

Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the [ENTCS Macro Home Page](#).

Complementarity of Error Detection Techniques

Radek Pelánek¹ Václav Rosecký¹ Pavel Moravec²

*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:

- (i) Rather than focusing on “universal” techniques (suitable for both verification and falsification), it is better to develop specialised techniques and run them

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.

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*

```

1 proc ErrorDetection( $M, \varphi$ )
2   insert initial state to Wait
3   while not finished do
4     get  $s$  from Wait
5     if  $s$  violates  $\varphi$  then return path to  $s$  fi
6     foreach  $s' \in$  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.

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 stack³),
- 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]).

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]).

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.

- 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 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:

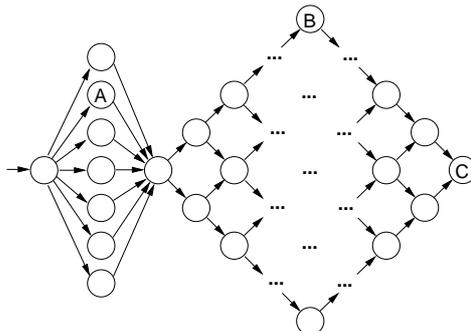


Fig. 2. An artificial example, the right part of the graph is comprised of a ‘big diamond’.

- 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.

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.

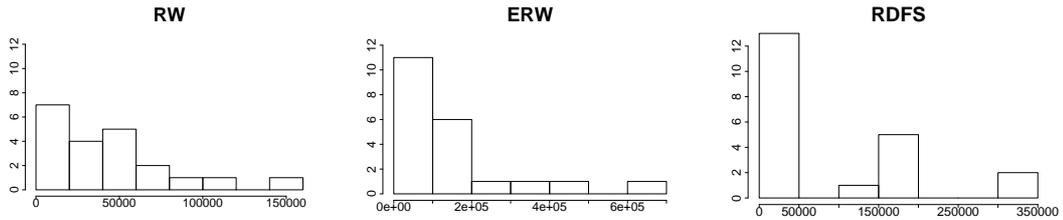


Fig. 3. Histograms showing the number of processed states for 21 runs of three randomized techniques over the model `firewire_link.x1` (x-axis: states, y-axis: number of cases).

3.4 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.

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:

Class 1	$N_B = N_T$
Class 2	$N_B < N_T \leq 2 \cdot N_B$
Class 3	$2 \cdot N_B < N_T \leq 10 \cdot N_B$
Class 4	$10 \cdot N_B < N_T$

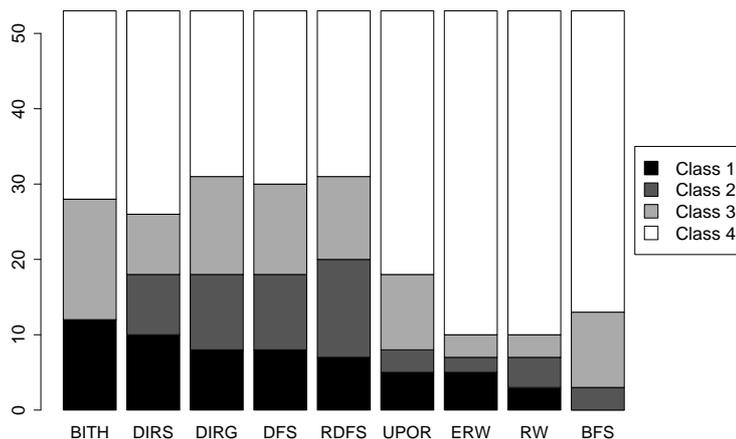


Fig. 4. Comparison of techniques for error detection.

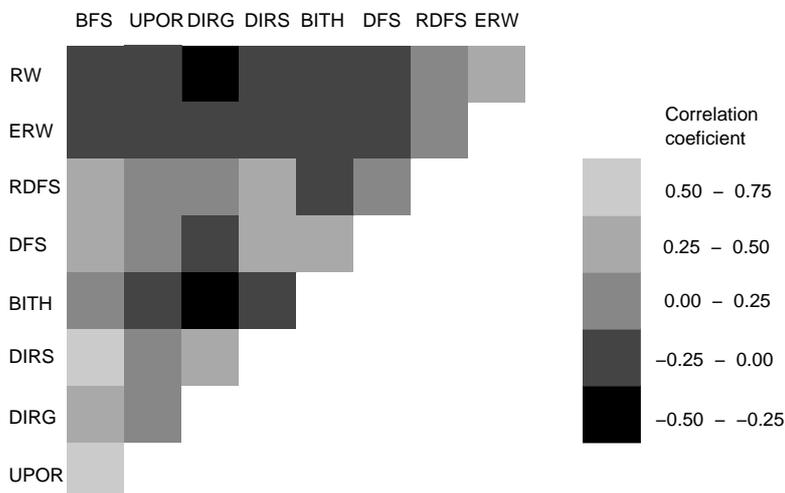


Fig. 5. Correlations among techniques. A light color means high correlation (techniques work/fail on same verification problems), a dark color means low correlation (techniques work/fail on different verification problems).

Fig. 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

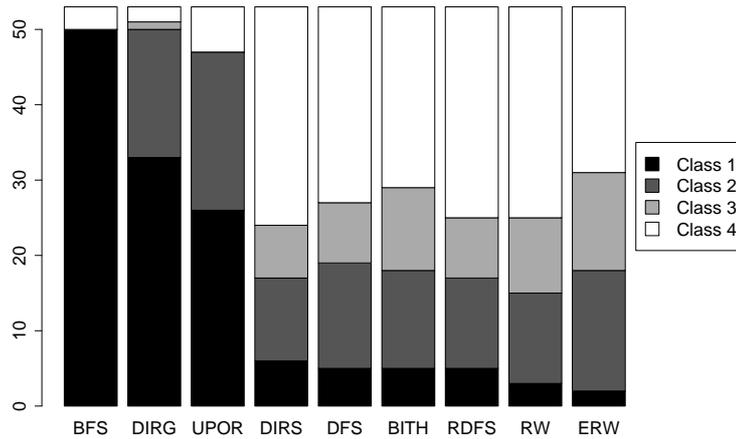


Fig. 6. Lengths of counterexamples produced by techniques. The graph uses the same type of classification as for number of visited states.

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, 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.

Condition coverage

Multiple condition coverage

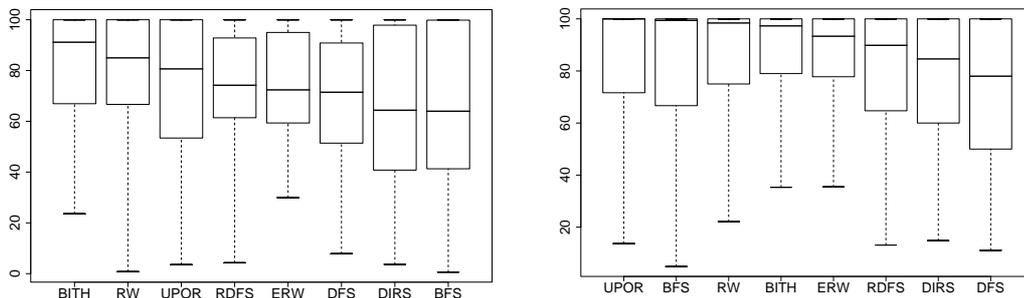


Fig. 7. Comparison of techniques for two coverage metrics; the results are normalized by the best technique (best = 100); results are shown using the boxplot method (minimum, 1st quartile, median, 3rd quartile, maximum) and sorted by the median.

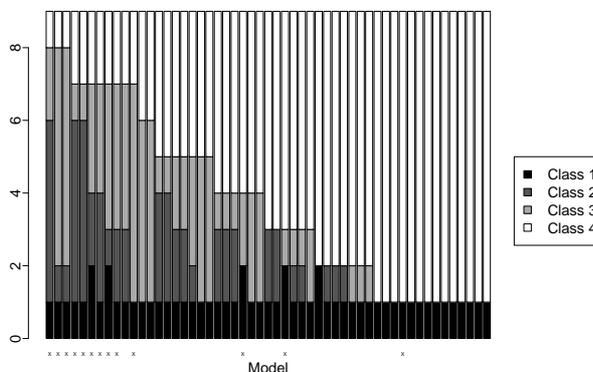


Fig. 8. The performance of techniques over individual models. Each column corresponds to one model and shows the number of techniques in each class. The marked columns correspond to toy models.

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.

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

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

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 cases⁴:

- 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.

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.

- [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 *LNCIS*, 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-Universität Freiburg, 2003.
- [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. Beam: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCIS*, 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 *LNCIS*, 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.