An Evaluation of Model Checkers for Specification Based Test Case Generation

Gordon Fraser*
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2
A-8010 Graz, Austria
fraser@ist.tugraz.at

Angelo Gargantini
Dip. di Ing. dell'Informazione e Metodi Mat.
University of Bergamo
Viale Marconi 5
24044 Dalmine, Italia
angelo.gargantini@unibg.it

Abstract

Under certain constraints the test case generation problem can be represented as a model checking problem, thus enabling the use of powerful model checking tools to perform the test case generation automatically. There are, however, several different model checking techniques, and to date there is little evidence and comparison on which of these techniques is best suited for test case generation. This paper presents the results of an evaluation of several different model checkers on a set of realistic formal specifications given in the SCR [21] notation. For each specification test cases are generated for a set of coverage criteria with each of the model checkers using different configurations. The evaluation shows that the best suited model checking technique and optimization very much depend on the specification that is used to generate test cases. However, from the experiments we can draw general conclusions about which optimizations are useful and which model checking technique is best suited for which type of model. Finally, we demonstrate that by combining several model checking techniques it is possible to significantly speed up test case generation and also achieve full test coverage for cases where none of the techniques by itself would succeed.

1. Introduction

Software testing is an essential part of every software development process. Due to its complexity and incompleteness automation is desirable. Several different techniques to automatically derive test cases have been presented in the past. One particular category of such approaches uses dedicated test models or formal specifications to systematically derive test cases. In this context, the formal verification technique model checking has been proposed as a versatile technique to derive test cases and can adapt to many different requirements of the testing process.

The term model checking describes the process of verifying an automaton model against a temporal logic specification; several different techniques to perform this verification have been presented in the past. The techniques differ in the underlying data structures and algorithms, and optimizations have been presented for all of them. To date it is not clear which of these techniques is best suited when using model checking to derive test cases, which is usually done by formalizing test objectives as temporal logic properties and deriving counterexamples as test cases for these properties. In fact there are critical voices that claim that model checking is not suitable for test case generation at all. However, there is little empirical evidence on this topic and very little comparison of the different techniques.

This paper aims to fill this gap and reports on experiments to compare and evaluate different model checking techniques with regard to their suitability for test case generation. Previous work on this subject was usually based on a translation from a formal language to only one model checker, and that translation was seldom automated. In contrast, we are able for the first time to automatically translate the same specifications to five different model checkers, and this has allowed us to perform a novel comparison of the techniques implemented by these model checkers.

The evaluation is based on a set of real-life specifications given in the requirements specification language SCR (software cost reduction method [21]). Several restrictions apply to this type of testing: Specifications are assumed to be deterministic, and models are assumed to be fully observable. The specifications are automatically translated to the input languages of several different available model checkers. After the conversion, test cases are generated for each specification/model checker pair for a set of different coverage

*The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

criteria using different configurations of the model checker; each coverage criterion can be presented as a set of temporal logic predicates such that counterexample sequences to these predicates represent test cases. Furthermore, random test cases are generated in order to allow a comparison between random testing and test case generation based on model checking. In detail, the analysis tries to answer the following questions:

- Which model checking technique is best suited for test case generation?
- Which optimizations are useful in the context of test case generation?
- What impact does the choice of model checking technique have on the test case generation?

This paper is organized as follows: Section 2 gives a brief overview on model checking techniques and describes how the test case generation problem can be represented as a model checking problem. Section 3 describes the tools, specifications, and coverage criteria used in the evaluation. Then, Section 4 presents the results of our experiments and Section 5 discusses and analyzes the results. Finally, Section 6 draws some general conclusions.

2. Background

In general, model checking describes the process of determining whether an automaton model satisfies or violates a specification given as temporal logic properties. The most common logics are the linear time logic LTL [30] (Linear Temporal Logic), and the branching time logic CTL [12] (Computation Tree Logic). One of the most useful aspects of this process is that model checkers can create witnesses or counterexamples that illustrate how a property is satisfied or violated, respectively. This particular feature is exploited for software testing, as counterexample sequences can be interpreted as test cases under certain constraints (e.g., determinism and observability).

2.1. Model checking

Historically, the first successful model checking approach was *explicit model checking*, which performs an explicit search in a model's state space, considering one state at a time. The search might be based on a breadth-first search (BFS), depth-first search (DFS) or possibly also heuristic search algorithm. There are several different approaches based on different temporal logics [13]. The state explosion problem limits the applicability of exhaustive explicit model checking, but when the goal is not verification but generation of counterexamples then explicit model checking is useful even for large models. *Symbolic model checking* [28] uses ordered binary decision diagrams (BDDs [7]) to represent sets of states and function relations on these states efficiently, which allows the representation of significantly larger state spaces. Finally, *bounded model checking* [6] (BMC) reformulates the model checking problem as a propositional satisfiability (SAT) problem, which contains the unfolded transition relation and the negation of a property up to a certain bound. If this problem is solvable then any solution is a counterexample; the main intention of bounded model checking is not exhaustive verification but fast generation of counterexamples.

Due to space restrictions it is not possible to give detailed overviews of the various model checking techniques; instead, we refer the interested reader to the standard literature on the topic, e.g., [6, 13, 28].

2.2. Testing with model checkers

In order to use model checkers to generate test cases, the test objective (e.g., satisfaction of a coverage criterion) has to be encoded in temporal logic. Each distinct test requirement (e.g., coverage item) of the overall objective is encoded as a temporal logic property (*trap property* or *test predicate*) [19, 25, 31]. When using counterexamples these properties claim that the test requirements cannot be satisfied; consequently, any counterexample to a test predicate represents a test case that satisfies the test requirement posed by the property. Other test objectives include, for example, mutation testing [1] or combinatorial testing [8, 27]. The charm of testing with model checkers is that once a framework has been created it is very easy to apply any of these techniques, or combine several at the same time.

Counterexamples returned by model checkers are sequences of states. In this paper, we assume that a test case is also a sequence of states, and each state represents input and output values serving as test data and test oracle: For each state, test data is provided as input to the system under test and the returned outputs are compared to the expected output values. The length of a test case is defined as the number of states it consists of. For example, this type of test cases is used for reactive systems, such as those specified in SCR notation. Note that counterexamples can also have other interpretations; for example, in [25] counterexamples are control flow paths.

For a detailed overview of testing with model checkers we refer to [18]. The evaluation presented in this paper uses coverage based techniques: Each coverage item represented by a given coverage criterion is encoded in temporal logic, such that a counterexample derived for the property is a test case for the underlying coverage item.

3. Experimental setup

In order to evaluate different model checking techniques and optimizations we use different model checkers in various configurations to create test cases for various specifications and coverage criteria; this section contains all relevant details about the experiments.

3.1. Model checkers

The following freely available model checkers were used in our evaluation:

Spin: Spin [24] (Simple Promela Interpreter) is an explicit state model checker which supports a wide range of options. In particular, we are interested in the main supported search strategies depth-first search (DFS) and breadth-first search (BFS). Spin supports various techniques of compression which aim to reduce the amount of memory needed for storing states. We consider bitstate hashing (BH), introduced by Holzmann [22], which is an optimization technique that can greatly reduce the memory requirements of a model checker by probabilistically storing the visited states. In addition, we are interested in Wolper's hash-compact (HC) method [33], and the lossless collapse compression (CC) [23]. In all runs partial order reduction is disabled as the specifications do not have several agents; weak-fairness is disabled, and an optimization for the case where no cycle detection is needed is enabled. Finally, Spin also supports generation of random paths, which allows us to perform a 'sanity check' whether test case generation with model checkers outperforms random test cases. With random simulation, we force Spin to randomly choose with a pseudo-uniform distribution a monitored variable and then to choose a legal change of that variable (in SCR one monitored variable changes at every step and such a change is called *input event*). In this way, even if the system admits more changes of a variable x than another variable y, the probabilities of a change of x or y at every step do not differ. We have generated random paths of length up to 10 millions states.

NuSMV: NuSMV [10], based on ideas of the classical SMV [26] (Symbolic Model Verifier) developed by Ken McMillan at CMU, is a symbolic model checker which supports both BDD based symbolic model checking as well as SAT based bounded model checking. For BDD based model checking NuSMV supports both CTL and LTL properties; NuSMV is able to analyze LTL specifications with the BDD-based algorithm by using the approach presented in [11]: For each LTL property, a tableau of the behaviors falsifying the property is constructed, and then synchronously composed with the model. Several SAT solvers are supported (MiniSAT [16], ZChaff [29], and the SIM solver developed by the NuSMV team). NuSMV also supports many options; in particular, we are interested in the following optimization techniques: COI [3] (cone of influence reduction), which results in considering only the variables which the property to be proved depends on. Furthermore, NuSMV provides a simplified algorithm (AG) for cases when the property to be proved is a simple invariant and does not contain temporal operators, which is the case for some of the test predicates used in the evaluation.

Cadence SMV: Cadence SMV[1] is also related to SMV [26]. In our evaluation we use the BDD based symbolic model checking technique provided by Cadence SMV, in order to compare different symbolic model checkers.

SAL: SAL [14] (Symbolic Analysis Laboratory), is a model checker implemented in the language Scheme, and supports both BDD based model checking and bounded model checking. It uses the same BDD library as NuSMV (CUDD [32]) but it does not use the same direct tableau construction. The SAL bounded model checker is based on the Yices SAT solver, and it can also perform a unique "infinite" model checking based on SMT solving.

In our experience, different model checkers can behave very differently even if they are based on the same principles. For example, at times NuSMV might run out of memory when SAL has no problems, or vice versa. In addition, using several model checkers that implement similar techniques should allow us to gain insights on the effects of certain optimizations, implementations, and data types.

3.2. Specifications

In order to increase the confidence in the generality of the results we obtain from the experiments, the evaluation uses several specifications with different characteristics. Table 1 lists statistics of the formal specifications used in the evaluation: the number of monitored variables (Mon.) and the number of dependent variables (Dep.), i.e., internal variables and outputs, and the number of possible input combinations (Input Comb.) and states (Total States). The specifications are given in SCR (Software Cost Reduction method [21]) notation, and for the experiments they are automatically translated to the input languages of the model checkers. The 'cruise' specification models a simple automotive cruise control [2], 'sis' models a Safety Injection System which controls the coolant injection of a nuclear power plant [21], the 'autopilot' specifies the requirements of a simplified mode control panel for the Boeing 737 autopilot [4], 'bombrel' is a simplified subset of the bomb release requirements of a U.S. Navy attack aircraft [5], and 'car3prop' is the complete specification of a "car overtaking" protocol, intended to coordinate intelligent vehicles on a road [17]. Autopilot, bombrel, and sis are examples of classical reactive systems which monitor some integer inputs and control few critical outputs: in order to adequately test them, one should choose the right values of the inputs in possible big intervals. Cruise and car3prop do not have integer inputs and the internal logic of the controller is the most critical part to be tested. For this reason, these two specifications do not have disequality split (DS) and boundary (B) test predicates, while they present a relatively high number of test predicates for MCDC (see Table 2 and Section 3.3).

[1] http://www.kenmcmil.com/smv.html

Table 1. Specification statistics.

Name	Variables		Size	
	Mon.	Dep.	Input Comb.	Total States
autopilot	10	9	2.82e+11	4.78e+18
bombrel	9	3	8.28e+07	9.94e+08
car3prop	5	12	1.23e+05	1.01e+13
cruise	4	1	3.20e+01	1.28e+02
sis	3	3	2.00e+04	2.40e+05

Table 2. Test predicates.

Name	T	SM	MCDC	DS	B	Σ
autopilot	44	99	178	32	2	355
bombrel	13	21	39	4	2	79
car3prop	53	171	289	0	0	513
cruise	12	15	51	0	0	78
sis	13	21	41	8	8	91

3.3. Coverage criteria

For each of the SCR specifications several sets of test predicates are automatically generated according to different coverage criteria. SCR specifications consist of different types of tables which in turn consist of cells that contain logical expression over modes. In SCR, a mode is an internal term of a specification that captures the system history: Each mode defines an equivalence class of system states useful both in specifying and in understanding the required system behavior. We used the following coverage criteria in our experiments:

- Table coverage (T): Every cell is covered once.
- Split mode coverage (SM): If a cell refers to several modes, it is covered for every mode.
- Disequality split (DS): Expressions containing disequality operators are covered for both cases of the disequality. For example, $\geq$ is split into $>$ and $=$.
- Boundary coverage (B): Cover boundary values of disequalities in expressions. For example, $y > C$ (with x and C integers) is split into $x = C + 1$ and $x > C + 1$.
- Modified Condition Decision Coverage (MCDC): Each literal (condition) is shown to independently affect the value of the expression (decision) it is part of. We use masking MCDC [9].

Table 2 summarizes the test predicates generated for each of the specifications and coverage criteria. To keep the number of tables in the paper to a tractable number each experiment uses the union of a specification's test predicates.

4. Results

Tables 3-5 contain some data about the experiments we performed[2]. For space reasons we can only report a small fraction of the data we have, since for every case study we ran around 45 different model checkers and options; the presented data was selected with respect to the discussion in the next section. Tables report the best run for every model checker and several other runs to give evidence to our conclusions. We ran all the experiments on a Linux PC with Intel(R) Xeon(R) CPU at 2.66GHz and 4 GB of RAM. We set the timeout for every call of the model checker to one hour. To evaluate the memory consumption we use the memusage command available on Linux systems, which unfortunately in some cases did not report reliable results.

Each table lists results for one or more specifications: First, it contains the model checker and options used. Options are listed in terms of their abbreviations and acronyms as introduced above. In addition, values of the type 'dx' represent a search depth of x for bounded model checking, 'mx' means a search depth of x for Spin[3] , and 'wx' specifies the size of the hashtable for Spin. For random test generation the option represents the number of states (length) that each random test case should have.

The number of runs shows how often the model checker was called until either all test predicates were covered, a timeout was reached, memory was exhausted, or there were no more test predicates that the model checker had not been called on. Consequently, each run can either successfully generate a test case, return with an error (e.g., timeout reached or no counterexample found within boundary), or determine that a test predicate is infeasible, i.e., there exists no test case to cover the test predicate. If a test case is generated, all remaining test predicates that are also covered by that test case are removed, and so only uncovered test cases are considered. The tables list the number of unique test cases, which means that if a test case is subsumed by another test case (e.g., if it is a prefix of the other test case) only the subsuming test case is considered.

In addition, the tables list the total time the model checker took for all its runs and the maximum amount of memory used at any time. Finally, the sum of states gives the total length of the test suite, i.e., the sum of the lengths of each test case, and max states gives the length of the longest test case which has been generated. Note that the longest test case might not necessarily add to the coverage, therefore the total number of states in a test suite can be smaller than the length of the longest generated test case (e.g., see experiments 2 and 3).

5. Discussion

From the data we have we can draw two kinds of comments: The first kind concerns the use of specific options of the same model checker and the second kind concerns comparisons between the different model checkers.

[2]For the entire set of results see http://cs.unibg.it/gargantini/software

[3]mnM (mnK) means a search depth m of $n \cdot 10^6$ ($n \cdot 10^3$)

Table 3. Sis (91 test predicates).

exp	Model Checker	Runs	Tests	Test predicates		Time	Mem	States	
				Error	Infeasible	(Total)	(Max)	Sum	Max
1	Random 1000	66	1	66	0	14.39	0.124	1003	1003
2	Random 100K	41	2	37	0	67.28	0.120	43237	100003
3	Random 10M	5	1	49	0	20.65	0.121	18151	3170983
4	Spin DFS m1K w20	68	1	66	0	41.91	17.3	7	7
5	Spin DFS m10M w20	18	5	8	0	147.75	754	112881	23515
6	Spin DFS m1M w20	17	5	8	0	141.79	410	112881	23515
7	Spin DFS m1M w25	13	6	0	7	106.21	627	117761	23515
8	Spin CC m1M w25	19	6	7	0	166.87	662	117761	23515
9	Spin HC m1M w25	17	6	7	0	111.55	561	117761	23515
10	Spin BH m10M w24	19	6	7	0	85.74	384	117898	23123
11	Spin BFS m1M w20	46	9	34	0	440.5	705	556	92
12	NuSMV CTL	23	14	0	7	280.85	119	2563	402
13	NuSMV LTL	21	11	0	7	266.58	211	2499	406
14	NuSMV LTL COI	18	10	0	7	222.15	249	2092	406
15	NuSMV LTL AG COI	67	2	47	0	40.21	64.6	403	400
16	NuSMV/BMC MiniSAT COI	69	2	67	0	2170.06	456	6	3
17	NuSMV/BMC ZChaff COI	70	3	66	0	1188.03	517	8	3
18	Cad. SMV CTL	25	15	0	7	345.57	-	2655	402
19	Cad. SMV LTL	28	15	0	7	562.28	-	2964	402
20	SAL/SMC	26	12	0	7	68.53	-	2393	403
21	SAL/BMC d2	73	2	67	3	19.09	-	6	3
22	SAL/BMC d10	70	4	63	3	23.99	-	28	11
23	SAL/BMC d50	68	2	63	3	109.05	-	11	6

Table 4. Cruise (78 test predicates).

exp	Model Checker	Runs	Tests	Test predicates		Time	Mem	States	
				Error	Infeasible	(Total)	(Max)	Sum	Max
24	Random 1M	9	1	6	0	127.74	0.0949	16349	16349
25	Spin DFS w20	17	9	0	6	9.52	8.58	315	39
26	Spin BFS m1M w20	21	13	0	6	11.81	8.29	67	7
27	Spin BH m1M w19	15	9	6	0	8.1	38.2	315	39
28	Spin BH BFS	22	13	6	0	12.13	2.29	67	7
29	Spin HC m1M w20	17	9	6	0	6.78	46.3	315	39
30	NuSMV LTL COI	17	11	0	6	2.7	19.4	90	10
31	Cad. SMV LTL	20	13	0	6	1.73	-	65	7
32	SAL/SMC	22	12	0	6	8.67	-	74	8
33	SAL/BMC d10	16	7	6	0	4.99	-	68	11

Table 5. Generation statistics.

exp	Model Checker	Runs	Tests	Test predicates		Time	Mem	States	
				Error	Infeasible	(Total)	(Max)	Sum	Max
bombrel (79 test predicates)									
34	Random 1M	4	1	7	0	1.18	0.117	2342	2342
35	Spin BH BFS m100M w24	26	9	15	0	108.24	245	53	8
36	NuSMV LTL COI	11	9	0	1	80.59	114	180	60
37	Cad. SMV CTL	52	5	48	1	18.49	-	76	58
38	SAL/BMC d10	37	4	21	1	10.27	-	40	11
autopilot (355 test predicates)									
39	Random 100K	6	1	3	0	42.9	0.176	19084	19084
40	Spin DFS m1M w31	43	23	4	0	2577.3	294	574997	25001
41	NuSMV/BMC ZChaff COI	83	32	44	0	83.31	720	158	8
42	NuSMV/LTL COI	355	0	352	3	10758.82	270	-	-
43	Cad. SMV CTL	61	39	9	3	158.1	-	197	8
44	SAL/SMC	81	30	23	3	741.38	-	170	9
car3prop (513 test predicates)									
45	Random 1K	233	5	231	0	123.78	0.419	5015	1003
46	Spin BH m10M w24	309	3	301	0	883.05	384	32022	11711
47	Spin DFS m1M w31	266	6	258	0	58844.82	294	89585	20409
48	NuSMV LTL COI	447	6	436	0	55.8	12.9	12	2
49	Cad. SMV LTL	20	9	371	5	8.28	-	25	4
50	SAL/BMC	513	0	513	0	69.72	-	0	0

5.1. Model checker options and optimizations

Regarding the comparison among the different options for the same model checker or technique, we can say that:

Random testing: Increasing the length of the randomly generated tests normally has increased the number of test predicates covered (see exp. 1 and 2), although there is no guarantee for this. As our tool that calls the model checkers was sometimes not able to deal with test sequences as long as several millions states, the random tests with the maximum length of 10 million states actually often covered fewer test predicates (see exp 2 and 3). We found that for big specifications the randomly generated test suite which covered most test predicates it was never that with the maximum length (see exp. 34, 39, 45). In fact, the best random test suite for the two biggest specifications (see exp. 39, 45) has maximum 100K states. Moreover, we believe that test sequences consisting of millions of states, although achieving high coverage, may result useless in the end in both conformance testing and in validation (i.e., to check by hand if the model behaves as expected). In conformance testing, the execution of so long tests may require too much time on the real system and so long tests may be difficult to understand during validation. Consequently, it is necessary to find a balance between a very high maximum length to possibly cover more test predicates and a small length to keep the tests still useful and tractable by the involved tools.

Spin options: Spin supports a great variety of options and search strategies. This can be a strong advantage for the user, since he/she can choose the strategy which suits his/her goal best, but it can be really confusing at the same time too. Even for small examples like sis, using Spin with no options or with a small search depth (-m option) caused Spin to neither find all the tests nor to complete the state search and therefore it was unable to prove infeasible test predicates (see exp. 4). On the other hand, very large values for *m* and *w* caused Spin to not complete the task in the same cases (exp 5). The choice of the right options is not easy and a heuristic would be very useful.

State compression techniques: Spin state compression techniques like the lossless collapse compression (CC) and the lossy hash compact (HC) were not as effective as expected (for example, compare exp. 7, 8, and 9). They generally reduced the memory consumption but increased the time required and were unable to prove infeasible test predicates: they never outperformed the standard DFS search in terms of total time to get the complete test suite (even excluding infeasible tests).

The Spin approximation technique bitstate hashing (BH) generally performed better than the equivalent (in `m` and `w`) non approximated techniques (see exp. 10 and exp. 35) regarding the time and memory required to complete the task. However, as it is not able to prove that a test predicate is in-

feasible the user does not know if a possible error is due to a too small search depth (in this case it can be increased) or because the test predicate is infeasible (in this case the BH approximation is hopeless). Moreover, sometimes precise representations of the state space prove to be more effective (see exp 46 and 47). A good policy could be to run an approximation algorithm first and then a complete search for only those test predicates which are left uncovered.

BFS vs. DFS: Spin BFS performed slightly worse than the standard DFS in terms of the final results: often BFS failed to complete the task where the DFS succeeded (see exp 6,7 and 11). However, the tests generated by the BFS are much shorter than the tests generated by the DFS. On the other hand, this caused a greater number of runs of the model checker and then a much longer time to obtain the complete test suite. BFS can be useful when the user prefers short tests, for example for system validation.

LTL vs. CTL: As expected the LTL symbolic model checker was slightly slower than the CTL model checker, since it must transform the LTL properties in equivalent tableaus. However, there is no clear winner in terms of required memory and number of test predicates solved.

COI abstraction: The cone of influence abstraction (COI) of NuSMV always improved the test generation process by lowering the memory and the time required to complete the task. The use of this option is strongly recommended. For the biggest specification we were unable to complete the generation without COI abstraction.

AG only search: Although the AG option improved the generation of some test cases, it is not usable for all the test predicates, since SCR specifications contain two state trap properties which refer to the current state and the next one; for these properties the simplified algorithm of AG is not suitable. For this reason, the number of errors increased when the AG variant was applied (see exp 14 and 15).

Bounded model checking: The bounded model checkers never completed the test suite generation, even when excluding infeasible test predicates. Moreover, only SAL/BMC is able to prove infeasible test predicates by induction. In general, using the BMC results in short test cases, which, on the other hand, usually increases the total number of test cases. With regard to the SAT solvers, we found that ZChaff was always faster than MiniSAT, but required much more memory (exp 16 and 17). With the default SAT solver (SIM) NuSMV was not able to complete even simple tests with a small depth. Increasing the search depth in BMC from 2 to 10 decreased the number of failed test case generation runs, while increasing the depth to 50 did not improve the final result (see 21, 22, and 23). In general, increasing the depth should increase the number of test cases covered at the expense of a greater time and memory consumption. In our specifications, this did not happen: they have the same number of test predicates which can be covered by test sequences of length up to 10 and 50.

5.2. Comparison of model checking techniques

The second kind of comments regards the comparison of the different model checkers to see whether there is a model checker which works clearly better than the others when generating test cases. We found some generic conclusions:

Random testing: Random generation worked better than expected, especially on the biggest specification. It slightly outperformed the best model checker for two specifications (exp 39 and 45) with regard to the time required to cover all test predicates. This means that when dealing with model based test generation, a comparison with random generation is mandatory and that the random generation must be treated as complementary to other more complex techniques, also since it has some disadvantages: in order to achieve high coverage it usually produces very long test sequences, it is unable to prove infeasible test predicates, and it may perform much worse than other guided technique (as in exp 1).

Bounded model checking: Bounded model checking was not as efficient as symbolic model checking in finding tests. However, there may exist specifications with very short tests and very complex test predicates for which BMC may perform better than the other model checkers. This was, for example, reported for a case study in [20].

Symbolic model checking vs. explicit model checking: Also between symbolic model checking and explicit model checking there is no clear winner. However, the explicit model checker seems to be able to deal with at least some test cases in large specifications (see exp 47) since it does not need to search the entire space state in case a violation is found, while the symbolic model checker simply cannot represent the entire state space and fails with fewer test cases covered (see exp 48 and 49). This is an interesting observation, as symbolic model checking was introduced precisely to address the problems of explicit state model checking with regard to large state spaces. The conclusion is that exhaustive verification is not necessary to generate test cases, only for certain difficult test predicates and to prove infeasible test predicates.

NuSMV vs. Cadence SMV: NuSMV and Cadence SMV have comparable performance. However, in some cases (exp 42), NuSMV was not able to complete the task (mainly due to the timeout), while Cadence SMV was.

Table 6 reports the best model checker and its configuration for each of the specifications we have analyzed. With 'best' model checker we mean the fastest to finish among those which made fewest (possibly 0) errors. As proved by Table 6, there is no single model checker that outperforms the others in all the specifications.

Table 7 illustrates for each specification for how many

Table 6. Best model checker.

Spec.	Errors	Best model checker
autopilot	4	Spin DFS m1M w31
bombrel	0	NuSMV LTL COI
car3prop	258	Spin DFS m1M w31
cruise	0	Cadence SMV LTL
sis	0	SAL SMC

Table 7. Test predicates with fastest counterexample generation.

		BDD Based			BMC	
Spec.	Spin	NuSMV	SAL	Cad.SMV	NuSMV	SAL
autopilot	0	0	0	216	147	0
bombrel	0	0	0	78	0	0
car3prop	99	58	0	174	131	0
cruise	0	0	0	72	0	0
sis	1	0	58	0	0	25

Table 8. Test predicates with shortest counterexample.

		BDD Based			BMC	
Spec.	Spin	NuSMV	SAL	Cad.SMV	NuSMV	SAL
autopilot	272	0	81	299	261	86
bombrel	54	27	5	60	51	21
car3prop	253	200	0	125	202	0
cruise	62	0	4	68	62	2
sis	45	130	10	71	24	12

Table 9. Potential time savings.

Spec.	Min	Max	Worst	Possible
autopilot	41.9s	512.6s	2577.3s	8.8s
bombrel	18.1s	80.6s	3431.9s	8.6s
car3prop	-	-	58102.9s	1264.7s
cruise	1.7s	29s	127.7s	1.3s
sis	106.2s	562.3s	1995.1s	14.5s

test predicates each model checker created a counterexample the fastest. This is an approximation, as for each test predicate we used the test case covering the predicate that was created the fastest, and not necessarily the one for which the model checker was called. If two different techniques need the same amount of time to create a test case then the test predicate is counted for both techniques. For specifications that are not too large symbolic model checking is the dominant technique. If the specification is more complex, as for example seen in the car3prop example, then explicit state model checking creates test cases faster. Bounded model checking is very fast in creating test cases up to a certain length, but the performance deteriorates with increasing length, therefore the number of test predicates where the bounded model checker is the fastest is smaller than that of a BDD based model checker. Random generation can generate test cases quicker for the majority of test predicates for all specifications but the cruise example, but there are always certain hard to find test predicates.

Table 8 shows which model checkers performed best with regard to creating shorter test cases. Here, Spin can create very good results when using BFS, but much worse results when using DFS. This table also illustrates that Spin with BFS succeeds for more test predicates than bounded model checking.

5.3. Potential improvements

Theoretically, if one could know in advance for a given test predicate for a given model which model checker to apply, one could save time and get better results (more complete). To assess the potential of this we simulate the following experiments based on our data: Let O be the optimized set of runs, initially empty. Order every run for the time taken for it divided by number of test predicates not yet covered, then add the first run to the set O and discard all the test predicates covered by it; repeat this until all test predicates are covered or there are no more test cases left.

Table 9 lists for each specification the time of the fastest model checker run that created a complete test suite (Min), the time of the worst model checker run that satisfied all test predicates (Max), the time of the worst run (Worst) including runs with errors, and the possible result if one would know ahead of time which model checker to use. The potential optimization is significant in most cases, which suggests that a hybrid approach exploiting several techniques could theoretically be used to improve the test case generation process. In addition, no single technique was able to cover all test predicates for the car3prop example, but combining the techniques would allow the creation of a complete test suite. This is an interesting observation, as it shows that a mix of several techniques may increase the applicability of model checking for test generation in practice.

5.4. Coverage efficiency

While performance is one of the main concerns whenever using model checkers, the question is not only how fast we can generate test cases but also how good these test cases are. This is a difficult question to answer, because intuitively more and longer test cases will generally mean better testing. However, resources for testing are usually

Table 10. Coverage efficiency.

Spec.	Random	Spin	BDD Based			BMC	
			NuSMV	SAL	Cad.SMV	NuSMV	SAL
autopilot	0.003	0.008	0.0	14.2	15.7	16.0	7.3
bombrel	0.001	0.000	1.9	1.5	1.8	4.0	1.0
car3prop	0.005	0.034	19.5	0.0	17.2	18.5	0.0
cruise	0.000	1.289	3.3	3.3	3.8	3.7	1.3
sis	0.000	0.004	0.2	0.17	0.2	6.8	3.4

limited, and so test cases should preferably maximize coverage while being as short as possible. To this extent we define *coverage efficiency* as the ratio between the length of a test case to the number of test predicates covered; the efficiency of a test suite simply is the sum of the efficiency values of each test case (which means the same test predicate can contribute to the efficiency of several test cases).

Table 10 lists these values for the different specifications and model checkers used in our evaluation. Note that the efficiency value depends on the underlying specification, therefore it is only possible to compare values related to one specification. Of all techniques, random testing has the lowest efficiency values. This means that while random testing is a very cheap method with regard to test case generation, it results in expensive (long) test cases. Explicit state model checking using DFS also results in very low values; BFS (not distinguished in the table) results in better values. Bounded model checking and BDD based model checking both result in more efficient test cases, where bounded model checking is slightly better.

6. Conclusions

In this paper we have presented the results of an empirical evaluation of different model checkers for generating test cases. The main concern when using model checkers for this task is the performance, which can be problematic due to the state space explosion. In general it is difficult to predict how a particular model checking technique will perform for any given specification. Consequently it is impossible to give detailed rules about which model checker to use for a given specification.

Our results show that if a specification is too big for symbolic model checkers to handle, then explicit state model checking with DFS can still achieve good results, although it is not guaranteed to result in a test suite that covers all test predicates. Explicit model checking can also potentially be improved by applying heuristic search [15], which was not considered in our evaluation. Even though bounded model checking is very efficient there are some concerns when applying it to big models: The performance of a bounded model checker depends on the length of the counterexamples it should consider, and as large models tend to result in long test cases a bounded model checker is likely to be unable to generate many test cases. This is especially the case if the considered specification contains integer variables. If, however, the state space can be handled by a symbolic model checker, then this is often the preferred method as it can prove infeasible test predicates unlike bounded model checking in general, and returns short counterexamples unlike explicit model checking DFS.

We were surprised to see how well random testing performed in comparison to model checking: As our results show, random testing can very quickly create test cases that cover a large portion of the test predicates. However, random testing has problems covering 'difficult' test predicates, cannot be used to prove infeasible test predicates, and probably most problematic, it creates very long test cases which might not be usable in practice due to limited resources and the difficulty of interpreting failed executions.

A possible idea to benefit from the respective advantages of the different techniques would be to apply a hybrid strategy: First use a cheaper method to cover the 'easy' test predicates, e.g., with random testing, an explicit model checker using depth first search, or a bounded model checker with a low bound. Then, in a second step, use a complete method like symbolic model checking to cover the more difficult test predicates and detect infeasible test predicates. Such an approach can not only increase the speed with which a test suite is generated, but can also make it possible to apply testing with model checkers to larger specifications than by just using a single technique.

References

[1] P. E. Ammann, P. E. Black, and W. Majurski. Using Model Checking to Generate Tests from Specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, 1998.

[2] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *International Symposium on Software Testing and Analysis*. ACM, 1996.

[3] S. Berezin, S. V. A. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 81–102, London, UK, 1998. Springer-Verlag.

[4] R. Bharadwaj and C. Heitmeyer. Applying the SCR requirements method to a simple autopilot. In *In Proc. Fourth NASA Langley Formal Methods Workshop (LFM97), NASA Langley Research*, 1997.

[5] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Autom. Softw. Eng*, 6(1):37–68, 1999.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[7] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[8] A. Calvagna and A. Gargantini. A Logic-Based Approach to Combinatorial Testing with Constraints. In *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 66–83. Springer-Verlag, 2008.

[9] J. Chilenski and L. A. Richey. Definition for a masking form of modified condition decision coverage (MCDC). Technical report, Boeing, 1997.

[10] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A New Symbolic Model Verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.

[11] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. In *Formal Methods in System Design, 10(1):57–71*, February 1997.

[12] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA., 1 edition, 2001. 3rd printing.

[14] L. de Moura, S. Owre, H. Rueß, J. R. N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[15] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed Explicit Model Checking with HSF-SPIN. In *SPIN '01: Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 57–79, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[16] N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[17] P. Ernberg, L. Fredlund, and B. Jonsson. Specification and validation of a simple overtaking protocol using LOTOS. Technical report, Swedish Institute of Computer Science, Kista, Sweden, 1990.

[18] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 2009. To appear.

[19] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests From Requirements Specifications. In *ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687, pages 146–162. Springer, 1999.

[20] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-Generating Test Sequences using Model Checkers: A Case Study. In *Third International International Workshop on Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 42–59. Springer Verlag, October 2003.

[21] C. L. Heitmeyer. Formal methods for specifying, validating, and verifying requirements. *J. UCS*, 13(5):607–618, 2007.

[22] G. J. Holzmann. On Limits and Possibilities of Automated Protocol Analysis. In *Proceedings of the IFIP WG6.1 Seventh International Conference on Protocol Specification, Testing and Verification VII*, pages 339–344, Amsterdam, The Netherlands, 1987. North-Holland Publishing Co.

[23] G. J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *In Proceedings of the Third International SPIN Workshop*, 1997.

[24] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[25] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In *TACAS '02: Proceedings of the 8th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341, London, UK, 2002. Springer-Verlag.

[26] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie-Mellon University, 1992.

[27] D. R. Kuhn and V. Okun. Pseudo-Exhaustive Testing for Software. In *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*, pages 153–158. IEEE Computer Society, 2006.

[28] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[29] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM.

[30] A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA*, pages 46–57. IEEE, 1977.

[31] S. Rayadurgam and M. P. E. Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91, Washington, DC, April 2001. IEEE Computer Society.

[32] F. Somenzi. CUDD: CU Decision Diagram Package Release, 1998.

[33] P. Wolper and D. Leroy. Reliable Hashing without Collosion Detection. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 59–70, London, UK, 1993. Springer-Verlag.