of possible method executions.

In order to apply this technique for `BinTree` we use the environment illustrated in Figure 3, in which statement `8: Verify.ignoreIf(store(abstractMap(t)))` is included.

`abstractMap` computes an abstraction of the concrete container state of the binary tree referenced by `t`;

`store` directs the model checker to store the computed abstraction;

`Verify.ignoreIf` directs the model checker to backtrack if it has seen this abstraction before.

Note that state matching is now performed only on the state of the container object (referenced by `t`). This allows us to abstract away the information that is irrelevant to test generation, i.e. the values of local variables `i` and `v` are no longer considered to be part of the state.

JPF provides *automated support* for two powerful abstractions, that we have found useful in the analysis of containers.

- *The shape abstraction* records only the (concrete) heap shape of a container, while it abstracts away the data fields from each container element. The abstraction is illustrated in Figure 4, which depicts two binary search trees. Circles denote tree nodes; numbers inside circles denote the `elem` values; null nodes are not represented. The trees have the same heap shape - hence they will be matched during model checking (although the actual `elem` values are not the same). Heap shapes are represented in a normalized form, as sequences of integers (depicted in rectangles in Figure 4), and are obtained through a process called *linearization* [18, 32]. The linearization of an object (e.g. the tree root) starts from the root and traverses the heap in depth first search order; it assigns a unique identifier to each object and it backtracks when it detects a cycle; null pointers have values 0. Comparing shapes reduces to comparing sequences. Linearization has complexity $O(n)$ (where $n$ is the number of heap nodes that are reachable from the root) and it can be performed efficiently during garbage collection.

- *The complete abstraction* records the shape of the container together with *all* the data fields from each container element. Strictly speaking, this is not really an abstraction (there is no information loss) but rather a canonical complete encoding of the container state, similar to the linearization used for representing the complete concrete heap (shape plus data), to achieve heap symmetry reduction in model checking [18].

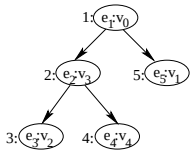
```
static int M; /* sequence length */

static BinTree t = new BinTree();
public static void main(String[] args) {...
1:  for (int i=0;i<M;i++) {
2:    Verify.beginAtomic();
3:    SymbolicInt v = new SymbolicInt(''v''+i);
4:    switch (Verify.random(1)) {
5:      case 0: t.add(v); break;
6:      case 1: t.remove(v); break;
      }
7:    Verify.endAtomic();
8:    Verify.ignoreIf(checkSubsumptionAndStore(t));
    }  }
```

**Figure 5: Environment for symbolic search**

## 4.3 Symbolic Execution with State Matching

The test generation techniques that we have presented so far use concrete values for the method parameters. We now present an alternative technique, which assigns symbolic values for the input parameters, and it uses JPF to perform *symbolic* execution of the data structure's methods. The model checker manipulates symbolic states which describe *sets* of concrete states. As a result, this technique has the potential to yield significant improvements over explicit execution techniques.

To generates test inputs for `BinTree` (with symbolic execution) we use the environment illustrated in Figure 5. The environment is similar to what we had before, except now the type of variable $v$ is `SymbolicInt` (rather than `int`) and `v` is assigned a new symbolic value each time the `for` loop is executed. Moreover, the code of the `add` and `remove` methods is instrumented to enable JPF to perform symbolic execution. State matching (line `8`) is discussed below.




Shape: 1 2 3 0 0 4 0 0 5 0 0

PC: $v_0 < v_1 \wedge v_3 > v_2 \wedge$
$v_3 < v_0 \wedge v_3 < v_4 \wedge$
$v_5 < v_1 \wedge v_5 > v_0 \wedge$
$v_4 < v_0$

**Figure 6: A symbolic state**

Here is an example of a generated test sequence:

```
BinTree t = new BinTree();
t.add(v0);t.add(v1); t.remove(v2);

PC: v2 == v1 && v2 < v0 && v1 < v0;
Solution: v0: 1, v1: 0, v2: 0;
```

The path condition (PC) encodes the constraints on the input parameters. JPF also solves the constraints and it provides numeric solutions to be used as the concrete parameters for the actual test input. Our implementation is currently handling linear integer constraints. Other numeric domains could be handled similarly (provided the availability of appropriate decision procedures).

Let us now analyze a symbolic object state at line `8` - as illustrated in Figure 6. In each node, we write the symbolic value of the `elem` field, e.g. $e_1 : v_0$ means that the `elem` field of node 1 ($e_1$) has symbolic value $v_0$ ($v_0 ... v_5$ denote the symbolic values that were given as input parameters). The path condition encodes the constraints on the input values, and it may refer to symbolic values that are no longer stored in the tree, e.g. $v_5$. Intuitively, this means that this particular tree was created by a sequence that contained a `remove` call which removed value $v_5$ from the tree.

For state matching, we normalize the representation for symbolic states, using existential quantifier elimination. Intuitively, we are only interested in the relative order of the elements in the tree. For the example presented in Figure 6, we write the following constraints:

$$\exists v_0, v_1, v_2, v_3, v_4, v_5 :$$
$$e_1 = v_0 \wedge e_2 = v_3 \wedge e_3 = v_2 \wedge e_4 = v_4 \wedge e_5 = v_1 \wedge PC$$

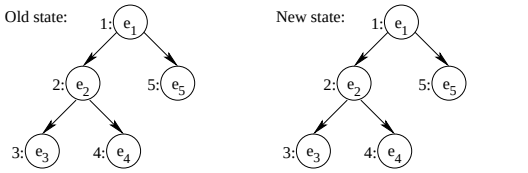
We use the Omega library for existential quantifier elimination, which results in the following simplified constraints:

$$e_1 > e_2 \wedge e_2 > e_3 \wedge e_2 < e_4 \wedge e_5 > e_1 \wedge e_4 < e_1$$

The normalized symbolic state (shape plus simplified constraints) is used for state storing and comparison. A symbolic state encodes all the concrete states that have the same shape and whose elements satisfy the constraints.

Since symbolic states represent multiple concrete states, state matching involves checking *subsumption* between states. Let $s_1$ and $s_2$ be two symbolic states, and let $\gamma(s_1)$ and $\gamma(s_2)$ denote the sets of concrete states represented by $s_1$ and $s_2$ respectively. A symbolic state $s_1$ *subsumes* another symbolic state $s_2$, written $s_1 \supseteq s_2$, if the set of concrete states represented by $s_1$ contains the set of concrete states represented by $s_2$, i.e. $\gamma(s_1) \supseteq \gamma(s_2)$.

We check subsumption of two object states by checking that they have the same shape (as given by linearization) and that there is a valid implication between the corresponding constraints - we use the Omega library for checking validity. For example, in Figure 7, *Old state* $\supseteq$ *New state*: they have the same shape and the following implication is valid:




Constraints:
$e_1{>}e_2 \wedge e_2{>}e_3 \wedge e_2{<}e_4 \wedge e_5{\geq}e_1 \quad \Leftarrow \quad e_1{>}e_2 \wedge e_2{>}e_3 \wedge e_2{<}e_4 \wedge e_5{>}e_1$

**Figure 7: Two symbolic states**

$$e_1 > e_2 \wedge e_2 > e_3 \wedge e_2 < e_4 \wedge e_5 \geq e_1$$
$$\Leftarrow e_1 > e_2 \wedge e_2 > e_3 \wedge e_2 < e_4 \wedge e_5 > e_1$$

In our framework, we have implemented bi-directional subsumption checking (line `8` in Figure 5). Let $new_s$ denote a new symbolic object state, and let $old_s$ denote a previously visited and stored symbolic state:

- If $old_s \supseteq new_s$, then `checkSubsumptionAndStore` returns `true` and the model checker backtracks.

- If $new_s \supseteq old_s$ then $old_s$ is replaced with $new_s$ ($new_s$ is "more general" than $old_s$), `checkSubsumptionAndStore` returns `false` and the model checker's search continues (it does not backtrack).

Note that for each heap shape, we would like to store a *disjunction* of constraints, i.e. to store *unions* of symbolic states. In this case the bi-directional subsumption checking would no longer be needed. However, a small technicality prevents us (a bug in the Java implementation of the Omega library that JPF uses).

We should also note that symbolic execution can be used with abstract matching, i.e. replace line `8` in Figure 5 with `Verify.ignoreIf(store(abstractMap(t)))`. In particular, the model checker can use only the *shape* of a symbolic object state, for storing and matching, while discarding the numeric constraints (see the *shape abstraction* in the previous section).

## 4.4 Random Selection

The environment that we use for random selection is similar to the one presented in Figure 3, except that the nondeterminism is solved by random choice. When one (random) run is completed the search is restarted from the initial state and this process is repeated up to a user specified limit. In our experiments, we set the limit on the number of runs to 1000. Random search can be run stand-alone or using JPF – for our experiments we chose to run it inside of JPF. Note that due to technical reasons of JPF implementation, states are stored during the search (but they are never used).

## 4.5 Search Order

When considering an approach that uses state matching to prune the search space of test input sequences up to a *fixed* length we should note that we prefer to use breadth-first search order (BFS) rather than depth-first search (DFS) order. The reason is that DFS can miss portions of the state

space due to matching states that were created by a shorter sequence with those previously generated by a longer sequence (which was truncated due to hitting the length limit). This problem is amplified when considering abstract state matching, i.e. when matching states that are not necessarily identical. Therefore, all the test input generation techniques (besides random search) are used with BFS. Of course BFS also has the desirable characteristic that it produces shorter input sequences.

# 5. EVALUATION

As mentioned, we used the JPF model checking tool (version 3) to implement our testing framework. In particular, we used the *listener* mechanism [19] to observe the sequences of API calls performed and output the sequence when a specific coverage goal is reached. This test listener keeps track of the coverage obtained and calculates the average test input length. The user can specify on the command line the techniques to be used during the analysis. For the experiments we also added a facility to run tests described in a configuration file. The results of each run are collected in a file and a script generates a latex table with the results sorted by coverage.

## 5.1 Experimental Set-up

As a system under test we used Java implementations of four container classes: binary tree (`BinTree` – 154 LOC), binomial heap (`BinomialHeap` – 355 LOC), Fibonacci heap (`FibHeap` – 286 LOC), and red-black tree (extracted from `java.util.TreeMap` – 580 LOC). The methods of these classes were instrumented to measure basic block coverage (which implies statement coverage). At each basic block, we also measure predicate coverage. As mentioned, we have no way of computing the optimal predicate coverage. Therefore, we refer to the highest obtained coverage as optimal. Also note that our coverage numbers are absolute and not percentages as is commonly the case for test adequacy measures.

Note that we only focus here on obtaining code coverage and not on finding errors – this was a conscious decision to avoid bias from different fault seeding approaches. However in the future we would like to investigate whether the tests that obtain high coverage are also likely to detect faults.

Each container class is augmented with an environment as described in Section 4. For `BinTree` and `TreeMap` we only considered `add` and `remove` API calls. For `FibHeap` we also considered `removeMin` and for `BinomialHeap` we considered `add`, `remove`, `extractMin` and `decreaseKey(x,y)`. We considered these additional methods to determine the sensitivity of the techniques to the complexity of the environment.

We compare all the techniques described in the previous section. We divide the techniques into two categories: *exhaustive* and *lossy*. Exhaustive techniques include: explicit state model checking, explicit execution with complete abstract matching (i.e. linearization of a structure with all fields included) and symbolic execution with subsumption checking. Lossy techniques include: explicit and symbolic execution with abstract matching based only on shape and random selection.

For the techniques that use abstract matching, the order in which the state successors are generated can impact the search performance significantly. Therefore, we consider here successors taken in random order and we repeat each experiment 10 times. We run each technique for different values of sequence length `M` (from 1 to 30). For techniques which perform explicit execution we also need to specify the number of input parameters (`N`). In order to make the experiments tractable, we always set `M = N`. Note that this decision is quite justified in the context of containers, since the sequence length typically defines the size of the container, if each value added to the container is unique. If `M > N` then containers of size `M` cannot be generated. For random search we considered sequences up to length 50 and for each length we perform 1000 runs – as with the other lossy techniques each configuration is repeated 10 times. Since we use longer sequences for random it might be argued that `M = N` is unfair, and therefore we did some additional experiments where `M > N` when using random search (discussed later in the section).

## 5.2 Results

The results for exhaustive vs. lossy techniques measuring basic block and predicate coverage are reported in Tables 1–4. These results were produced from more that 10000 runs that took two months CPU time to complete. The results are split into four tables to show the difference in basic block coverage and predicate coverage during exhaustive and lossy search. The exhaustive experiments were preformed on a 2.66GHz Pentium machine running Linux and the lossy experiments on a 2.2Ghz Pentium running Windows 2000. In all cases memory was limited to 1GB.

For each technique we report the best result, i.e. the best coverage that was obtained at the shortest sequence length without running out of memory. Due to the randomization in the lossy techniques, it may happen that some results are obtained "luckily". We report only "stable" results, i.e. results achieved by all runs with a given parameter. The exception is random selection, where we report the best result, even if it happened only on one run. It turned out that the best results were always stable (of course excluding those for random selection), i.e. all 10 runs reported the same results.

We report the coverage, the sequence length (the minimum sequence length at which the coverage was obtained), the time taken (in seconds), memory used (in MB) and the average test sequence length. For the lossy techniques, the time, memory and average length are calculated by taking an average of the 10 runs. Numbers in bold show the maximum sequence length for which exhaustive results could be obtained.

## 5.3 Discussion

We will follow each discussion segment with some concrete conclusions (given in *italics*).

### 5.3.1 Exhaustive vs. lossy techniques

It is interesting to first note the complexity of some of the analyzed containers. For example, one needs sequences of length 14 to obtain basic block coverage (therefore also statement coverage) for `BinomialHeap` – 21 is the optimal coverage. From the exhaustive techniques only the symbolic execution approach using subsumption achieved this coverage. For `FibHeap` the optimal coverage is 25 and none of the exhaustive techniques could obtain this coverage before running out of memory. For `BinTree` and `TreeMap` the exhaustive techniques fared better and only model checking failed to get the optimal coverage for `TreeMap`. It is interesting to note that the two cases for which the exhaustive

**Table 1: Exhaustive Techniques – Basic Block Coverage**

| Container | Technique | Coverage | Seq. Length | Time (s) | Memory (MB) | Avg. Length |
|---|---|---|---|---|---|---|
| BinTree | Model Checking | 14 | 3 | 1 | 4 | 2.2 |
| | Complete Abstraction | 14 | 3 | 1 | 3 | 2.2 |
| | Symbolic Subsumption | 14 | 3 | 1 | 4 | 2.2 |
| BinomialHeap | Model Checking | 17 | **5** | 35 | 129 | 2.8 |
| | Complete Abstraction | 17 | 5 | 8 | 29 | 2.8 |
| | Symbolic Subsumption | 21 | **14** | 910 | 1016 | 4.2 |
| FibHeap | Model Checking | 20 | **5** | 55 | 214 | 3.7 |
| | Complete Abstraction | 24 | 7 | 8 | 19 | 4.2 |
| | Symbolic Subsumption | 24 | 7 | 15 | 54 | 4.2 |
| TreeMap | Model Checking | 37 | **6** | 38 | 243 | 4.2 |
| | Complete Abstraction | 39 | 7 | 9 | 34 | 4.3 |
| | Symbolic Subsumption | 39 | 7 | 15 | 22 | 4.3 |

**Table 2: Lossy Techniques – Basic Block Coverage**

| Container | Technique | Coverage | Seq. Length | Time (s) | Memory (MB) | Avg. Length |
|---|---|---|---|---|---|---|
| BinTree | Shape Abstraction | 14 | 4 | 1 | 3 | 2.2 |
| | Symbolic Shape Abstraction | 14 | 3 | 1 | 4 | 2.2 |
| | Random Selection | 14 | 3 | 7 | 3 | 2.4 |
| BinomialHeap | Shape Abstraction | 21 | 15 | 7 | 8 | 4.3 |
| | Symbolic Shape Abstraction | 21 | 14 | 1084 | 1016 | 4.2 |
| | Random Selection | 21 | 32 | 59 | 9 | 13.2 |
| FibHeap | Shape Abstraction | 25 | 12 | 26 | 34 | 4.4 |
| | Symbolic Shape Abstraction | 25 | 12 | 216 | 608 | 4.5 |
| | Random Selection | 25 | 25 | 41 | 9 | 10.2 |
| TreeMap | Shape Abstraction | 39 | 10 | 2 | 6 | 4.6 |
| | Symbolic Shape Abstraction | 39 | 7 | 7 | 22 | 4.3 |
| | Random Selection | 39 | 10 | 18 | 5 | 7.1 |

techniques fare better have simpler environments than the two cases where these techniques perform less well. All the lossy techniques achieved the optimal basic block coverage and where comparable, they achieved it faster and with less memory than the exhaustive techniques.

*We anticipated that the exhaustive techniques would easily generate all tests to obtain basic block coverage – this is clearly not the case. The lossy techniques seem better suited for achieving code coverage. "Classic" model checking scales poorly even for the basic block coverage.*

### 5.3.2 Symbolic execution

With the exception of `BinTree` – which is the simplest example – none of the exhaustive techniques obtain the optimal predicate coverage. With one exception, the symbolic execution with subsumption performs the best of the exhaustive techniques for both coverage measures. This is to be expected since the symbolic reasoning covers infinitely more cases than the explicit execution.

*Symbolic execution with subsumption checking is, as anticipated, the most effective exhaustive technique. For the* `TreeMap` *example it achieves good predicate coverage, even considering the lossy techniques.*

### 5.3.3 Abstract matching

State matching based on the shape abstraction is a lossy technique that performs the best of all the techniques: not only does it obtain the highest coverage (joint with others in some cases), but it obtains it for shorter sequences and it is faster. Only random selection, that essentially has no memory footprint, uses less memory when coverage is the same. We conjecture that for the analyzed containers, the shape is a very accurate representation of its state and hence the shape abstraction is appropriate here. It is an open question whether this will hold for general programs – it is likely to be the case for programs that manipulate complex data.

The complete abstraction that takes the shape and all fields into account performs almost as well, but uses more time and memory. This technique also performs consistently better than "classic" model checking which is closely related, but takes more than just the state of the container in consideration for state matching. Note that the complete abstraction also includes what is called a (data) symmetry reduction in model checking [18], and points to the fact that this kind of reduction is very useful in analyzing containers through API calls as we do here.

*Shape abstraction as a means for state matching performs better than the other techniques considered. We conjecture that for the analyzed containers, the shape is a very accurate representation of its state. When doing test input generation for programs that manipulate complex data, this should be tried before other, more expensive, techniques.*

### 5.3.4 Random selection

This is a traditional approach to test case generation; it is not based on state matching, hence it is the dual of the other methods suggested here and forms a useful point of comparison. Interestingly, in the related literature, it is almost never included in this kind of comparison. Here random search got

**Table 3: Exhaustive Techniques – Predicate Coverage**

| Container | Technique | Coverage | Seq. Length | Time (s) | Memory (MB) | Avg. Length |
|---|---|---|---|---|---|---|
| BinTree | Model Checking | 54 | **6** | 81 | 251 | 3.5 |
| | Complete Abstraction | 54 | 6 | 14 | 84 | 3.5 |
| | Symbolic Subsumption | 54 | 6 | 19 | 39 | 3.5 |
| BinomialHeap | Model Checking | 34 | **5** | 43 | 130 | 3.4 |
| | Complete Abstraction | 39 | **6** | 93 | 365 | 3.8 |
| | Symbolic Subsumption | 84 | **14** | 954 | 1016 | 6.8 |
| FibHeap | Model Checking | 31 | **5** | 59 | 208 | 4.0 |
| | Complete Abstraction | 89 | **11** | 733 | 1016 | 6.8 |
| | Symbolic Subsumption | 76 | **9** | 187 | 582 | 6.1 |
| TreeMap | Model Checking | 55 | **6** | 38 | 229 | 4.5 |
| | Complete Abstraction | 95 | **10** | 271 | 844 | 5.8 |
| | Symbolic Subsumption | 104 | **12** | 594 | 896 | 6.3 |

**Table 4: Lossy Techniques – Predicate Coverage**

| Container | Technique | Coverage | Seq. Length | Time (s) | Memory (MB) | Avg. Length |
|---|---|---|---|---|---|---|
| BinTree | Shape Abstraction | 54 | 9 | 4 | 11 | 3.6 |
| | Symbolic Shape Abstraction | 54 | 6 | 21 | 35 | 3.5 |
| | Random Selection | 54 | 8 | 15 | 4 | 6.2 |
| BinomialHeap | Shape Abstraction | 101 | 29 | 42 | 26 | 9.0 |
| | Symbolic Shape Abstraction | 84 | 14 | 1050 | 1016 | 6.8 |
| | Random Selection | 94 | 48 | 85 | 13 | 32.8 |
| FibHeap | Shape Abstraction | 93 | 15 | 243 | 292 | 7.1 |
| | Symbolic Shape Abstraction | 92 | 13 | 539 | 1016 | 6.9 |
| | Random Selection | 90 | 39 | 65 | 12 | 20.2 |
| TreeMap | Shape Abstraction | 106 | 20 | 281 | 1016 | 7.2 |
| | Symbolic Shape Abstraction | 102 | 13 | 1309 | 1016 | 6.2 |
| | Random Selection | 106 | 39 | 78 | 17 | 25.5 |

the optimal basic block coverage, but as expected for longer sequence lengths than the other techniques. In two cases, it also got the optimal predicate coverage, but for the other two it got considerably less than optimal. Again it is interesting to note that the two it fared worse in are the two examples with more complex environments (`BinomialHeap` and `FibHeap`). This supports the belief that random selection suffers when the environment is not only large, but only a (small) subset of the options will produce the desired result: note that the environment for basic block coverage is the same as for predicate coverage, but predicate coverage requires very particular sequences to obtain high coverage.

It is probably human nature to want the simplest solution also to perform the best. With this in mind we also wanted to improve on the results obtained for random selection. It was conjectured that picking `M = N` might adversely affect random search since the number of choices increase at higher values of `M` and `N` although the space of useful values (i.e. the ones that can obtain the optimal coverage) increase much slower. Therefore we repeated the experiments for predicate coverage with `M` being fixed at the value we found the coverage from Table 4, but with `N` being brought down from `M` to the smallest value we knew could still get the coverage according to the other results obtained. For example for `TreeMap` this meant running with `M = 39` and `N` varying from 39 down to 20 (a conservative lower-bound seen for the shape abstraction analysis).

Note that these results are somewhat biased towards getting good results for random selection. Ironically, the results indicate that random selection is less effective, when we add one more dimension to the quality criteria, namely, frequency of highest coverage obtained. Originally we measured the coverage once during any run of the random selection, and that is what is reported in Table 4. However doing all the additional experiments, and taking into account only runs that *could* obtain the optimal coverage we found that the percentage chance of getting the optimal coverage for random (note, not necessarily the optimal coverage for all techniques) were as follows: 1.5%(6/400) for `TreeMap`, 0.38%(2/520) for `FibHeap` and 0.17%(1/600) for `BinomialHeap`. Note that again the reduction in chance of finding optimal coverage follows the order of the complexity of the environments, where `TreeMap`'s environment is simpler than `FibHeap`'s environment which in turn is simpler than `BinomialHeap`'s environment.

*Considering the likelihood of obtaining the optimal results, random search performed poorly for obtaining high predicate coverage. We believe that the reason is that the input search space is complex/large and only a selected subset gives optimal results; in these situations the techniques based on shape abstractions yielded superior results.*

### 5.3.5 Complex data structures as input parameters

Here we did not consider API calls that take structured data as input. In prior work [30] we analyzed `TreeMap` with structured inputs with a black-box test input generation technique similar to Korat [7] and a white-box symbolic execution technique based on lazy initialization. These techniques would be applicable in the current evaluation too. However, we believe that it is unlikely that they would scale

to large enough structures to get the optimal coverage obtained here. Furthermore, we believe that random search will also not work well for complex data, since the domain of possible structures will be very large, whereas the subset of these that are valid structures will be small. However, we need to do more experiments to validate these claims.

#### 5.3.6 Challenge

In addition to the techniques and results reported here we also performed some experiments using the heuristic search facilities in JPF [13]. For the most part, the results were similar to what was reported here. However, using heuristic search for `FibHeap` with symbolic shape abstraction produced predicate coverage of 141 for sequence length 23 (better coverage than any other result here). This result indicates that there are more interesting test input generation techniques that need to be explored. In light of this we will make all our sources available on the JPF open-source website [19] for others to try additional techniques.

## 6. RELATED WORK

The work related to the topic of this paper is vast, and for brevity we only highlight here some of the closely related work. The most closely related works to ours are tools (and techniques) that generate test sequences for object oriented programs. We summarize them first.

JTest [20] is a commercial tool that generates test sequences for Java classes using "dynamic" symbolic execution, which combines concrete and symbolic execution over randomly generated paths. Unlike our work, this tool generates tests that may be redundant (exercise the same code), with little guarantees in terms of testing coverage. However, as we have seen in our experiments, random selection turns out to be pretty effective.

The AsmLT model-based testing tool [12] uses concrete state space exploration techniques and abstraction mappings, in a way similar to what we present here. Rostra [32] also generates unit tests for Java classes, using bounded-exhaustive exploration of sequences with concrete arguments and abstraction mappings. While both these tools require the user to provide the abstraction mappings, we provide automated support for shape abstractions that we have found very useful (see the experiments).

In previous work [30], we showed how to use model checking and symbolic execution to generate test inputs to achieve structural coverage for code that manipulates complex data structures, such as `TreeMap`. The approach was used in a black-box fashion (but it required an input specification written as a Java predicate – similarly to the Korat [7] tool) or in a white-box fashion (in which case only the source code for the method under test was needed). However, this approach was not used to generate method sequences (as we do here), but rather to build complex input structures that exercise the analyzed code in the desired fashion. Similarly, [4] discusses techniques to build complex input for testing red black tree implementation. On the other hand, Symstra [33] is a test generation tool that uses symbolic execution and state matching for generating test sequences for Java code – this is similar to our technique that uses symbolic execution and subsumption checking. Our paper contributes a novel combination of symbolic execution with abstraction, evaluation on Java container classes in terms of predicate coverage and comparison with random testing.

In a short paper [29], we discuss the use of explicit state model checking and abstractions for generating input test sequences for red black trees. In this paper we extend that work in several ways: we discuss the use of abstraction in the context of symbolic execution for test input generation and we provide an extensive evaluation using several Java container implementations (in addition to red black trees).

The Korat [7] tool, see also TestEra [23], supports non-isomorphic generation of complex input structures. Unlike the work presented here, this tool requires the availability of constraints representing these inputs. Korat uses constraints given as Java predicates (e.g. `repOK` methods encoding class invariants). Similarly, TestEra [23] uses constraints given in Alloy to generate complex structures.

The ASTOOT tool [9] requires algebraic specifications to generate tests (including oracles) for object oriented programs. The tool generates sequences of interface events and checks whether the resulting objects are observationally equivalent (as specified by the specification). Although here we were only interested in generating test sequences, using an algebraic specification to check the functional requirements of the code is a straightforward extension.

A set of techniques, not investigated in this paper, use optimization based techniques (e.g. genetic algorithms) for automated test case generation [27, 5]. In the future, we plan to compare these optimization based techniques with the state matching based techniques that are implemented in our framework.

The work presented here is related to the use of model checking for test input generation [1, 10, 15, 17]. In these works, one specifies as a (temporal) property that a specific coverage cannot be achieved and a model checker is used to produce counterexample traces, if they exists, that then can be transformed into test inputs to achieve the stated coverage. Our work shows how to enable an off-the-shelf model checker to generate test sequences for complex data structures. Note that our techniques can be implemented in a straightforward fashion in other software model checkers (e.g. [11, 8]).

Recently two popular software model checkers, BLAST and SLAM, have been used for generating test inputs with the goal of covering a specific predicate or a combination of predicates [6, 3]. Both these tools use *over-approximation* based predicate abstraction and use some form of symbolic evaluation for the analysis of (spurious) abstract counterexamples and refinement. We use *under-approximation* based abstraction – hence no spurious behavior is explored. The work in [3] describes predicate coverage as a new testing metric – we use a simplified version here; [3] also describes ways to measure when the optimal predicate coverage has been achieved (this a direction for future work that we would like to pursue).

The idea of using abstractions during model checking has been explored before. In [16], the abstractions need to be provided manually, while [24] uses automatic predicate abstraction for state matching during the the explicit execution of concurrent systems. In [2] we present a method for checking subsumption during symbolic execution; furthermore, shape abstractions are used to prune the search state space. Subsumption checking in [2] is more general than here (it handles partially initialized heap structures and summary nodes). However the abstractions in [2] can only handle lists and arrays, and not tree structures as we do here.

There is a lot of work on comparing random with partition testing in terms of cost effectiveness, e.g. [31, 14]. The verdict is still uncertain: [31] seems to suggest that random testing could be superior to partition testing under certain assumptions, while [14] suggests that, under different assumptions, partition testing is superior. Although fairly preliminary, we hope that our experiments will shed some light on this controversy, in the context of testing of data structures. Abstract matching can be seen as a form of partition testing (the state space explored by the model checker is partitioned according to the abstraction mapping) and seems superior to the other techniques.

## 7. CONCLUSIONS

We presented test input generation techniques that use state matching to avoid generation of redundant tests. The techniques range from exhaustive techniques such as classic model checking and symbolic execution with subsumption checking, through lossy abstraction techniques that use the shape of a container for state matching. We evaluated the techniques in terms of testing coverage achieved by the generated tests and we also compared them to random selection.

For the simple basic block coverage the exhaustive techniques are comparable to the lossy ones while for predicate coverage (which is more difficult to achieve) the lossy techniques fared better at obtaining high coverage. Random selection performed well except on the examples that contained complex input spaces, where the shape abstractions performed better. However, one should not lose sight of the strong guarantees that an exhaustive search, such as symbolic execution with subsumption, can provide: up to the maximum sequence length that allows exhaustive analysis, one can show that the implementation is free of errors.

For the future, we plan to investigate whether the shape abstraction that proved to be effective here, will also work for generating tests for other (more general) Java programs. We also plan to investigate other abstractions for our framework, e.g. abstractions used in shape analysis [26], and we plan to extend our evaluation to Java methods that take complex data structures and arrays as inputs.

Another direction for future research is to investigate the use of predicate abstraction for the automatic generation of different abstraction mappings. Towards this end, we plan to extend our work on automatic derivation of under approximation based abstractions [24], where we used a (backward) weakest precondition based calculation for automatic abstraction refinement. In the current setting we plan to adapt this algorithm to use (forward) symbolic execution.

## 8. REFERENCES


[1] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. 2nd IEEE International Conference on Formal Engineering Methods*, 1998.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *Proc. 13th International SPIN Workshop*, 2006.

[3] T. Ball. A theory of predicate-complete test coverage and generation, 2004. Microsoft Research Technical Report MSR-TR-2004-28.

[4] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. J. White. State generation and automated class testing. *Softw. Test., Verif. Reliab.*, 10(3):149–170, 2000.

[5] A. Baresel, M. Harman, D. Binkley, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[6] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. 26th International Conference on Software Engineering (ICSE)*, 2004.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[8] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[9] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):101–130, 1994.

[10] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 1999.

[11] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, Paris, France, Jan. 1997.

[12] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 112–122, July 2002.

[13] A. Groce and W. Visser. Heuristics for model checking Java programs. *STTT Journal*, 6(4), December 2004.

[14] W. J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, 1999.

[15] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, D. George, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, Canada, Oct. 2003.

[16] G. J. Holzmann and R. Joshi. Model-driven software verification. In *Proc. 11th International SPIN Workshop*, 2004.

[17] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proc. 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Grenoble, France, April 2002.

[18] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, Nov. 2001.

[19] Java PathFinder. `http://javapathfinder.sourceforge.net`.

[20] JTest. `http://www.parasoft.com/jsp/home.jsp`.

[21] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[22] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[23] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.

[24] C. S. Păsăreanu, R. Pelanek, and W. Visser. Concrete model checking with abstract matching. In *Proc. 17th International Conference on Computer Aided Verification (CAV)*, 2005.

[25] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 31(8), Aug. 1992.

[26] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, January 1998.

[27] P. Tonella. Evolutionary testing of classes. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 2004.

[28] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[29] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for red black trees using abstraction (short presentation). In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[30] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation in Java Pathfinder. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[31] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[32] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE)*, 2004.

[33] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2005.

# Chapter 9

# Model Classifications and Automated Verification

In this paper we discuss the issue of automating the verification process, formulate the verification meta-search problem, and propose the concept of a verification manager. We also discuss general ideas for the realization of the verification manager.

On a specific level the paper is concerned with development of model classifications. Proposed classifications are based both on the syntax of the model (e.g., communication mode, process similarity, application domain) and on properties of state space (e.g., structure of SCC components, shape of the state space, local structure). Classifications were derived from experimental study of models in the BEEM set.

This paper was published in proceedings of Formal Methods for Industrial Critical Systems (FMICS) in 2007:

- R. Pelánek. Fighting state space explosion: Review and evaluation. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'08)*, 2008.

# Model Classifications and Automated Verification

Radek Pelánek*

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

**Abstract.** Due to the significant progress in automated verification, there are often several techniques for a particular verification problem. In many circumstances different techniques are complementary — each technique works well for different type of input instances. Unfortunately, it is not clear how to choose an appropriate technique for a specific instance of a problem. In this work we argue that this problem, selection of a technique and tuning its parameter values, should be considered as a standalone problem (a verification meta-search). We propose several classifications of models of asynchronous system and discuss applications of these classifications in the context of explicit finite state model checking.

## 1 Introduction

One of the main goals of computer aided formal methods is automated verification of computer systems. In recent years, very good progress has been achieved in automating specific verification problems. However, even automated verification techniques like model checking are far from being a push-button technology. With current verification techniques many realistic systems can be automatically verified, but only if applied to the right level of abstraction of a system and if suitable verification techniques are used and right parameter values are selected.

The first problem is addressed by automated abstraction refinement techniques and received lot of attention recently (e.g., [1,9]). The second problem, however, did not receive much attention so far and there are only few works in this direction. Ruys and Brinksma [38] describe methodology for model checking 'in the large'. Sahoo et al. [39] use sampling of the state space to decide which BDD based reachability technique is the best for a given model. Mony et al. [29] use expert system for automating proof strategies. Eytani et al. [11] give a high-level proposal to use an 'observation database' for sharing relevant information among different verification techniques.

Automation of the verification process is necessary for practical applicability of formal verification. Any self-respecting verification tool has a large number of options and parameters, which can significantly influence the complexity of verification. In order to verify any reasonable system, it is necessary to set these

* Partially supported by GA ČR grant no. 201/07/P035.

S. Leue and P. Merino (Eds.): FMICS 2007, LNCS 4916, pp. 149–163, 2008.

parameters properly. This can be done only by an expert user and it requires lot of time. We believe that the research focus should not be only on the development of new automated techniques, but also on an automated selection of an existing technique.

### 1.1 Verification Meta-search

So far most of the research in automated verification has been focused on questions of the verification problem: given a system $S$ and a property (or specification) $\varphi$, determine whether $S$ satisfies $\varphi$. This is a *search* problem — an algorithm searches for an incorrect behaviour or for a proof. Research has been focused on solving the problem for different formalisms and optimizing it for the most useful ones.

We believe that it is worthwhile to consider the following problem as well: given a system and a property, find a technique $T$ and parameter values $p$ such that $T(p)$ can provide answer to the verification problem. This can be viewed as a *verification meta-search problem*. Let an entity responsible for the verification meta-search be called a *verification manager*. The manager has the following tasks:

1. Decide which approach to the verification should be used, e.g., symbolic versus explicit approach, whether to use on-the-fly verification or whether to generate the full state space and then perform verification, etc.
2. Combine relevant information obtained from different techniques, see e.g., Synergy approach [18] for combination of over-approximation and testing.
3. Choose among different techniques (implementations) for a particular verification task and set parameters of a chosen technique.

In this work we focus mainly on the third task of the manager. To give a practical example of this task, we provide two specific cases. Firstly, consider on-the-fly memory reduction techniques — the goal of these techniques is to reduce memory requirements of exhaustive finite state verification. Examples of such techniques are partial order reduction, symmetry reduction, state compression, and caching. Each of these techniques has its merits and disadvantages, none of them is universal (see [32] for an evaluation). Moreover, most of these techniques have parameters which can tune a time/memory trade-off. Secondly, consider algorithms for accepting cycle detection on networks of workstations, which are used for LTL verification of large finite state models. Currently there are at least five different algorithms, each with specific disadvantages and parameters [2].

At the moment the verification manager is usually a human expert. Expert can perform this role rather well, however such 'implementation' of the verification manager is far from automated. There has been attempts to facilitate the human involvement, e.g., by using special purpose scripting languages [25], but such an approach automatizes only stereotypical steps during the verification, not decisions.

The problem can be addressed by an expert system, which perform the meta-search with the use of a set of rules provided by experts. Example of such rules may be:

- If the model is a mutual exclusion protocol then use explicit model checking with partial order reduction.
- If state vector is longer then 30 bytes, then use state compression.
- If the state space is expected to contain a large strongly connected component, then use cycle detection algorithm $X$ else use cycle detection algorithm $Y$.

Another option is to employ an adaptive learning system which remembers characteristics of verification tasks and their results and learns from its own experiences.

### 1.2 The Need for Classifications

At this moment, it is not clear what rules should the verification manager use. But more fundamentally, it is not even clear what criteria should be used in rules. Whatever is the realization of the manager, the manager needs to make decision based on some information about an input model. The information used for this decision should be carefully chosen:

- If the information was too coarse, the manager would not be able to choose among potentially suitable techniques.
- If the information was too detailed, it would be very hard for the manager to apply its expertise and experiences.

We believe that an appropriate approach is to develop several categorical classifications of models and then use these classification for manager's decisions. To be applicable, it must be possible to determine suitable techniques for individual classes of the classification. Moreover, it must be possible to determine a class of a given model without much effort — either automatically by a fast algorithm or easily by user judgement.

In this work we focus on asynchronous concurrent systems, for the evaluation we use models from the BEEM set [35]. For asynchronous concurrent systems one of the most suitable verification techniques is explicit state space exploration. Therefore, we focus not only on analysis of a model structure, but also on the analysis of state spaces. In this work we propose classifications based on a model structure (communication mode, process similarity, application domain) and also classifications based on properties of state spaces (structure of strongly connected components, shape and local structure of a state space). We study relation of these classifications and discuss how they can be useful for the selection of suitable techniques and parameters (i.e., for guiding the meta-search).

The restriction to explicit model checking techniques limits the applicability of our contribution. Note, however, that even for this restricted area, there is a very large number of techniques and optimizations (there are at least 80 research papers dealing with explicit model checking techniques, see [34] for a list). Moreover, the goal of this work is not to present the ultimate model classification, but rather to pinpoint a direction, which can be fruitful.

## 2 Background

**Used models.** For the evaluation of properties of practically used models we employ models from the benchmark set BEEM [35]. This set contains large number of models of asynchronous systems. The set contains classical models studied in academic literature as well as realistic case studies. Models are provided in a low-level specification language (communicating finite state machines) and in Promela [20]. For our study we have used 115 instances obtained by instantiation of 57 principally different models.

**State spaces.** For each instance we have generated its state space. We 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 $v_0$. Vertices are states of the model, edges represent valid transitions between states. For our purposes we ignore any labeling of states or edges. We are concerned only with the reachable part of the state space.

Let us define several parameters of state spaces. We use these parameters for classifications. We have also studied other parameters (particularly those reported in [31]), but these parameters do not lead to interesting classification.

An *average degree* of $G$ is the ratio $|E|/|V|$. 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. Let us consider the breadth-first search (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, *BFS width* is the maximal size of a BFS 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.

**Reduction techniques.** In the following, we often mention two semantics based reduction techniques. Under the notion *partial order reduction* (POR) we consider all techniques which aim at reducing the number of explored states by reducing the amount of interleaving in the model, i.e., we denote by this notion not just the classic partial order reduction technique [16], but also other related techniques, e.g., confluence reduction [5], simultaneous reachability analysis [30], transition compression [24]. *Symmetry reduction* techniques aim at reducing the number of explored states by considering symmetric states as equivalent, see e.g. [22].

## 3 State Space Classifications

We consider three classifications based on properties of state spaces. Two of them are based on "global" properties (structure of SCC and shape), one is based on "local" features of state spaces.

### 3.1 Structure of SCC Components

There is an interesting dichotomy with respect to structure of strongly connected components, particularly concerning the size of the largest SCC (see Fig. 1). A state space either contains one large SCC, which includes nearly all states, or there are only small SCCs. Based on this observation, we propose the following classification:

**A type** (acyclic): a state space is acyclic, i.e., it contains only trivial components with one state,

**S type** (small components): a state space is not acyclic, but contains only small components; more precisely we consider a state space to be of this type if the size of the largest component is smaller then 50% states.

**B type** (big component): a state space contain one large component, most states are in this component.

In order to apply the classification for automated verification, we need to be able to detect the class of a model without searching its full state space. This can be done by random walk exploration [36], for example by the following simple method based on detection of cycles by random walk. We run 100 independent random walks through the state space. Each random walk starts at the initial state and is limited to at most 500 steps. During the walk we store visited states, i.e., path through the state space. If a state is revisited then a cycle is detected and its length can be easily computed. At the end, we return the length of the longest detected cycle. Fig. 1 shows results of this method. For the class **A** the longest detected cycle is, of course, always 0. For the class **S** the longest detected cycle is usually between 10 and 35, for the class **B** it is usually above 30. This illustrates that even such a simple method can be used to quickly classify state spaces with a reasonable precision.

What are possible applications of this classification? For the **A** type it is possible to use specialized algorithms, e.g., dynamic partial order reduction [12] or bisimulation based reduction [33, p. 43-47]. The sweep line method [8] deletes from memory states, which will never be visited again. This method is useful only for models with state spaces of the type **A** or **S**.

The performance of cycle detection algorithms[1], which are used for LTL verification, is often dependent on the SCC structure. For example a distributed algorithm based on localization of cycles is suitable only for **S** type state spaces; depth-first search based algorithm [21] can also be reasonably applied only for **S** type state spaces, because for **B** type state spaces it tends to produce very long counterexamples, which are not practical. On the other hand, (explicit) one-way-catch-them-young algorithm [6] has complexity $O(nh)$, where $h$ is height of the

[1] Note that cycle detection algorithm are usually executed on the product graph with a formula [42] and not on the state space itself. However, our measurements indicate that the structure of product graphs is very similar to structure of plain state spaces. The measurements were performed on product graphs included in the BEEM [35] set.

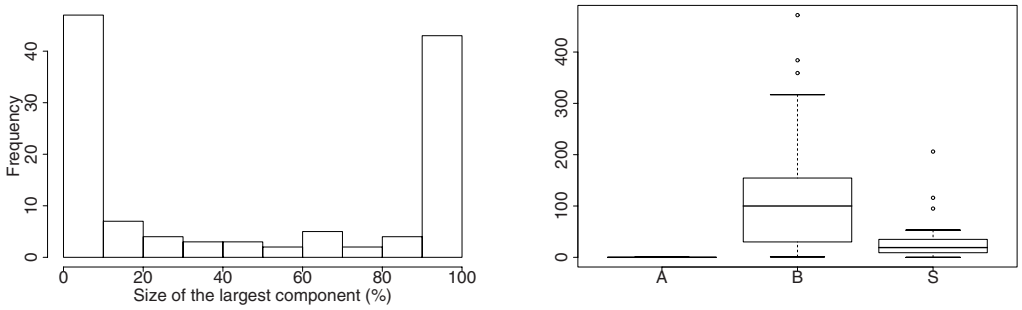



**Fig. 1.** The first graph shows the histogram of sizes of the largest SCC component in a state space. The second graph shows the longest detected cycle using random walk; results are grouped according to class and presented using a boxplot method (lines denote minimum, 25th quartile, median, 75th quartile and maximum, circles are outliers).

SCC quotient graph, i.e., this algorithm is more suitable for **B** type state spaces. Similarly, the classification can be employed for verification of branching time logics (e.g., the algorithm in [7] does not work well for state spaces consisting of one SCC).

### 3.2 Shape of the State Space

We have found that several global state space parameters are to certain extent related: average degree, BFS height and width, number and length of back level edges. In this case the division into classes is not so clear as in the previous case. Nevertheless, it is possible to identify two main classes with respect to these parameters[2]:

**H type** (high): small average degree, large BFS height, small BFS width, few long back level edges.

**W type** (wide): large average degree, small BFS height, large BFS width, many short back level edges.

This classification can be approximated using an initial sample of the BFS search. The classification can be used in similar way as the previous one. Sweep line [8] and caching based on transition locality [37] work well only for state spaces with short back level edges, i.e., these techniques are suitable only for **W** type state spaces. On the other hand, the complexity of BFS-based distributed cycle detection algorithm [3] is proportional to number of back level edges, i.e., this algorithm works well only on **H** type state spaces.

For many techniques the **H**/**W** classification can be used to set parameters appropriately: algorithms which exploit magnetic disk often work with individual BFS levels [41]; random walk search [36] and bounded search [23] need to

[2] Note that this classification is not complete partition of all possible state spaces. The remaining classes, however, do not occur in practice. The same holds for as several other classifications which we introduce later.

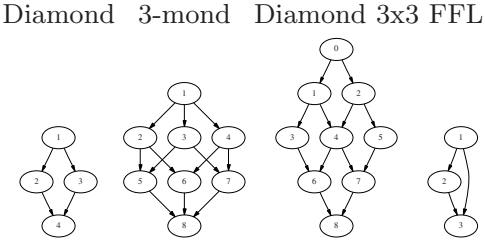



**Fig. 2.** Illustrations of motifs

estimate the height of the state space; techniques using stratified caching [15] and selective storing of states [4] could also take the shape of the state space into account.

### 3.3 Local Structure

Now we turn to a local structure of state spaces, particularly to typical subgraphs. Recently, so called 'network motifs' [28,27] were intensively studied in complex networks. 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 have found the following motifs to be of specific interest either for abundant presence or for total absence in many state spaces: *diamonds* (we have studied several variations of diamond-like structures, see Fig. 2), which are well known to be present in state spaces of asynchronous concurrent systems due to the interleaving semantics; *chains* of states with just one successor, we have measured occurrences of chains of length 3, 4, 5; *short cycles* of lengths 2, 3, 4, 5, which are not very common in most state spaces; and *feed forward loop* (see Fig. 2), which is a typical motif for networks derived from biological systems [28], in state spaces it is rather rare.

We have measured number of occurrences of these motifs and studied correlations of their occurrences. With respect to motifs we propose the following classes:

**D type** (diamond)**:** a state space contains many diamonds, usually no short cycles and only few chains of feed forward loops,
**C type** (chain)**:** a state space contains many chains, very few diamonds or short cycles,
**O type** (other)**:** a state space either contains short cycles and/or feed forward loops, chains are nearly absent, diamonds may be present, but they are not dominant.

Identification of these classes can be performed by exploration of a small sample of the state space. This classification can be used to choose among memory reduction techniques. For **D** type state spaces it is reasonable to try to employ POR, whereas for **C** type state spaces this reduction is unlikely to yield significant improvement. On the other hand, for **C** type state spaces good memory

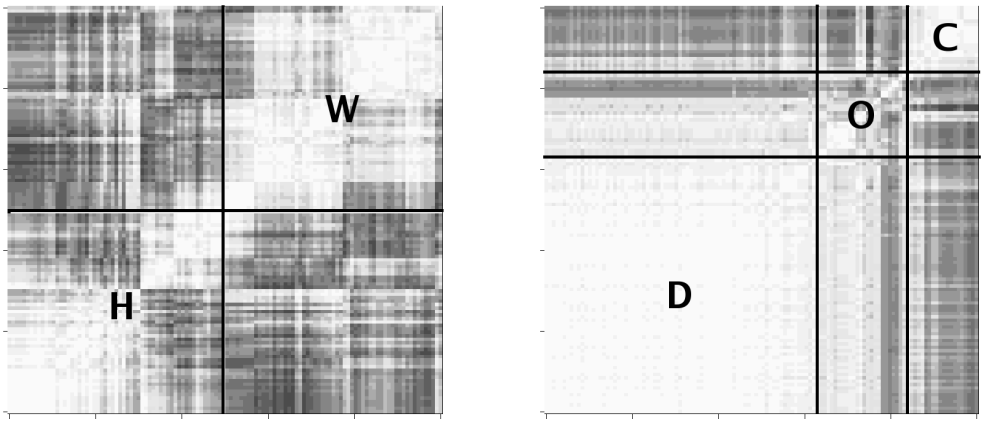



**Fig. 3.** Correlation matrix displaying correlation of 116 state spaces. Light color means positive correlation, dark color means negative correlation. The first matrix shows correlation with respect to average degree, BFS height and width, number and length of back level edges (all parameters are normalized). The second matrix shows correlations with respect to presence of studied motifs.

reduction can be obtained by selective storing of states [4]. The classification can be also used for tuning parameter values, particularly for technique which employ local search, e.g., random walk enhancements [36,40], sibling caching and children lookahead in distributed computation [26], heuristic search.

### 3.4 Relation Among State Space Classifications

Table 1. presents number of models in different combinations of classes. Specific numbers presented in the table are influenced by the selection of used models. Nevertheless, it is clear that presented classifications are rather orthogonal, there is just slight relation between the shape and the local structure.

## 4 Model Classifications

Now we turn to classifications based directly on a model. At first, we study classifications according to model structure, which are relevant particularly with respect to reduction techniques based on semantics (e.g., partial order reduction, symmetry reduction). Secondly, we study models from different application domains and show that each application domain has its characteristics with respect to presented classifications.

### 4.1 Model Structure

Classifications based on structure of a model are to some extent dependent on a specific syntax of the specification language. There are many specification languages and individual specification languages significantly differ on syntactical level. However, if we restrict our attention to models of asynchronous systems, we find that most specification languages share the following features:

- a model is comprised of a set of processes,
- a process can be viewed as a finite state machine extended with variables,
- processes communicate either via channels or via globally shared variables.

We discuss several possible classifications based on these basic features. Categorization of a model according to these classifications can be determined automatically by static analysis of a model. This issue is dependent on a particular specification language and it is rather straightforward, therefore, we do not discuss it in detail.

**Communication Mode.** With respect to communication we can study the predominant mean of communication (shared variables or channels) and the communication structure (ring, line, clique, star). It turns out that these two features are coupled, i.e., with respect to communication we can consider the following main classes:

**DV type** (dense, variable)**:** processes communicate via shared variables, the communication structure is dense, i.e., every process can communicate with (nearly) every other process,

**SC type** (sparse, channel)**:** processes communicate via (buffered) channels, the communication structure is rather sparse, e.g., ring, star, or tree.

**N type** (none)**:** no communication, i.e., the model is comprised of just one process.

This classification is related particularly to partial order reduction techniques. The classification is completely orthogonal to state space classifications (see Table 1.).

**Process Similarity.** A common feature in models of asynchronous systems is the occurrence of several similar processes (e.g., several participants in a mutual exclusion protocol, several users of an elevator, several identical nodes in a communication protocol). By 'similarity' we mean that processes are generated from one template by different instantiations of some parameters, i.e., we do not consider symmetry in any formal sense (cf. [22]). With respect to similarity, a reasonable classification is the following:

**S2 type.** All processes are similar.

**S1 type.** There exists some similar processes, but not all of them.

**S0 type.** There is no similarity among processes.

This classification is clearly related to symmetry reduction. It can also be employed for state compression [19]. This classification is again orthogonal to state space classifications and only slightly correlated with the communication mode classification (**S1** is related to **SC**, **S2** is related to **DV**), for details see Table 1.

**Table 1.** Relations among classifications. For each combination of classes we state the number of models in the combination. In total there are 115 classified models, all models are from the BEEM set [35]. Reported state space classifications are based on traversal of the full state space.

State space classifications

| | all | **H** | **W** | **D** | **O** | **C** |
|---|---|---|---|---|---|---|
| all | 115 | 58 | 57 | 75 | 23 | 17 |
| **A** | 24 | 10 | 14 | 19 | 3 | 2 |
| **S** | 37 | 18 | 19 | 21 | 9 | 7 |
| **B** | 54 | 30 | 24 | 35 | 11 | 8 |
| **H** | 58 | | | 43 | 3 | 12 |
| **W** | 57 | | | 32 | 20 | 5 |

Model classifications

| | all | **S2** | **S1** | **S0** |
|---|---|---|---|---|
| all | 115 | 41 | 39 | 35 |
| **DV** | 44 | 31 | 9 | 4 |
| **SC** | 61 | 10 | 30 | 21 |
| **N** | 10 | 0 | 0 | 10 |

State space versus model classifications

| | all | **A** | **S** | **B** | **H** | **W** | **D** | **O** | **C** |
|---|---|---|---|---|---|---|---|---|---|
| all | 115 | 24 | 37 | 54 | 58 | 57 | 75 | 23 | 17 |
| **S2** | 41 | 10 | 9 | 22 | 20 | 21 | 37 | 3 | 1 |
| **S1** | 39 | 8 | 17 | 14 | 20 | 19 | 20 | 10 | 9 |
| **S0** | 35 | 6 | 11 | 18 | 18 | 17 | 18 | 10 | 7 |
| **DV** | 44 | 8 | 12 | 24 | 20 | 24 | 33 | 6 | 5 |
| **SC** | 61 | 12 | 22 | 27 | 36 | 25 | 40 | 10 | 11 |
| **N** | 10 | 4 | 3 | 3 | 2 | 8 | 2 | 7 | 1 |

**Other.** We briefly mention several other possible classifications and their applications:

- Data/Control intensity of a model (Is a model concerned with data manipulation and arithmetic?); related to abstraction techniques [17,1,9], which focus on reducing the data part of the model.
- Tightly/Loosely coupled processes (What is the proportion of interprocess and intraprocess computation?); important for thread-modular techniques [13].
- Length of a state vector; relevant particularly for state compression techniques [19].

### 4.2 Application Domain

Finally, we discuss application domains of asynchronous concurrent systems. Table 2. presents relations among application domains and previously discussed classifications. The table demonstrates that models from each application domain have specific characteristics. Knowledge of these characteristics can be helpful for the development of (commercial) verification tools specialized for a particular application domain. Beside that, characteristics of models can be used to develop templates and design patterns [14,10], which can facilitate the modeling process.

**Mutual exclusion algorithms.** The goal of a mutual exclusion algorithm is to ensure an exclusive access to a shared resource. Models of these algorithms usually consist of several nearly identical processes which communicate via shared

**Table 2.** Relation of state space classification and model type

| | | SCC struct. | | | shape | | local struct | | | comm. | | | proc. sim. | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | all | **A** | **S** | **B** | **H** | **W** | **C** | **D** | **O** | **SC** | **DV** | **N** | **S2** | **S1** | **S0** |
| all | 115 | 24 | 37 | 57 | 58 | 57 | 17 | 75 | 23 | 61 | 44 | 10 | 41 | 39 | 35 |
| com. protocol | 24 | 0 | 10 | 14 | 15 | 9 | 5 | 18 | 1 | 24 | 0 | 0 | 0 | 7 | 17 |
| controller | 17 | 1 | 7 | 9 | 15 | 2 | 3 | 12 | 2 | 12 | 5 | 0 | 0 | 13 | 4 |
| leader el. | 12 | 12 | 0 | 0 | 6 | 6 | 0 | 12 | 0 | 8 | 4 | 0 | 9 | 3 | 0 |
| mutex | 28 | 0 | 8 | 20 | 13 | 15 | 0 | 25 | 3 | 2 | 26 | 0 | 28 | 0 | 0 |
| sched. | 18 | 9 | 3 | 6 | 4 | 14 | 2 | 5 | 11 | 2 | 6 | 10 | 1 | 5 | 12 |
| other | 16 | 2 | 9 | 5 | 5 | 11 | 7 | 3 | 6 | 13 | 3 | 0 | 3 | 11 | 2 |

variables; communication structure is either clique or ring; individual processes are usually rather simple. State vectors are relatively short; state space usually contains one big strongly connected component, with many diamonds. POR and symmetry reduction may be useful, but careful modeling may be necessary in order to make them applicable.

**Communication protocols.** The goal of communication protocols is to ensure communication over an unreliable medium. The core of a model is a sender process, a receiver process, and a bus/medium; the communication structure is therefore usually linear (or simple tree). Processes communicate by handshake; shared variables are not used. Processes are not similar, sender/receiver processes can be rather complicated. State vectors are rather long; state space is not acyclic, it is rather high, often with many diamonds. POR is usually applicable.

**Leader election algorithms.** The goal of leader election algorithms is to choose a unique leader from a set of nodes. Models consist of a set of (nearly) identical processes, which are rather simple. Processes are connected in a ring, tree, or arbitrary graph; communication is via (buffered) channels. State spaces are acyclic with diamonds. POR, symmetry reduction, and specialized techniques for acyclic state spaces [12] may be applicable.

**Controllers.** Models of controllers usually have centralized architecture: a controller process communicates with processes representing individual parts of the system. The controller process is rather complex, other processes may be simple. The communication can be both by shared variables and handshake. State vectors are rather long; state spaces are high, usually with diamonds. Due to the centralized architecture semantics-based reduction techniques are hard to apply.

**Scheduling, planning, puzzles.** Planning and scheduling problems and puzzles are not the main application domain of explicit model checkers. Nevertheless, there are good reasons to consider them together with asynchronous systems (similar modeling formalism, research in combinations of model checking and artificial intelligence techniques). Models often consist of just one process.

Planning, scheduling problems have wide state space without prevalence of diamonds or chains. State spaces are often acyclic.

**Other application domains.** Similar characterizations can be provided for many other application domains. Examples of other often studied application domains are cache coherence protocols, device drivers or data containers.

## 5 Conclusions and Future Work

We argue that it is important not just to develop (narrowly focused) techniques for automated verification, but also to automatize the verification *process*, which is currently usually performed by an expert user. To this end, it is desirable to have classifications of models. We propose such classifications for asynchronous systems; these classifications are based on properties of state spaces and on structure of models. We also discuss examples of applications of these classifications; particularly the following two types of application:

- indication of suitable techniques to use for verification of the given model,
- setting suitable parameter values for the verification.

In the paper we provide several specific examples of such application; we note that these are just examples, not a full list of possible applications. The presented classifications are also not meant to be the final classifications of asynchronous systems. We suppose that further research will expose the need for other classification or for the refinement of presented classification. Moreover, for other application domains and verification techniques (e.g., synchronous systems, symbolic techniques, bounded model checking) it will be probably necessary to develop completely new classifications. Nevertheless, we believe that our approach can provide valuable inspiration even for this direction.

This work is a part of a long term endeavour. We are continuously developing the benchmark set BEEM [35]. Using the presented classification, we are working on experimental evaluation of the relation of classes and performance of different techniques. We are also developing techniques for estimation of state space parameters from samples of a state spaces, such estimations (e.g., the size of a state space), can be useful for guiding the verification meta-search. Finally, the long term goal is to develop an automated 'verification manager', which would be able to learn from experience.

## References

1. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: Proc. of Programming Language Design and Implementation (PLDI 2001), pp. 203–213. ACM Press, New York (2001)
2. Barnat, J., Brim, L., Cerná, I.: Cluster-based ltl model checking of large systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 259–279. Springer, Heidelberg (2006)

3. Barnat, J., Brim, L., Chaloupka, J.: Parallel breadth-first search LTL model-checking. In: Proc. Automated Software Engineering (ASE 2003), pp. 106–115. IEEE Computer Society, Los Alamitos (2003)
4. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725. Springer, Heidelberg (2003)
5. Blom, S., van de Pol, J.: State space reduction by proving confluence. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 596–609. Springer, Heidelberg (2002)
6. Černá, I., Pelánek, R.: Distributed explicit fair cycle detection. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
7. Cheng, A., Christensen, S., Mortensen, K.: Model checking coloured petri nets exploiting strongly connected components. Technical Report DAIMI PB – 519, Computer Science Department, University of Aarhus (1997)
8. Christensen, S., Kristensen, L.M., Mailund, T.: A Sweep-Line Method for State Space Exploration. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
10. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proc. Workshop on Formal Methods in Software Practice, pp. 7–15. ACM Press, New York (1998)
11. Eytani, Y., Havelund, K., Stoller, S.D., Ur, S.: Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs. Concurrency and Computation: Practice and Experience 19(3), 267–279 (2007)
12. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proc. of Principles of programming languages (POPL 2005), pp. 110–121. ACM Press, New York (2005)
13. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1995)
15. Geldenhuys, J.: State caching reconsidered. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 23–39. Springer, Heidelberg (2004)
16. Godefroid, P.: Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
17. Graf, S., Saidi, H.: Construction of abstract state graphs with pvs. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
18. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: A new algorithm for property checking. In: Proc. of Foundations of software engineering, pp. 117–127. ACM Press, New York (2006)
19. Holzmann, G.J.: State compression in SPIN: Recursive indexing and compression training runs. In: Proc. of SPIN Workshop (1997)
20. Holzmann, G.J.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading, Massachusetts (2003)
21. Holzmann, G.J., Peled, D., Yannakakis, M.: On nested depth first search. In: Proc. SPIN Workshop, pp. 23–32. American Mathematical Society, Providence, RI (1996)

22. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design 9(1–2), 41–75 (1996)
23. Krčál, P.: Distributed explicit bounded ltl model checking. In: Proc. of Parallel and Distributed Methods in verifiCation (PDMC 2003). ENTCS, vol. 89. Elsevier, Amsterdam (2003)
24. Kurshan, R.P., Levin, V., Yenigün, H.: Compressing transitions for model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 569–581. Springer, Heidelberg (2002)
25. Lang, F.: Compositional verification using svl scripts. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 465–469. Springer, Heidelberg (2002)
26. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 80–102. Springer, Heidelberg (2001)
27. Milo, R., Itzkovitz, S., Kashtan, N., Levitt, R., Shen-Orr, S., Ayzenshtat, I., Sheffer, M., Alon, U.: Superfamilies of evolved and designed networks. Science 303(5663), 1538–1542 (2004)
28. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: Simple building blocks of complex networks. Science 298(5594), 824–827 (2002)
29. Mony, H., Baumgartner, J., Paruthi, V., Kanzelman, R., Kuehlmann, A.: Scalable automated verification via expert-system guided transformations. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 159–173. Springer, Heidelberg (2004)
30. Ozdemir, K., Ural, H.: Protocol validation by simultaneous reachability analysis. Computer Communications 20, 772–788 (1997)
31. Pelánek, R.: Typical structural properties of state spaces. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 5–22. Springer, Heidelberg (2004)
32. Pelánek, R.: Evaluation of on-the-fly state space reductions. In: Proc. of Mathematical and Engineering Methods in Computer Science (MEMICS 2005), pp. 121–127 (2005)
33. Pelánek, R.: Reduction and Abstraction Techniques for Model Checking. PhD thesis, Faculty of Informatics, Masaryk University, Brno (2006)
34. Pelánek, R.: Web portal for benchmarking explicit model checkers. Technical Report FIMU-RS-2006-03, Masaryk University Brno (2006), `http://anna.fi.muni.cz/models/`
35. Pelánek, R.: Beem: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
36. Pelánek, R., Hanžl, T., Černá, I., Brim, L.: Enhancing random walk state space exploration. In: Proc. of Formal Methods for Industrial Critical Systems (FMICS 2005), pp. 98–105. ACM Press, New York (2005)
37. Penna, G.D., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.V.: Exploiting transition locality in automatic verification of finite state concurrent systems. Software Tools for Technology Transfer (STTT) 6(4), 320–341 (2004)
38. Ruys, T.C., Brinksma, E.: Managing the verification trajectory. International Journal on Software Tools for Technology Transfer (STTT) 4(2), 246–259 (2003)
39. Sahoo, D., Jain, J., Iyer, S.K., Dill, D., Emerson, E.A.: Predictive reachability using a sample-based approach. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 388–392. Springer, Heidelberg (2005)

40. Sivaraj, H., Gopalakrishnan, G.: Random walk based heuristic algorithms for distributed memory model checking. In: Proc. of Parallel and Distributed Model Checking (PDMC 2003). ENTCS, vol. 89 (2003)
41. Stern, U., Dill, D.L.: Using magnetic disk instead of main memory in the Murphi verifier. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
42. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Kozen, D. (ed.) Proc. of Logic in Computer Science (LICS 1986), pp. 332–344. IEEE Computer Society Press, Los Alamitos (1986)

# Chapter 10

# Fighting State Space Explosion: Review and Evaluation

This work comprises an important piece in our long term effort. It summarises both the work in the area and our own experiences, and provides a basic argument for our approach to automating the verification process.

In this paper we provide a systematic overview of techniques for fighting state space explosion and we analyse trends in the relevant research. We also report on our own experience with practical performance of techniques – the report is a concise summary of several other papers and technical reports [57, 58, 59, 67, 68]. Main conclusion of the study is a recommendation for both practitioners and researchers: be critical to claims of dramatic improvement brought by a single sophisticated technique, rather use many different simple techniques and combine them.

This paper was published in proceedings of Formal Methods for Industrial Critical Systems (FMICS) in 2008:

- R. Pelánek. Model classifications and automated verification. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'07)*, volume 4916 of *LNCS*, pages 149–163. Springer, 2008.

# Fighting State Space Explosion: Review and Evaluation

Radek Pelánek*

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

**Abstract.** In order to apply formal methods in practice, the practitioner has to comprehend a vast amount of research literature and realistically evaluate practical merits of different approaches. In this paper we focus on explicit finite state model checking and study this area from practitioner's point of view. We provide a systematic overview of techniques for fighting state space explosion and we analyse trends in the research. We also report on our own experience with practical performance of techniques. Our main conclusion and recommendation for practitioner is the following: be critical to claims of dramatic improvement brought by a single sophisticated technique, rather use many different simple techniques and combine them.

## 1 Introduction

If you are a practitioner who wants to apply formal methods in industrial critical systems, you have to address following problems: Which of the many approaches should I use? If I want to improve the performance of my tool, which of the techniques described by researchers should I use? Which techniques are worth the implementation effort? These are important questions which are not easy to answer, nevertheless they are seldom addressed in research papers.

Research papers rather propose a steady flow of novel techniques, improvements, and optimizations. However, the experimental work reported in research papers has often poor quality [59] and it is difficult to judge the practical merit of proposed techniques. The goal of this paper is to provide an overview of research and realistic assessment of practical merits of techniques. The paper should serve as a guide for a practitioner who is trying to answer the above given questions.

It is not feasible to realize this goal for the whole field of formal verification at once. Therefore, we focus on one particular area (explicit model checking) and give an overview of research in this area and report on practical experience. Even though our discussion is focused on one specific area, we believe that our main recommendation — that it is better to combine several simple techniques rather than to focus on one sophisticated one — is applicable to many other areas of formal methods.

* Partially supported by GA ČR grant no. 201/07/P035.

D. Cofer and A. Fantechi (Eds.): FMICS 2008, LNCS 5596, pp. 37–52, 2009.

Explicit model checking is principally very simple — a brute force traversal of all possible model states. Despite the simplicity of the basic idea, explicit model checking is still the best approach for many practically important areas of application, e.g., verification of communication protocols and software. The popularity of the approach is illustrated by large number of available tools (e.g., Spin, CADP, mCRL2, Uppaal, Divine, Java PathFinder, Helena) and widespread availability of courses and textbooks on the topic (e.g., [10]).

The main obstacle in applying explicit model checking in practice is the state space explosion problem. Hence, the research focuses mainly on techniques for fighting state space explosions — during the last 15 years more than 100 papers have been published on the topic, proposing various techniques for fighting state space explosion. What are these techniques and how can we classify them? What is the real improvement brought by these techniques? Which techniques are practically useful? Which techniques should a practitioner study and use?

We try to answer these questions, particularly we provide the following:

- We overview techniques for fighting state space explosion in explicit model checking and divide them into four main areas (Section 2).
- We review and analyse research on fighting state space explosion, and discuss main trends in this research (Section 3).
- We report on our own practical experience with application and evaluation of techniques for fighting state space explosion (Section 4).
- Based on the review of literature and our experience, we provide specific recommendation for practitioner in industry (Section 5).

The main aim of this paper is to present and support the following message: Rather than optimizing the performance of a single sophisticated technique, we should use many different simple techniques, study how to combine them, and how to run them effectively in parallel.

## 2 Overview of Techniques for Fighting State Space Explosion

Fig. 1. gives an algorithm Explore which explores the reachable part of the state space. This basic algorithm can be directly used for verification of simple safety properties; for more complex properties, we have to use more sophisticated algorithms (e.g., cycle detection [72]). Nevertheless, the basic ideas of techniques for fighting state space explosion are similar. For clarity, we discuss these techniques mainly with respect to the basic Explore algorithm.

The main problem of explicit state space exploration is state space explosion problem and consequently memory and time requirements of the algorithm Explore. Techniques for fighting state space explosion can be divided into four main groups:

```
proc Explore(M)
  Wait = {s0}; Visited = ∅
  while Wait ≠ ∅ do
        get s from Wait
        explore state s
        foreach s' ∈ successors of s do
          if s' ∉ Visited then
                              add s' to Wait
                              add s to Visited fi od
  od
end
```

**Fig. 1.** The basic algorithm which explores all reachable states. Data structure *Wait* (also called open list) holds states to be visited, data structure *Visited* (also called visited list, closed list, transposition table, or just hash table) stores already explored states.

1. Reduce the number of states that need to be explored.
2. Reduce the memory requirements needed for storing explored states.
3. Use parallelism or distributed environment.
4. Give up the requirement on completeness and explore only part of the state space.

In the following we discuss these four types of approaches and for each of them we list examples of specific techniques.

### 2.1 State Space Reductions

When we inspect some simple models and their state spaces, we quickly notice significant redundancy in these state spaces. So the straightforward idea is to try to exploit this redundancy and reduce the number of states visited during the search. In order to exploit this idea in practice, we have to specify which states are omitted from the search and we have to show the correctness of the approach, i.e., prove that the visited part of the state space is equivalent to the whole state space with respect to some equivalence (typically bisimulation or stutter equivalence).

**State based reductions.** State based reductions exploit observation that if two states are bisimilar then it is sufficient to explore successors of only one of them. The reduction can be performed either on-the-fly during the exploration or by a static modification of the model before the exploration. Examples of such reductions are symmetry reduction [12,20,43,44,70,74], live variable reduction [21,69], cone of influence reductions, and slicing [19,35].

**Path based reductions.** Path based reductions exploit observation that sometimes it is sufficient to explore only one of two sequences of actions because they are just different linearizations of "independent" actions and therefore have the

same effect. These reductions try to reduce the number of equivalent interleavings. Examples of such reductions are transition merging [18,48], partial order reduction [27,33,40,63,64], $\tau$-confluence [11], and simultaneous reachability analysis [55].

**Compositional methods.** Systems are often specified as a composition of several components. This structure can be exploited in two ways: compositional generation of the state space [46] and assume-guarantee approach [22,32,66].

### 2.2 Storage Size Reductions

The main bottleneck of model checking are usually memory requirements. Therefore, we can save some memory at the cost of using more time, i.e., by employing some kind of time-memory trade-off. The main source of memory requirements of the algorithm Explore is the structure *Visited* which stores previously visited states. Hence, techniques, which try to lower memory requirements, focus mainly on this structure.

**State compression.** During the search, each state is represented as a byte vector which can be quite large (e.g., 100 bytes). In order to save space, this vector can be compressed [25,26,30,39,49,56,73] or common components can be shared [38]. Instead of compressing individual states, we can also represent the whole structure *Visited* implicitly as a minimized deterministic automaton [41].

**Caching and selective storing.** Instead of storing all states in the structure *Visited*, we can store only some of these states — this approach can lead to revisits of some states and hence can increase runtime, but it saves memory. Techniques of this type are for example:

- caching [24,28,65], which deletes some currently stored states when the memory is full,
- selective storing [9,49], which stores only some states according to given heuristics,
- sweep line method [15,54,68], which uses so called progress function; this function guarantees that some states will not be revisited in the future and hence these states can be deleted from the memory.

**Use of magnetic disk.** Simple use of magnetic disk leads to an extensive swapping and slows down the computation extremely. So the magnetic disk have to be used in a sophisticated way [7,8,71] in order to minimize disk operations.

### 2.3 Parallel and Distributed Computation

Another approach to manage a large number of states is to use even more brute force — more processors.

**Networks of workstations.** Distributed computation can be realized most easily by network of workstations connected by fast communication medium

(i.e., workstations communicate by message passing). In this setting the state space is partitioned among workstations (i.e., each workstation stores part of the data structure *Visited*) and workstations exchange messages about states to be visited (*Wait* structure), see e.g., [23,50,51]. The application of distributed environment for verification of liveness properties is more complicated, because classical algorithms are based on depth-first search, which cannot be easily adapted for distributed environment. Hence, for verification of liveness properties we have to use more sophisticated algorithms, see e.g., [1,2,3,5,13,14].

**Multi-core processors.** Recently, multi-core processors become widely available. Multi-core processors provide parallelism with shared memory, i.e., the possibility to reduce run-time of the verification by parallel exploration of several states, see e.g., [4,42].

### 2.4 Randomized Techniques and Heuristics

If the memory requirements of the search are too large even after the application of above given techniques, we can use randomized techniques and heuristics. These techniques explore only part of the state space. Therefore, they can help only in the detection of an error; they cannot assist us in proving correctness.

**Heuristic search** (also called directed or guided search). States are visited in an order given by some heuristics, i.e., *Wait* list is implemented as priority queue [31,47,67]. Different heuristic approach is to use genetic algorithm which tries to 'evolve' a path to a goal state [29].

**Random walk and partial search.** Random walk does not store any information and always visits just one successor of a current state [34,60]. This basic strategy can be extended in several ways, e.g., by visiting a subset of all successors (instead of just one state), storing some states in the *Visited* structure, or combining random walk with local breadth-first search, see e.g., [36,45,52,53,60].

**Bitstate hashing.** The algorithm does not store whole states but only one bit per state in a large hash table [37]. In a case of collision some states are omitted by the search. A more involved version of this technique is based on Bloom filters [16,17].

## 3 Research Analysis

What are the trends in the research literature about techniques for fighting state space explosion? Is the quality of experimental evidence improving? How significant is the improvement reported in research papers? How is this improvement changing over time?

### 3.1 Research Papers

In order to answer the above given questions, we have collected and analyzed large set of research papers. More specifically, we collected research papers that

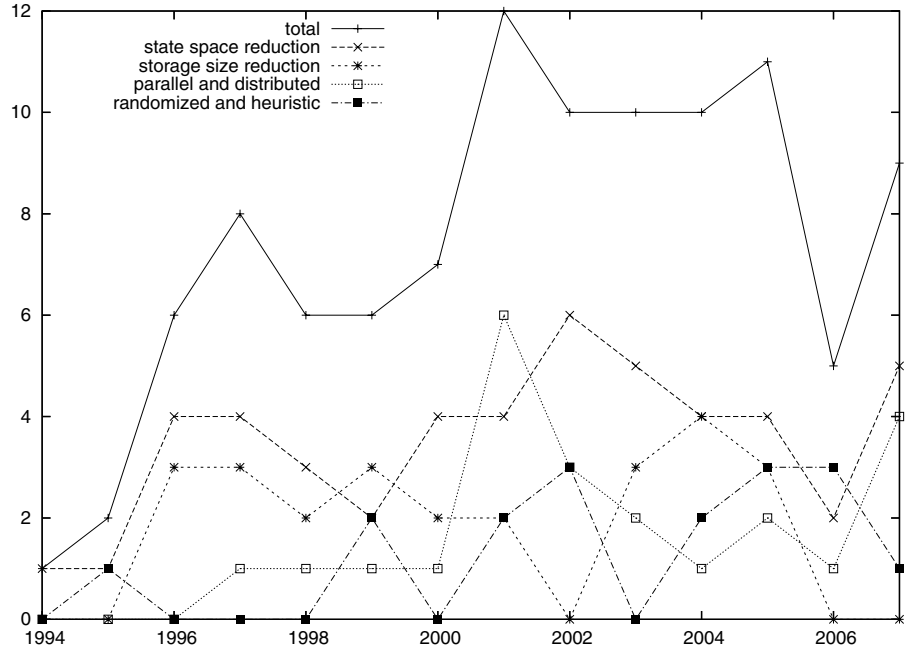



**Fig. 2.** Numbers of publications; note that some papers can be counted in two categories

describe techniques for fighting state space explosion in explicit model checking of finite state systems[1].

The collection contains more than 100 papers – these papers were obtained by systematically collecting papers from the most relevant conferences and by citation tracking. The full list of reviewed papers, which includes all papers referenced above, is freely available[2]. The collection is certainly not complete, but we believe that it is a good sample of research in the area.

Fig. 2. shows the number of publications in each year during the last 13 years. Although there are rises and downfalls, the overall flow of publications on the topic is rather steady. The figure also shows that all four areas described in the previous section are pursued concurrently.

### 3.2 Quality of Experiments

Although some of the considered research is rather theoretically oriented (e.g., partial order reduction), all considered techniques are in fact heuristics which aim at improving performance of model checking tools. So what really matters is the practical improvement brought by each technique. To assess the improvement

[1] In few cases we also include techniques which are not purely explicit, but target the similar application domain (i.e., the experiments are done on same models as for other included papers).

[2] `http://www.fi.muni.cz/~xpelanek/amase/reductions.bib`

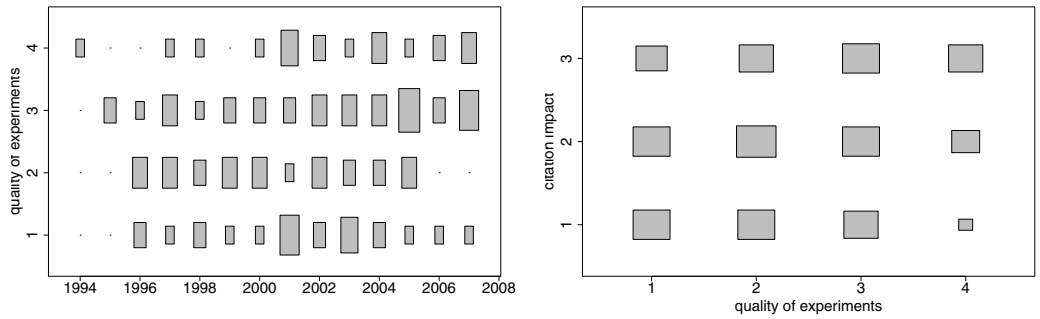



**Fig. 3.** The first graph shows the quality of experiments reported in model checking papers during time. The size of a box corresponds to a number of published papers in a given year and quality category. The second graph shows the relation between experiment quality and citation impact; citation impact is divided into three categories: less than 10 citations, 10–30 citations, more than 30 citations; only publication before 2004 are used.

it is necessary to perform experimental evaluation. Only good experiments can provide realistic evaluation of practical merits of proposed techniques.

In order to study the quality of experiments, we classify experiments in each paper into one of four classes, depending on the number and type of used models[3]:

1. Random inputs or few toy models.
2. Several toy models (possibly parametrized) or few simple models.
3. Several simple models (possibly parametrized) or one large case study.
4. Exhaustive study of parametrized simple models or several case studies.

Fig 3. presents the quality of experiments in papers from our sample. The figure shows that the quality on average is not very good and, what is even more disappointing, that there is slow progress in time, although many realistic case studies are available (see [59] for more detailed discussion of these issues).

Since there is a large number of techniques, it is important to compare performance of novel techniques with previously studied one. However, analysis of our research sample shows that only about 40% papers contain some comparison with similar techniques; this ratio is improving with time, but only slowly. Moreover, the comparison is usually only shallow.

Our analysis also shows one encouraging trend. Fig. 3. shows that there is a relation between quality of experiments and citation impact of a paper — research with better experiments is more cited.

### 3.3 Reported Improvement

Before the discussion of improvements reported in research papers, we clarify the terminology that we use to measure this improvement. We use the notion

[3] The classification is clearly slightly subjective. Nevertheless, we believe that the main conclusions of our analysis do not depend on the subjective factor.

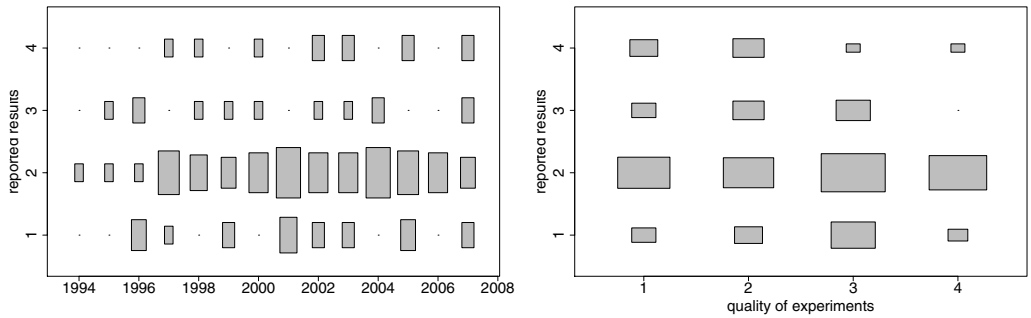



**Fig. 4.** Reported improvement with respect to time and quality of experiments

'reduction ratio' to denote the ratio between the memory consumption of the technique for fighting state space explosion and the memory consumption of the standard reachability (exploration of full reachable state space). Some authors report 'reduced by' factor, i.e., if we report 'reduction ratio' 80%, it means that the memory consumption was 'reduced by' 20%. Note that in this section we analyze reduction ratios as reported by authors, not what we consider realistic reduction ratios of techniques.

For clarity of presentation, we again divide the reported reduction ratios into four classes:

1. Reported reduction ratio is 50% or worse (or sometimes good but sometimes worse than 100%).
2. Reported reduction ratio is in most cases 10%-50%.
3. Reported reduction ratio is in most cases 1%-10%.
4. Reported reduction ratio is better than 1% (or exponential improvement is reported or only out-of memory for full search is reported, i.e., reduction ratio is impossible to assess).

Fig. 4. shows that in most cases the reported reduction is in the second category (reduction between 10% and 50%). The relation with the quality of experiments clearly demonstrates that this is also the most realistic evaluation — better results are often caused by poor experiments, not by special features of techniques.

There is no clear trend with respect to time, i.e., it seems that novel techniques do not significantly improve on performance of previous techniques. This does not automatically mean that the recent research is misguided. In some cases novel technique provides principally different way how to obtain the reduction and can be combined with previously proposed techniques in orthogonal way. Novel technique can also extend the application domain of previously studied techniques.

As we already mentioned, the research in this domain is purely practically motivated. However, the amount of research into certain topic is not really related to its practical merit. For example, our experience (described in next section) shows that the dead variable reduction brings similar improvement as partial order reduction. Nevertheless, there are significantly more research papers about

partial order reduction than about dead variable reduction. This is probably not due to the practical merits of partial order reduction, but because it can be extensively studied theoretically.

## 4 Practical Experience

In this section we report on our experience with techniques for fighting state space explosion. Our experience is based on large-scale research studies, which are described in stand-alone publications. Here we provide only a brief description of these studies and present their main conclusions. Technical details can be found in cited papers.

Our experience report obviously does not cover all techniques for fighting state space explosion. However, we cover all four areas described in Section 2 and the main conclusions are in all cases similar, so we believe that it is reasonable to generalize our experience.

### 4.1 On-the-fly State Space Reductions

We evaluated several techniques for on-the-fly state space reductions. Setting of this study (see [57] for details):

- Implementation: publicly available implementations of explicit model checkers (Spin, Murphy, DiVinE).
- Models: models included in tool distributions plus few more publicly available case studies.
- Techniques: dead variable reduction, partial order reduction, transition merging, symmetry reduction.

When we measured the performance of techniques over realistic models, we found that the reduction ratio is usually worse than what is reported in research papers — research papers often use simple models with artificially high values of model parameters. More specifically, the main results of our evaluation are the following: dead variable reduction works on nearly all models, reduction ratio is usually between 10% and 90%; partial order reduction works only in some cases, reduction ratio is between 5% and 90%; transition merging works in similar cases as partial order reduction, it is weaker but easier to realize, reduction ratio is usually between 50% and 95%; symmetry reduction works only for few models (symmetric ones), reduction ratio is usually between 8% and 50%.

Our main conclusion from this study are the following:

- Each technique is applicable only to some types of models. No technique works really universally; more specialized techniques yield better reduction.
- On real models, no single technique is able to achieve reduction ratio significantly under 5%. Claims about drastic reduction, which occur in some papers, are not really appropriate.
- Since there are many techniques and many of them are orthogonal, most models can be reduced quite significantly.

### 4.2 Caching and Compression

From the area of 'storage size reduction' techniques we evaluated two techniques for reducing memory consumption of the data structure *Visited*. Setting of this study (see [62] for details):

- Implementation: all techniques are implemented in uniform way using the DiVinE environment [6] (source codes are publicly available).
- Models: 120 models from BEEM (BEnchmarks for Explicit Model checkers) [59].
- Techniques: state caching with 7 different caching strategies, state compression with Huffman coding (two variants: static code and code computed by training runs).

In the study we also reviewed previous research on storage size reduction techniques. We found that using proper parameter values with our simple and easy-to-implement techniques, we were able to achieve very similar results to those reported in other works which use far more sophisticated approaches. Concrete results of the evaluation are the following:

- Caching strategies are to a certain degree complementary. Using an appropriate state caching strategy, the reduction ratio is in most cases 10% to 30%.
- Using state compression, the reduction ratio is usually around 60%.
- The two techniques combine well.

### 4.3 Distributed Exploration

From the area of parallel and distributed techniques we report on the basic distributed approach to explicit model checking: we have a network of workstations connected by fast Ethernet, workstations communicate via message passing (MPI library), state space is partitioned among workstations. In this setting the reduction ratio is clearly bounded by $1/n$, where $n$ is the number of workstations. In practice the reduction ratio is worse because of communication overhead. Here we report on results of our evaluation, however the results are rather typical in this area.

Setting of this study (see [58] for details):

- Implementation: the DiVinE tool (public version).
- Models: 120 models from BEEM (BEnchmarks for Explicit Model checkers) [59].
- Techniques: distributed reachability on 20 workstations.

In this study the speedup varies from 2 to 12, typical value of the speedup is between 4 and 6 (i.e., reduction ratio around 20%). We also found that the speedup is negatively correlated with the speed of successor generation by the tool.

### 4.4 Error Detection Techniques

From the area of 'randomized techniques and heuristics' we have chosen 9 techniques and evaluated their performance. In this case we do not study the reduction ratio, because it is not known — the experiments are done on models for which the standard reachability is not feasible. Therefore, we focus on relative performance of techniques and on the issue of complementarity.

Setting of this study (see [61] for details):

- Implementation: all techniques are implemented in uniform way using the DiVinE environment [6] (source codes are publicly available),
- Models: 54 models (with very large state space) from BEEM [59].
- Techniques: breadth-first search, depth-first search, randomized DFS, two variants of random walk, bitstate hashing with repetition, two variants of directed search, and under-approximation refinement based on partial order reduction.

For the evaluation we used several performance measures: number of steps needed to find an error, length of reported counterexample, and coverage metrics. The main results of this study are the following:

- There is no single best technique. Results depend on used performance metrics, even for a given metric, the most successful technique is the best one only over 25% of models.
- It is important to focus on complementarity of techniques, not just on their overall (average) performance. For example, in our study the random walk technique had rather poor overall performance, but it was successful on models where other techniques fall, i.e., it is a useful technique which we should not discard.

## 5 Conclusions

This paper is concerned with techniques for fighting state space explosion problem in explicit model checking. We review the research in the area during the last 15 years (more than 100 research papers) and report on our practical experience. As a result of our review we identify four main groups of techniques: state space reductions, storage size reductions, parallel and distributed computation, randomization and heuristics. These four groups are rather orthogonal and can be combined; within each group techniques are often based on similar ideas and their combination can be difficult.

The review of research shows that despite a steady flow of publications on the topic, the progress is not very significant — in fact the reduction ratio reported in research papers stays practically the same over the last 15 years. This analysis stresses the need for good practical evaluation. However, realistic evaluation of research progress is complicated by rather poor experimental standards and by unjustified claims by researchers.

Results reported in research papers often make an impression of dramatic improvements. Our practical experience suggests that it is not realistic to get better reduction ratio than 5% with a single technique, in fact in most cases the obtained reduction ratio is between 20% and 80%. Nevertheless, this does not mean that techniques for fighting state space explosion are not useful. Techniques of different types can be combined, and together they might be able to bring a significant improvement.

Our experience also suggest that simple techniques are often sufficient. The performance obtained by sophisticated techniques is often similar to performance of basic techniques from each area. Complicated techniques often achieve better results only for specialized application domains. This observation can be also supported by analysis of techniques implemented in model checking tools. Tools usually implement basic versions of many techniques, sophisticated techniques are often implemented only in a tool used by authors of the technique.

To summarise, we propose following recommendations for those who want to apply model checking in practice:

- Use large number of simple techniques of different types.
- Do not try to find 'the best' technique of a specific type. Try to find a set of simple complementary techniques and run all of them (preferably in parallel).
- Be critical to claims in research papers, particularly if the experimental evidence is poor.
- Use sophisticated techniques only if they are specifically targeted at your domain of application.
- Focus on combination of orthogonal techniques.

Researchers, we believe, should focus not just on the development of novel techniques, but also on issues of techniques combination, selection, and efficient scheduling: How to select right technique for a given model? In what order we should try available techniques? Can information gathered by one technique be used by another techniques?

## Acknowledgment

I thank Václav Rosecký, Pavel Moravec, and Jaroslav Šeděnka for cooperation on practical evaluation of techniques.

## References

1. Barnat, J., Brim, L., Černá, I.: Property driven distribution of nested DFS. In: Proc. of Workshop on Verification and Computational Logic, number DSSE-TR-2002-5 in DSSE Technical Report, pp. 1–10. University of Southampton, UK (2002)
2. Barnat, J., Brim, L., Chaloupka, J.: Parallel breadth-first search LTL model-checking. In: Proc. of Automated Software Engineering (ASE 2003), pp. 106–115. IEEE Computer Society, Los Alamitos (2003)

3. Barnat, J., Brim, L., Chaloupka, J.: From distributed memory cycle detection to parallel LTL model checking. ENTCS 133(1), 21–39 (2005)
4. Barnat, J., Brim, L., Rockai, P.: Scalable multi-core LTL model-checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 187–203. Springer, Heidelberg (2007)
5. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL model-checking in SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 200–216. Springer, Heidelberg (2001)
6. Barnat, J., Brim, L., Černá, I., Moravec, P., Rockai, P., Šimeček, P.: DiVinE - a tool for distributed verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006), `http://anna.fi.muni.cz/divine`
7. Barnat, J., Brim, L., Šimeček, P.: I/o efficient accepting cycle detection i/o efficient accepting cycle detection. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 281–293. Springer, Heidelberg (2007)
8. Barnat, J., Brim, L., Šimeček, P., Weber, M.: Revisiting resistance speeds up i/o-efficient ltl model checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 48–62. Springer, Heidelberg (2008)
9. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
10. Ben-Ari, M.: Principles of the SPIN Model Checker. Springer, Heidelberg (2008)
11. Blom, S., van de Pol, J.: State space reduction by proving confluence. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 596–609. Springer, Heidelberg (2002)
12. Bosnacki, D.: A light-weight algorithm for model checking with symmetry reduction and weak fairness. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 89–103. Springer, Heidelberg (2003)
13. Brim, L., Černá, I., Krčál, P., Pelánek, R.: Distributed LTL model checking based on negative cycle detection. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 96–107. Springer, Heidelberg (2001)
14. Černá, I., Pelánek, R.: Distributed explicit fair cycle detection. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
15. Christensen, S., Kristensen, L.M., Mailund, T.: A sweep-line method for state space exploration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
16. Dillinger, P.C., Manolios, P.: Bloom filters in probabilistic verification. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 367–381. Springer, Heidelberg (2004)
17. Dillinger, P.C., Manolios, P.: Fast and accurate bitstate verification for SPIN. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 57–75. Springer, Heidelberg (2004)
18. Dong, Y., Ramakrishnan, C.R.: An optimizing compiler for efficient model checking. In: Proc. of Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), pp. 241–256. Kluwer, Dordrecht (1999)
19. Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V.P.: Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 73–89. Springer, Heidelberg (2006)

20. Emerson, E.A., Wahl, T.: Dynamic symmetry reduction. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 382–396. Springer, Heidelberg (2005)
21. Fernandez, J.C., Bozga, M., Ghirvu, L.: State space reduction based on live variables analysis. Journal of Science of Computer Programming (SCP) 47(2-3), 203–220 (2003)
22. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
23. Garavel, H., Mateescu, R., Smarandache, I.: Parallel state space construction for model-checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 217–234. Springer, Heidelberg (2001)
24. Geldenhuys, J.: State caching reconsidered. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 23–38. Springer, Heidelberg (2004)
25. Geldenhuys, J., de Villiers, P.J.A.: Runtime efficient state compaction in SPIN. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 12–21. Springer, Heidelberg (1999)
26. Geldenhuys, J., Valmari, A.: A nearly memory-optimal data structure for sets and mappings. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 136–150. Springer, Heidelberg (2003)
27. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. In: Godefroid, P. (ed.) Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, p. 142. Springer, Heidelberg (1996)
28. Godefroid, P., Holzmann, G.J., Pirottin, D.: State space caching revisited. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 178–191. Springer, Heidelberg (1993)
29. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 266–280. Springer, Heidelberg (2002)
30. Gregoire, J.: State space compression in spin with GETSs. In: Proc. Second SPIN Workshop, Rutgers University, New Brunswick, New Jersey (1996)
31. Groce, A., Visser, W.: Heuristics for model checking java programs. Software Tools for Technology Transfer (STTT) 6(4), 260–276 (2004)
32. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16(3), 843–871 (1994)
33. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007)
34. Haslum, P.: Model checking by random walk. In: Proc. of ECSEL Workshop (1999)
35. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Higher Order Symbol. Comput. 13(4), 315–353 (2000)
36. Holzmann, G.J.: Algorithms for automated protocol verification. AT&T Technical Journal 69(2), 32–44 (1990)
37. Holzmann, G.J.: An analysis of bitstate hashing. In: Proc. of Protocol Specification, Testing, and Verification, pp. 301–314. Chapman & Hall, Boca Raton (1995)
38. Holzmann, G.J.: State compression in SPIN: Recursive indexing and compression training runs. In: Proc. of SPIN Workshop (1997)
39. Holzmann, G.J., Godefroid, P., Pirottin, D.: Coverage preserving reduction strategies for reachability analysis. In: Proc. of Protocol Specification, Testing, and Verification (1992)

40. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Proc. of Formal Description Techniques VII, pp. 197–211. Chapman & Hall, Ltd., Boca Raton (1995)
41. Holzmann, G.J., Puri, A.: A minimized automaton representation of reachable states. Software Tools for Technology Transfer (STTT) 3(1), 270–278 (1998)
42. Holzmann, G.J., Bosnacki, D.: The design of a multicore extension of the spin model checker. IEEE Transactions on Software Engineering 33(10), 659–674 (2007)
43. Iosif, R.: Symmetry reduction criteria for software model checking. In: Bošnački, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 22–41. Springer, Heidelberg (2002)
44. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design 9(1–2), 41–75 (1996)
45. Jones, M.D., Sorber, J.: Parallel search for LTL violations. Software Tools for Technology Transfer (STTT) 7(1), 31–42 (2005)
46. Krimm, J.P., Mounier, L.: Compositional state space generation from Lotos programs. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 239–258. Springer, Heidelberg (1997)
47. Kuehlmann, A., McMillan, K.L., Brayton, R.K.: Probabilistic state space search. In: Proc. of Computer-Aided Design (CAD 1999), pp. 574–579. IEEE Press, Los Alamitos (1999)
48. Kurshan, R.P., Levin, V., Yenigün, H.: Compressing transitions for model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 569–581. Springer, Heidelberg (2002)
49. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: Compact data structure and state-space reduction. In: Proc. of Real-Time Systems Symposium (RTSS 1997), pp. 14–24. IEEE Computer Society Press, Los Alamitos (1997)
50. Lerda, F., Sisto, R.: Distributed-memory model checking with SPIN. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, p. 22. Springer, Heidelberg (1999)
51. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 80–102. Springer, Heidelberg (2001)
52. Lin, F., Chu, P., Liu, M.: Protocol verification using reachability analysis: the state space explosion problem and relief strategies. Computer Communication Review 17(5), 126–134 (1987)
53. Mihail, M., Papadimitriou, C.H.: On the random walk method for protocol testing. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 132–141. Springer, Heidelberg (1994)
54. Mailund, T., Westergaard, W.: Obtaining memory-efficient reachability graph representations using the sweep-line method. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 177–191. Springer, Heidelberg (2004)
55. Ozdemir, K., Ural, H.: Protocol validation by simultaneous reachability analysis. Computer Communications 20, 772–788 (1997)
56. Parreaux, B.: Difference compression in SPIN. In: Proc. of Workshop on automata theoric verification with the SPIN model checker (SPIN 1998) (1998)
57. Pelánek, R.: Evaluation of on-the-fly state space reductions. In: Proc. of Mathematical and Engineering Methods in Computer Science (MEMICS 2005), pp. 121–127 (2005)
58. Pelánek, R.: Web portal for benchmarking explicit model checkers. Technical Report FIMU-RS-2006-03, Masaryk University Brno (2006)

59. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
60. Pelánek, R., Hanžl, T., Černá, I., Brim, L.: Enhancing random walk state space exploration. In: Proc. of Formal Methods for Industrial Critical Systems (FMICS 2005), pp. 98–105. ACM Press, New York (2005)
61. Pelánek, R., Rosecký, V., Moravec, P.: Complementarity of error detection techniques. In: Proc. of Parallel and Distributed Methods in verifiCation (PDMC) (2008)
62. Pelánek, R., Rosecký, V., Šeděnka, J.: Evaluation of state caching and state compression techniques. Technical Report FIMU-RS-2008-02, Masaryk University Brno (2008)
63. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
64. Penczek, W., Szreter, M., Gerth, R., Kuiper, R.: Improving partial order reductions for universal branching time properties. Fundamenta Informaticae 43(1-4), 245–267 (2000)
65. Penna, G.D., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.V.: Exploiting transition locality in automatic verification of finite state concurrent systems. Software Tools for Technology Transfer (STTT) 6(4), 320–341 (2004)
66. Pnueli, A.: In transition from global to modular temporal reasoning about programs. Logics and models of concurrent systems, 123–144 (1985)
67. Qian, K., Nymeyer, A.: Guided invariant model checking based on abstraction and symbolic pattern databases. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 497–511. Springer, Heidelberg (2004)
68. Schmidt, K.: Automated generation of a progress measure for the sweep-line method. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 192–204. Springer, Heidelberg (2004)
69. Self, J.P., Mercer, E.G.: On-the-fly dynamic dead variable analysis. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 113–130. Springer, Heidelberg (2007)
70. Sistla, A.P., Godefroid, P.: Symmetry and reduced symmetry in model checking. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 91–103. Springer, Heidelberg (2001)
71. Stern, U., Dill, D.L.: Using magnetic disk instead of main memory in the murphi verifier. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
72. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Kozen, D. (ed.) Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986), pp. 332–344. IEEE Computer Society Press, Los Alamitos (1986)
73. Visser, W.: Memory efficient state storage in SPIN. In: Proc. of SPIN Workshop, pp. 21–35 (1996)
74. Wahl, T.: Adaptive symmetry reduction. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 393–405. Springer, Heidelberg (2007)

# Chapter 11

# Verification Manager: Automating the Verification Process

In this work we further develop the concept of a verification manager (as outlined in Section 1.3.3). Particularly, we describe a practical realization of this concept for explicit model checking by building a tool EMMA (Explicit Model checking MAnager). The design of the tool is based on our experiences with evaluation of individual techniques (as discussed in other papers in the thesis), i.e., rather then developing few sophisticated techniques, we employ a large number of simple techniques which are executed in parallel.

We also discuss practical experience with the tool. We pay special attention to the problem of selection of problems for experiments. This issue is important (but often neglected) in all experiments, but it becomes crucial in evaluating strategies for the verification manager, which are in principle meta-heuristics.

This paper was published as a technical report, short version of the paper was published in proceedings of Model Checking Software (SPIN Workshop) in 2009:

- R. Pelánek and V. Rosecký. EMMA: Explicit Model Checking Manager (Tool Presentation). In *SPIN Workshop on Model Checking Software*, volume 5578 of LNCS, pages 169-173. Springer, 2009.
- R. Pelánek, V. Rosecký. Verification Manager: Automating the Verification Process. FIMU-RS-2009-02, 17 stran, 2009.

The author of the thesis is one of two coauthors and has done data analysis and most of the writing.

# Verification Manager: Automating the Verification Process*

Radek Pelánek and Václav Rosecký

Department of Information Technology, Faculty of Informatics
Masaryk University Brno, Czech Republic
pelanek,xrosecky@fi.muni.cz

**Abstract.** Although model checking is usually described as an automatic technique, the verification process with the use of model checker is far from being fully automatic. With the aim of automating the verification process, we elaborate on a concept of a verification manager. The manager automates some step of the verification process and enables efficient parallel combination of different verification techniques. We describe a realization of this concept for explicit model checking and discuss practical experience. Particularly, we discuss the problem of selection of input problems for evaluation of this kind of tool.

## 1 Introduction

Model checking consists of three phases: modeling, specification, and algorithmic verification. The first two are acknowledged to involve manual effort and user expertise, although researchers have proposed several techniques to automate these steps [2,4,7]. The third step is standardly considered to be fully automatic. We argue that in practice even the third step requires significant manual effort and user expertise and that it is important to focus on automating even this step.

### 1.1 Motivation for Automating the Verification Process

Given a model and a specification, model checking algorithmically checks all possible behaviours of the model and gives us 'yes' or 'no' answer. In practice, however, model checking techniques often reach a limit (on time or memory consumption) and do not give any clear answer. To obtain an answer, it is sometimes necessary to restore to a more abstract model, but in many cases it is sufficient to suitably tune parameters of the model checker. Hence the process of using a model checker can be quite elaborate and far from automatic.

In order to successfully verify a model, it is often necessary to select appropriate techniques and parameter values. The selection is difficult, because there is a very large number of different heuristics and optimization techniques – our

* Partially supported by GA ČR grant no. 201/07/P035.

review of techniques [17] found more than 100 papers just in the area of explicit model checking. These techniques are often complementary and there are non-trivial trade-offs which are hard to understand. In general, there is no best technique. Some techniques are more suited for verification, other techniques are better for detection of errors. Some techniques bring good improvement in a narrow domain of applicability, whereas in other cases they can worsen the performance [17]. The user needs a significant experience to choose good techniques.

Moreover, models are usually parametrized and there are several properties to be checked. Thus the process of verification requires not just experience, but also a laborious effort, which is itself error prone.

Another motivation for automating the verification process comes from trends in the development of hardware. Until recently, the performance of model checkers was continually improved by increasing processor speed. In last years, however, the improvement in processors speed has slowed down and processors designers have shifted their efforts towards parallelism [9]. This trend poses a challenge for further improvement of model checkers. A classic approach to application of parallelism in model checking is based on distribution of a state space among several workstations (processors) [8,11,25]. This approach, however, involves large communication overhead. Given the large number of techniques and hard-to-understand trade-offs, there is another way to employ parallelism: to run independent verification runs on individual workstations (processors) [9,17,20]. This approach, however, cannot be efficiently performed manually. We need to automate the verification process.

### 1.2 Our Proposal: Verification Manager

In our overview study of techniques for fighting state space explosion, we reached the following conclusion [17]: "Simple techniques are often sufficient. Rather then optimizing the performance of sophisticated techniques, we should use simple techniques, and study how to combine these simple techniques, and how to run them effectively in parallel." In this work we try to realize this idea.

We propose a concept of a verification manager. Verification manager is a tool which automates the verification process. As an input it takes a (parametrized) model and a list of properties. Then it employs available resources (hardware, verification techniques) to perform verification – the manager distributes the work among individual workstations, it collects results, and informs the user about progress and final results. Decisions of the manager (e.g., which technique should be started) are governed by a 'verification strategy'. The verification strategy needs to be written by an expert user, but since it is generic, it can be used on many different models. In this way even a layman user can exploit experiences of expert users. The general concept of verification manager is further developed in Section 2.

As a proof of concept we introduce a prototype of the verification manager for an area of explicit model checking – Explicit Model checking MAnager (EMMA). Realization of EMMA is described in Section 3. We also describe experiences

with EMMA over models from the benchmark set BEEM [16]. Our experiences with evaluation raise some methodological issues – it turns out that it is quite difficult to perform fair evaluation of this kind of tool. These experiences are described in Section 4.

### 1.3 Related work

This paper is part of our long term effort. We have developed the BEEM benchmark set [16] and using this benchmark set we have performed several evaluation studies [20,21]. With the use of these studies we have provided overview of techniques for fighting state space explosion [17]. In one of our publications [18] we have already suggested the basic concept of a verification manager. In this work we integrate all this previous progress and provide realization of the manager, which is based on insight gained from previous studies.

The most related work by other researchers is by Holzmann et al. Holzman and Smith [10] describe a tool for automated execution of verification runs for several model parameters and correctness properties; they use one fixed verification technique. Recently, Holzmann et al. [9] proposed 'swarm verification', which is based on parallel execution of many different techniques. Their approach, however, does not allow any communication among techniques and they do not discuss the selection of techniques that are used for the verification (verification strategy).

Owen et al. [13,14] discuss complementarity issues in verification and propose to combine different tools. They illustrate the principles on large case studies. The approach is, however, not automated. Garavel and Lang [6] propose a scripting language for description of verification strategies for automating the process of compositional verification. The script calls techniques sequentially and has to be written specifically for each model (as opposed to our approach in which the strategy can be used universally and which executes techniques in parallel).

There are several other works, which employ similar ideas in different context or on a more abstract level. Sahoo et al. [23] use sampling of the state space to decide which BDD based reachability technique is the best for a given model. Mony et al. [12] use expert system for automating proof strategies. Eytani et al. [5] give a high-level proposal to use an 'observation database' for sharing relevant information among different verification techniques.

## 2 Verification Manager

In this section we discuss the general proposal for the verification manager and verification strategy. In the next section we introduce a concrete realization of these concepts.

### 2.1 Verification Meta-Search

Most of the research in automated verification is focused on the verification search problem: given a model $M$ and a property $\varphi$, determine whether $M$

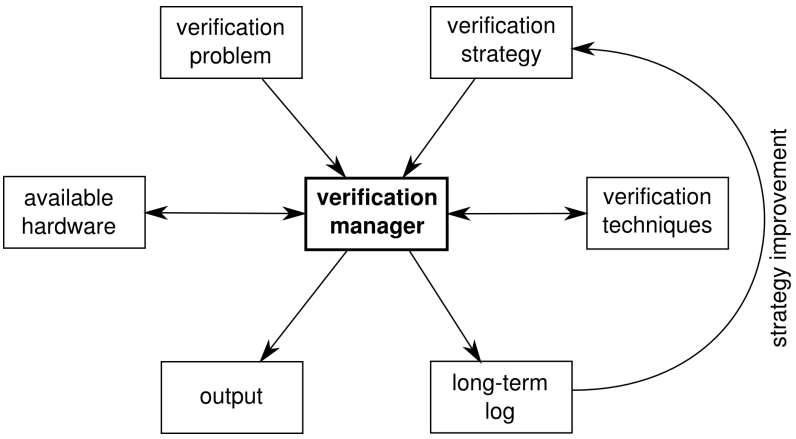



**Fig. 1.** Verification manager — context.

satisfies $\varphi$, i.e., the aim is to search for an incorrect behaviour or for a proof. We believe that it is worthwhile to consider the verification meta-search problem [18] as well: given a model and a property, find a technique $T$ and values $p$ of its parameters such that $T(p)$ can provide an answer to the verification problem. This can be viewed as a stand-alone search problem which uses algorithms for the verification problem as black-boxes. Our aim is to study methods for performing this meta-search and heuristics to guide the search.

We call an entity responsible for the verification meta-search a *verification manager* and a heuristic for the meta-search a *verification strategy*.

### 2.2 Functionality of the Manager

Let us be more specific about the functionality of the verification manager. Figure 1 gives the context, in which the manager operates:

**Verification problem** A model and a list of properties. The model may be parametrized, in such a case the input also contains list of instances (described by values of model parameters).

**Verification strategy** A heuristic for the meta-search problem (see below).

**Verification techniques** Available techniques which can be used for verification.

**Available hardware** Hardware available for verification (e.g., a network of workstations).

**Output** Running report about progress and final report about results.

**Long-term log** Stored data about performance of techniques. It can be used for improvement of strategy and for collecting benchmarking data (see discussion in Section 4).

Basic functionality of the verification manager consists of starting and stopping verification tasks. When starting a task, manager decides which technique

with what parameter values should be run on what computer. This decision is based on verification strategy, currently available hardware, and results obtained so far. There are several ways in which a technique can terminate:

- The technique simply finishes.
- Timeout specified by verification strategy is reached and manager forces the technique to terminate.
- Based on intermediate results manager concludes that all results for a given model instance are already known, and therefore it terminates the technique.

Manager also collects all results and presents them to the user.

Note that this is just a basic functionality, that we are able to realize at this moment (see Section 3). We believe that in future it is feasible and meaningful to further expand the functionality of the manager, e.g., by using static analysis to analyze a model, or by employing several tools and incorporating translation among specification languages into the manager (see future work in Section 5).

### 2.3 Verification Strategy

The verification manager performs the verification meta-search by starting and terminating verification tasks. But how should it proceed with the meta-search? Which techniques should be called? The "meta state space" cannot be searched exhaustively – taking into account parameters of techniques, the meta state space is infinite. Experimental results reported in research papers and overall experience of the community suggest that there is no optimal deterministic method to search the meta state space. We therefore need some heuristic for the meta-search – a verification strategy. This strategy should be optimized for an application domain of interest using a feedback from a long-term log.

There are two extreme approaches to realization of a verification strategy. The first one is to use "hard-wired" strategy, i.e., to include the strategy as an integral part of the manager and to encode it in a high-level programming language. The advantage of this approach is that we can encode arbitrary strategy in this way. The disadvantage is that the basic functionality of the manager is not separated from the heuristics and therefore it is harder to develop the strategy and to improve it with the use of feedback.

The second extreme approach is to let the manager construct the strategy on its own. With this approach we specify just the list of available techniques and some learning algorithm (e.g., classifier system or genetic algorithm) and we let the manager to construct a strategy by learning. This approach would require very large amount of data to work well. At this moment, we are not convinced that this approach would lead to better strategy than strategy constructed by human expert. Nevertheless, this approach may be useful in the long-term application of manager (learning with the use of feedback from long-term log).

We believe that a reasonable way is between these two approaches. We fix a basic skeleton of the strategy (e.g., priority based scheme) and implement support for this skeleton into the manager. Specifics of the strategy (e.g., order

of techniques, values of parameters) are specified separately in a simple format – this specification of strategy can be easily and quickly (re)written by an expert user.

## 3 Implementation: EMMA

In previous section we discussed the general concept of verification manager. Now we introduce a concrete implementation of the concept. Since this is the first step in this direction, we restrict our attention to explicit model checking and detection of safety errors. Our implementation is called EMMA (Explicit Model checking MAnager) and is available at:

`http://anna.fi.muni.cz/~xrosecky/emma_web`

### 3.1 Architecture

EMMA is based on the Distributed Verification Environment (DiVinE) [3]. All used verification techniques are implemented in C++ with the use of DiVinE environment. At the moment, we use the following techniques: breadth-first search, depth-first search, random walk, directed search, bitstate hashing (with refinement), and under-approximation based on partial order reduction. All techniques are publicly available either as a part of the DiVinE library or as a part of another published study [20,21]. Other techniques can be easily incorporated.

The manager itself is implemented in Java. At the moment manager supports as the underlying hardware a network of workstations connected by Ethernet. Communication is based on SSH and stream socket, it consists of the following messages (messages are encoded in XML):

- manager → workstation:
  - initialization of a particular verification technique,
  - forced termination of a verification technique (e.g., timeout),
- workstation → manager:
  - intermediate result when an error is detected,
  - final results after termination of a technique.

The verification manager EMMA takes the following inputs (in correspondence with Figure 1): strategy (see below), list of available techniques, list of available workstations, and description of a parametrized model. Model description consists of model source code (in DVE format [15]) and XML file describing parameter values and properties (compatible with the BEEM project [16]).

During the verification the manager outputs intermediate results and information about the progress of verification. At the end of the verification the manager provides all results and summary information.

```
<strategy>
  <total-timeout>1800</total-timeout>
  <run>
    <algorithm>random_walk</algorithm>
    <timeout>50</timeout>
    <params>
       <param>
          <name>max_depth</name>
          <value>500</value>
       </param>
    </params>
  </run>
  <run>
    <algorithm>random_dfs</algorithm>
    <timeout>10</timeout>
  </run>
  <run>
    <algorithm>bfs_reach</algorithm>
    <timeout>60</timeout>
  </run>
</strategy>
```

**Fig. 2.** Example of a strategy.

### 3.2 Strategy

Strategy description is given in the XML format. Specifically, we use two files to describe the strategy (see Fig. 2):

1. specification of verification techniques – this file contains for each technique:
   - name of a technique,
   - way to call the technique (i.e., name of executable file which should be called and its options),
   - list of parameters and their default values.
2. specification of a strategy – the verification strategy itself, i.e., specification of which techniques should be called, in what order.

For the first evaluation we use a simple priority-based strategies. For each technique we specify priority, timeout, and parameter values; techniques are executed in order according to their priorities.

The architecture of EMMA is built in such a way, that it can easily cope with more sophisticated strategies, particularly 'verification in phases'. With this approach, techniques are divided into several phases; in each phase techniques are called in parallel (e.g., in priority-based order); phases are executed sequentially. This approach provides the following possibilities:

- Start with a quick 'error detection phase' (many diverse error detection techniques) and then continue with a long 'verification phase' (few optimized technique for traversing the whole state space).

– We can have two consecutive phases with same techniques, based on result of the first phase we can modify priorities, timeouts, and parameter values for the second phase – employing the observation that different instances of a same parametrized model have similar state space properties and similar techniques work for them [19].

## 4 Experiences

In this section we report our (qualitative) experiences from using EMMA and we discuss why it is difficult to provide fair quantitative evaluation of the tool and different strategies.

### 4.1 General Experiences

We have done experiments with models from BEEM [16], using 4 workstations connected with fast Ethernet. The experiments were done with parametrized models with usually about 5 models and several properties to be checked. We set a overall timeout for the verification to 30 minutes. The general experiences are the following:

– Most of the results are obtained quickly (usually within the first minute). This stresses the usefulness of running report about results.
– The manager significantly simplifies the use of model checker for parametrized models even for an experienced user – this contribution is not easily measurable, but is very important for practical applications of model checkers.

### 4.2 Examples of Executions

EMMA distribution contains a script for visualization of executions. These visualizations can be used for better understanding of functionality of EMMA and for development and improvement of verification strategies.

Figure 3 shows a diagrams of a executions of two strategies over two models (more examples of visualizations are given on the tool web page). Firewire link [24] is a model of a communication protocol with many reachable properties[1], Szymanski protocol [1] is a mutual exclusion protocol with several versions, most of which are correct, i.e., specified properties are not reachable.

Strategy A consists of a large number of 'error detection' techniques with a short timeout. Strategy B consists of just one technique – simple depth first search with long timeout. On a Firewire model, which has many reachable goals, strategy A is much more successful: from 60 verification problems it can provide answer to 58 of them, whereas strategy B can answer only 45 of them; strategy A is moreover faster. On the other hand, on a Szymanski protocol, for which it is necessary to traverse the whole state space, the simple strategy B is more

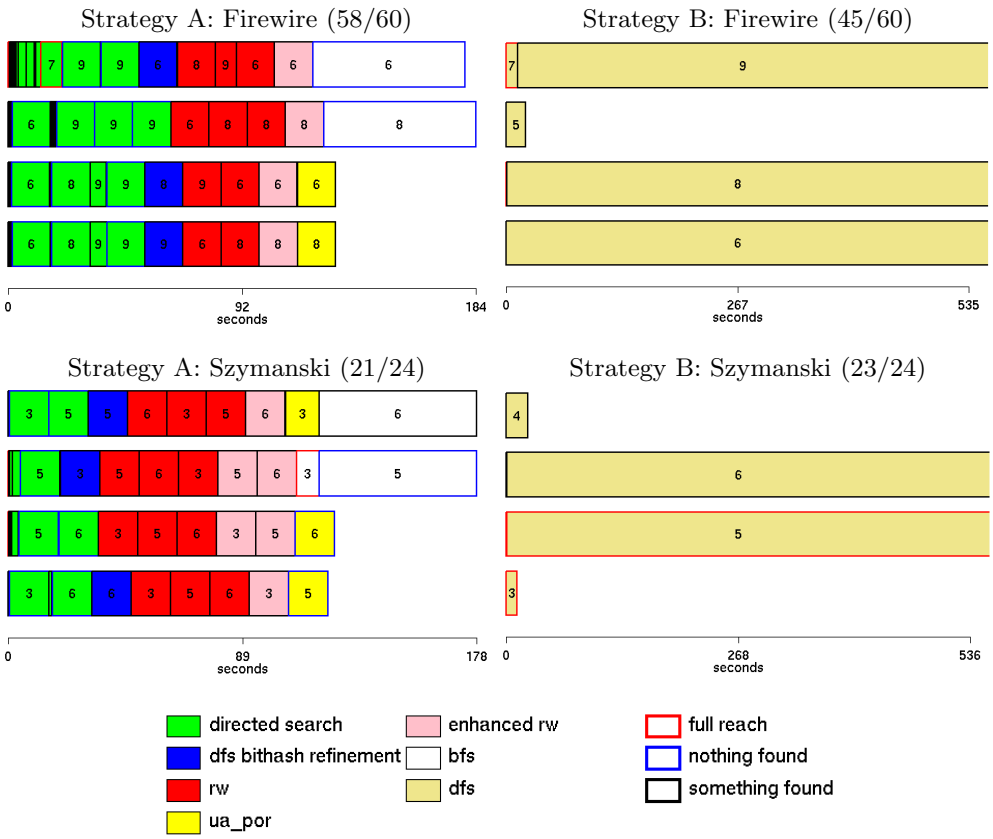



**Fig. 3.** Illustration of EMMA executions on 4 workstations for two models and two strategies. Each line corresponds to one workstation; numbers in boxes are identifications of model instances.

successful: from 24 verification problems it can answer 23 of them, whereas strategy A can answer only 21 of them.

These examples illustrate a more general issue with evaluation of a verification manager. How should we select input problems for evaluation?

### 4.3 Selection of Input Problems

Selection of input problems is an important, but often neglected issue in experimental evaluation. This issue is important even for evaluation of several techniques of the same type – see for example [20] for discussion of the influence of model selection (toy versus complex models) in the evaluation of error detection techniques. Selection of input problems becomes even more crucial issue when evaluating verification meta-search.

[1] Most of these properties are not errors, but just protocol configurations for which we want to check reachability.

By the selection of input data we determine to large extend the results that we obtain from experiments. When we use mainly models without errors, strategies which focus on verification are more successful than strategies tuned for finding errors. When we use models with many easy-to-find errors, there are negligible differences among strategies and we can be tempted to conclude that the choice of strategy does not matter. When we use models with just few hard-to-find errors, there are significant differences among strategies; the success of individual strategies is, however, dependent very much on a choice of particular models and errors. The preceding statements are not just speculations, they are observations based on our experiences with EMMA. By suitable selection of input problems we could "demonstrate" (even using quite large set of inputs) both that "verification manager brings significant improvement" and "verification manager is rather useless".

So what are the 'correct' input problems? The ideal case, in our opinion, is to use a large number of realistic case studies from an application domain of interest; moreover, these case studies should be used not just in their final correct versions, but also in developmental version with errors. However, this ideal is not realizable at this moment – although there is already a large number of available case studies in the domain of explicit model checking, developmental versions of these case studies are not publicly available.

The employment of verification manager could help to overcome this problem. The long-term log can be used to archive all models and properties for which verification was performed (of course with user's content). Data collected in this way can be latter used for evaluation.

### 4.4 Comparison of Strategies

We have performed comparison of different strategies by running EMMA over (different selections of) models from BEEM [16] (probably the largest collection of models for explicit model checkers). Due to the above described bias caused by selection of models, we do not provide numerical evaluation, but only general observations:

- For models with many errors, it is better to use strategy which employs several different (incomplete) techniques.
- For models, which satisfy given property, it is better to use strategy which calls just one simple state space traversal technique with a large timeout.
- If two strategies are comprised of same techniques (with just different priorities, timeouts), there can be a noticeable difference among them, but this difference is usually less than order of magnitude. Note that differences among individual verification techniques are often larger than order of magnitude [20].

Suitable verification strategy depends on the application domain and also on the "phase of verification" – different strategies are suitable for early debugging of a model and for final verification. Thus the expected usage of tool like EMMA is the following:

– Expert user does a bit of experimenting over particular application domain and writes few strategies and hints on how to use them.
– These strategies can be then easily used by other users.

## 5 Conclusions and Future Work

We argue that although model checking technique is automatic, the verification process with the use of model checker is rather complicated and involves significant human expertise. In order to automatize this process, we propose a concept of a verification manager. We also introduce a realization of this concept for explicit model checking (EMMA tool). Experience show that:

– The manager significantly enhances the usability of a model checker, even for an experienced user (particularly for parametrized models with several properties).
– The performance of the different verification strategies depends on selected input problems. It is difficult to perform fair evaluation of this kind of tool.
– We should not expect to find a universal strategy, rather we should look for different strategies for different application domains and verification phases. It should, however, be sufficient to have just few strategies, so that it is easy to select a suitable one even for an inexperienced user.

In order to evaluate a verification manager in a more quantitative way, it is necessary to collect a benchmark of developmental versions of models with realistic and hard-to-find errors (current benchmarks contain mainly correct final version of models). A proposed 'long-term log' of the verification manager can be an appropriate way to collect such a benchmark.

This is just first step towards automatized execution of verification techniques. There are several important directions for future research:

– For the first prototype we restricted our attention only to detection of safety errors. However, the approach can be directly used also for verification of properties expressed in temporal logic. The approach should be particularly useful for verification of LTL properties, since there are many different algorithms and each is suitable for a different purpose (error detection, verification). We also plan to add other techniques such as state caching, state compression, and partial order reduction.
– The architecture of EMMA is designed in such a way, that it should be possible to employ several model checking tools (with the use automatic translator between specification languages). This extension should improve performance and allow further expansion of applicability (e.g., symbolic techniques).
– The manager could use static analysis and estimators [22] to guide the verification meta-search.
– Using the data from the long-term log, it could be interesting to study ways for automatic construction (improvement) of verification strategy by machine learning.

– By analysis of result, manager should be able to provide additional information for the user, e.g., which errors are simple (hard) to find.

## References


1. J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distrib. Comput.*, 16(2-3):75–110, 2003.
2. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proc. of Programming Language Design and Implementation (PLDI 2001)*, pages 203–213. ACM Press, 2001.
3. 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`.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
5. Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs. *Concurrency and Computation: Practice and Experience*, 19(3):267–279, 2007.
6. H. Garavel and F. Lang. Svl: A scripting language for compositional verification. In *Proc. of Formal Techniques for Networked and Distributed Systems (FORTE '01)*, pages 377–394. Kluwer, B.V., 2001.
7. G. J. Holzmann and M. H. Smith. Software model checking: extracting verification models from source code. *Softw. Test., Verif. Reliab.*, 11(2):65–79, 2001.
8. G.J. Holzmann and D. Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.
9. G.J. Holzmann, R. Joshi, and A. Groce. Tackling large verification problems with the swarm tool. In *Proc. of Model Checking Software: The SPIN Workshop*, volume 5156 of *LNCS*, pages 134–143. Springer, 2008.
10. G.J. Holzmann and M.H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.
11. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of SPIN workshop*, volume 1680 of *LNCS*. Springer, 1999.
12. H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 159–173. Springer, 2004.
13. D. Owen, D. Desovski, and B. Cukic. Effectively combining software verification strategies: Understanding different assumptions. In *Proc. of International Symposium on Software Reliability Engineering (ISSRE '06)*, pages 321–330. IEEE Computer Society, 2006.
14. D.R. Owen. *Combining Complementary Formal Verification Strategies to Improve Performance and Accuracy*. PhD thesis, West Virginia University, 2007.
15. R. Pelánek. Web portal for benchmarking explicit model checkers. Technical Report FIMU-RS-2006-03, Masaryk University Brno, 2006.
16. R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

17. R. Pelánek. Fighting state space explosion: Review and evaluation. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'08)*, 2008. To appear.
18. R. Pelánek. Model classifications and automated verification. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'07)*, volume 4916 of *LNCS*, pages 149–163. Springer, 2008.
19. R. Pelánek. Properties of state spaces and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(5):443–454, 2008.
20. R. Pelánek, V. Rosecký, and P. Moravec. Complementarity of error detection techniques. In *Proc. of Parallel and Distributed Methods in verifiCation (PDMC)*, 2008.
21. R. Pelánek, V. Rosecký, and J. Šeděnka. Evaluation of state caching and state compression techniques. Technical Report FIMU-RS-2008-02, Masaryk University Brno, 2008.
22. R. Pelánek and P. Šimeček. Estimating state space parameters. Technical Report FIMU-RS-2008-01, Masaryk University Brno, 2008.
23. D. Sahoo, J. Jain, S. K. Iyer, D. Dill, and E. A. Emerson. Predictive reachability using a sample-based approach. In *Proc. of Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 388–392. Springer, 2005.
24. M. Sighireanu and R. Mateescu. Verification of the link layer protocol of the ieee-1394 serial bus (firewire). *Software Tools for Technology Transfer (STTT)*, 2(1):68–88, November 1998.
25. U. Stern and D. L. Dill. Parallelizing the Mur$\varphi$ verifier. In *Proc. of Computer Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.