On the Order of Test Goals in Specification-Based Testing

Gordon Fraser [a,*] Angelo Gargantini [b] Franz Wotawa [a]

[a] *Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/2, A-8010 Graz, Austria*

[b] *University of Bergamo, Dipartimento di Ingegneria dell'informazione e metodi matematici viale Marconi 5, 24044 Dalmine BG, Italia*

Abstract

Model-based testing techniques often select test cases according to test goals such as coverage criteria or mutation adequacy. Complex criteria and large models lead to large test suites, and a test case created for one coverage item usually covers several other items as well. This can be problematic if testing is expensive and resources are limited. Therefore, test case generation can be optimized in order to avoid unnecessary test cases and minimize the test generation and execution costs. Because of this optimization the order in which test goals are selected is expected to have an impact on both the performance of the test case generation and the size of resulting test suites, although finding the optimal order is not feasible in general. In this paper we report on experiments to determine the effects of the order in which test goals are selected on performance and the size of resulting test suites, and evaluate different heuristics to select test goals such that the time required to generate test suites as well as their size are minimized. The test case generation approach used for experimentation uses model checkers, and experimentation shows that good results can be achieved with any random ordering, but some improvement is still possible with simple heuristics.

Key words: Automated software testing, automated test case generation, testing with model checkers, performance, minimization, monitoring

* Corresponding author
Email address: fraser@ist.tugraz.at (Gordon Fraser).

[1] 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 Wirtschaftsfrderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

1 Introduction

Software testing remains the most important technique in order to determine whether the quality of a software system is acceptable. As the number of possible test cases is usually infinite, common testing techniques try to select finite representative subsets. For example, coverage criteria can not only be used to evaluate existing test suites, but also to create test suites automatically. A coverage criterion defines a set of test goals that a test suite should exercise – for each of these test goals a test case can be generated. Mutation based approaches are related: A test suite is mutation adequate if there exists a test case for every mutant such that the mutant is killed. This can be interpreted as a test goal for each mutant, stating that the mutant should be killed.

One test case will usually not only cover the single test goal it is created for, but will cover several other test goals as well. For example, test cases are often sequences of states and several different states are passed by a test case, thus covering many test goals. This means that a naive approach in which one test case is generated for each test goal will result in a test suite that contains more test cases than are strictly necessary in order to satisfy the coverage criterion.

With increasing model size there is an increasing number of possible coverage items or mutants. Consequently, test suite sizes quickly increase as well. The size of a test suite, however, is often critical because the resources available for testing are usually limited. An often used solution when this is the case is to select only a subset of a test suite, such that this subset still satisfies a chosen coverage criterion. This approach is known as *test suite minimization* or *test suite reduction*, and is known to greatly reduce the number of test cases in a test suite, while of course sacrificing some of the fault sensitivity by doing so.

Depending on the technique used to derive test cases, creating test suites can be a resource intensive task itself. In this case a lot of effort is wasted if initially a large test suite is created, which then has to be analyzed such that only a subset of the test suite is actually executed. This can be avoided by using a smarter approach to test case generation: Instead of naively generating all test cases at once the test goals can be *monitored* with regard to the test cases generated. For each new test case all those test goals it satisfies can be excluded from the test case generation.

Monitoring can be useful for online and offline testing: Offline testing describes approaches where first a complete test suite is generated, and the test cases are executed once the test suite is complete. Once completely generated the test suite can be optimized with regard to many aspects; for example, it might be minimized according to some coverage criterion. Monitoring can be used to reduce the size and effort of the test suite creation. In contrast, online

testing describes approaches where test case generation and execution are performed at the same time. In this case it is not possible to apply post-creation optimizations. If the test effort shall be reduced, monitoring is the only way to do so.

When monitoring test goals during test case generation, the order in which test goals are selected possibly has an impact on the result: Some test goals might result in long sequences covering many other goals. In the best case a single test case might cover all test goals, while in the worst case one test case is created for each test goal. Finding a good order in which to create test cases has the potential to reduce the number of test cases and the time used to generate them, but finding an optimal order is not feasible in general because there are too many possible orderings. Before conducting the study we expected a large impact of the test goal selection order on the resulting creation time and test suite size. Therefore, we report of experiments to answer the following research questions in this paper:

- What influence does the order of test goal selection have on the creation time and resulting test suite size?
- Which heuristics can be applied to prioritize test goals represented in temporal logic, and what is the achieved improvement?

This paper is an extended version of (Fraser and Wotawa, 2008), presented at the Fourth International Workshop on Advances in Model Based Testing (A-MOST 2008). The original paper describes a set of experiments that lead to the unexpected conclusion that the order can largely be ignored. To validate this result we performed more experiments with other test models and test case generation methods, and provide an in-depth analysis of these experiments. The general conclusion is that monitoring will in practice greatly reduce the test suite size regardless of the ordering, although some further reduction can be achieved by ordering test goals using simple heuristics.

This paper is organized as follows: First, we consider specification-based testing using model checkers in Section 2; all our experiments are performed using such test case generation techniques. This approach allows easy combination of different test techniques, and makes it possible to experiment with different test case generation techniques simply by changing the underlying model checking tool.

One main motivation of our experiments is that the size of a test suite is an important factor, either because test case generation or execution is expensive. Section 3 looks at two different but related approaches to reduce the size of a test suite: Test suite minimization is a widely accepted technique to reduce the size of a test suite; in contrast, test case generation monitoring tries to avoid that more test cases than necessary are generated. This paper considers

the latter technique, as test suite minimization assumes that all test cases are generated before minimization is applied.

The experiments performed in order to analyze the effect of the order in which test goals are chosen are described in Section 4, where the results are also listed. The results show that any random ordering can usually achieve quite good results and will often be sufficient in practice. However, in some scenarios where resources are limited and test case generation is costly even minor gains in performance will be important. Therefore, Section 5 describes and evaluates three simple techniques to sort test goals before generating any test cases. Finally, the paper concludes with a discussion of the results in Section 6.

2 Specification-Based Testing with Model Checkers

One of the main tasks of software testing is the selection of a finite set of test cases out of a possibly infinite number of possible test cases. Many different techniques have been proposed to solve this problem. Model-based testing describes a category of approaches where a dedicated test model is available for analysis and test case generation. The advantage of such approaches is that once the test model has been derived, test case generation is usually fully automatic, and the test model can serve as a test oracle at the same time. With specification-based testing we mean the case of model-based testing where a formal specification of the system under test is available that can be analyzed with formal methods; for example, model checking can be applied to verify formal specifications against certain properties. In the context of software testing it has been shown that model checkers can also be used to derive test cases from formal specifications.

A model checker is a verification tool which takes as input an automaton based model and a temporal logic property. It exhaustively examines the state space of the model using one of several available efficient techniques, and reports whether the property is satisfied or not. In case of violation a counterexample is returned, which in practice is a linear sequence of states illustrating the property violation. The idea of testing with model checkers is to interpret counterexamples as test cases, and force generation of counterexamples with different strategies. We assume a direct mapping of counterexamples to test cases:

Definition 1 (Test Case) *A test case* $t := \langle s_0, s_1, ...s_n \rangle$ *is a finite sequence of states* s_i*.*

Each of the states s_i of a test case is part of the model's state space, and can be interpreted as a valuation of the model's state variables. Execution of

test cases can be done by partitioning the variables into input, output, and internal variables; input variables are used as test data, and output variables are used as test oracle. The length of a test case $t := \langle s_0, s_1, \ldots, s_n \rangle$ is its number of states; i.e., $length(t) = n$. A test suite is a finite set of test cases; its size is the number of test cases in the set, and its length is the sum of the lengths of its test cases.

Strategies to generate test cases are mostly based on coverage criteria or mutation. A coverage criterion is represented as a set of *trap properties* (Gargantini and Heitmeyer, 1999), where each trap property claims that a certain coverage goal is not achievable. If possible, the model checker returns a counterexample which automatically covers the underlying coverage goal. A similar approach is possible using mutation: Ammann and Black (1999) create logical formulas that "reflect" the transition relation of a model; this process is called *reflection*. If these reflected properties are mutated, the mutants can be used as trap properties (Ammann et al., 1998). Resulting test cases execute a transition where the mutant transition differs from the original transition relation. Recently, a similar technique was proposed by Gargantini (2007) for testing abstract state machines. We will use the term trap property synonymously for test goal in this paper.

Trap properties are formulas specified with temporal logics, which are modal logics with special operators for time. CTL*, introduced by Emerson and Halpern (1982), is one of the most popular temporal logic. In practice, most current model checkers do not support full CTL* but only one of the subsets linear time logic LTL (Pnueli, 1977) (Linear Temporal Logic) or branching time logic CTL (Clarke and Emerson, 1982) (Computation Tree Logic). Most trap properties can be defined in the common subset of LTL and CTL, therefore in our experiments we used both CTL and LTL to represent the same trap properties, depending on the model checker used. For example, the CTL trap property $\mathbf{AG}\ (\neg x)$ (always globally x is false) results in a test case that reaches a state where x is true, and the LTL trap property $\Box\,(x \rightarrow \bigcirc \neg y)$ (always x implies not y in the next state) results in a test case where x is true in a state and y is true in the following state.

In this paper, we use Linear Temporal Logic (LTL) (Pnueli, 1977) for examples. An LTL formula consists of atomic propositions, Boolean operators and temporal operators. The operator "$\bigcirc$" refers to the *next* state. E.g., "$\bigcirc\ a$" expresses that a has to be true in the next state. "U" is the *until* operator, where "$a\ \mathsf{U}\ b$" means that a has to hold from the current state up to a state where b is true. "$\Box$" is the *always* operator, stating that a condition has to hold at all states of a path, and "$\Diamond$" is the *eventually* operator that requires a certain condition to eventually hold at some time in the future. The syntax of LTL is given as follows, where AP denotes the set of atomic propositions:

Definition 2 (LTL Syntax) *The BNF definition of LTL formulas is given below:*

$$\phi ::= true \mid false \mid a \in AP \mid \neg\, \phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid$$
$$\phi_1 \ \mathsf{U} \ \phi_2 \mid \bigcirc \phi \mid \Box\, \phi \mid \Diamond\, \phi$$

The semantics of LTL formulas is defined using infinite execution paths. When model checking, different techniques are applied in order to verify the execution paths of Kripke structures against given temporal logic properties. Explicit model checking (Clarke et al., 1983; Lichtenstein and Pnueli, 1985; Queille and Sifakis, 1982; Vardi and Wolper, 1986) treats the state space of the model explicitly, which can quickly lead to the state explosion problem. To attenuate the effects of the state explosion problem, *symbolic model checking* (McMillan, 1993) uses ordered binary decision diagrams (BDDs (Bryant, 1986)) to represent states and function relations on these states efficiently, which allows the representation of significantly larger state spaces. *Bounded model checking* (Biere et al., 1999) 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.

Trap properties are used to represent coverage criteria. In general, a coverage criterion is simply a rule or collection of rules that impose requirements on a test suite (Ammann and Offutt, 2008). In the case of testing with a model checker, these requirements are expressed as trap properties:

Definition 3 (Trap Property) *A trap property ϕ for a test model K is a property such that $K \not\models \phi$. Any counterexample for ϕ is interpreted as a test case that satisfies the test requirement posed by ϕ.*

Trap properties are usually generated automatically with regard to coverage criteria:

Definition 4 (Coverage Criterion) *A coverage criterion C is a rule for generating trap properties on a test model K. $C(K)$ denotes the set of trap properties obtained by applying C to K.*

Definition 5 (Coverage Satisfaction) *A test suite T satisfies a coverage criterion C on test model K if and only if for each trap property $\phi \in C(K)$, there exists, as a test $t \in T$, an interpretation of some counterexample of ϕ.*

3 Minimization and Monitoring

In practice, the resources available for software testing are limited. This means that the number of test cases that can be executed is also limited. Therefore, testing has to focus on a subset of the possible test cases, even though the number of test cases that could be generated is usually much larger. In order to assure that the selected test cases are equally distributed across the software's possible behavior, test cases are often selected with respect to coverage criteria. Note that there is almost no evidence of the relation between coverage and fault detection. In this section, we first consider the classical *test suite minimization* problem, which is often discussed in the context of regression testing, where a system is re-tested after some changes. Test suite minimization assumes that a large set of test cases is available, out of which a subset is chosen. In contrast, *test case generation monitoring* tries to avoid that unnecessary test cases are created in the first place. This is mainly necessary if test case generation is expensive.

3.1 Test Suite Minimization

Test suite minimization (also known as test suite reduction) is often applied in the context of regression testing, when software is re-tested after some modifications. The costs of running a complete test suite against the software repeatedly can be quite high. In general, not all test cases of a test suite are necessary to fulfill some given test goals. Therefore, the aim of test suite minimization is to find a subset of the test cases that still fulfills the test goals. The original test suite minimization problem is defined by Harrold et al. (1993) as follows:

Given: A test suite TS, a set of goals $r_1, r_2, , , r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of TS, $T_1, T_2, ..., T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to test r_i.

Problem: Find a representative set of test cases from TS that satisfies all r_is.

The goals r_i can represent any objectives of testing, e.g., test coverage. A representative set of test cases must contain at least one test case from each subset T_i. The problem of finding the optimal (minimal) subset is NP-hard, which can be shown by a reduction to minimum set covering problem (Garey and Johnson, 1979). A simple greedy heuristic (Chvatal, 1979) to the minimum set covering problem can therefore also be applied for test-suite minimization: The heuristic selects the test case that satisfies the most test goals and removes

all test goals satisfied by that test case. This is repeated until all test goals are satisfied. Several other heuristics have been presented (Gregg Rothermel, 2002; Harrold et al., 1993; Marré and Bertolino, 2003; Tallam and Gupta, 2005; Zhong et al., 2006).

Test suite reduction results in a new test suite, where only the relevant subset remains and the other test cases are discarded. Intuitively, removing any test case might reduce the overall ability of the test suite to detect faults. In fact, several experiments (Heimdahl and Devaraj, 2004; Jones and Harrold, 2003; Rothermel et al., 1998) have shown that this is indeed the case, although there are other claims (Wong et al., 1995). Note that the reduction of fault sensitivity would also occur when using an optimal instead of a heuristic solution.

3.2 Test Case Generation Monitoring

In most cases, test suite minimization is applied after creating a complete test suite, while only some techniques first analyze test goals and then create a test suite. For example, subsumption analysis as proposed by Marré and Bertolino (2003) or Hong and Ural (2005) first analyzes test goals to determine subsumption between test goals, and then calculates a good order from these results. However, as a test case might cover several test goals even though it does not subsume them, the test suite size could be further reduced. If the test case generation is computationally cheap then minimization as a post-processing step is acceptable. If, however, the test suite generation is costly, then it is preferrable to avoid the creation of test cases that are not going to be used. One possibly way to achieve this is to check which of the remaining test goals are satisfied whenever a new test case is created. We call this process *test case generation monitoring*, as the test goals are monitored.

Monitoring can reduce the size of a test suite significantly, but a resulting test suite is not necessarily a minimal test suite: For example, the last generated test case might cover several other test goals, and therefore some previously generated test cases could be removed by a traditional minimization approach. Monitoring can result in less test cases than subsumption techniques (e.g., (Hong and Ural, 2005; Marré and Bertolino, 2003)), because a test case created for one test goal might cover several other test goals which are not subsumed.

The actual monitoring is largely a technical problem and depends on the representation of test goals and test cases. If we consider a scenario where test cases are generated with a model checker, then in the simplest approach the model checker is called for each trap property, which results in a counterexample for each feasible trap property. To avoid that test goals are covered redundantly, the test case generation can be monitored. Each time a new test

case is created, the remaining test goals are analyzed to find out which are already covered. The underlying assumption is that this analysis is computationally cheaper than model checking the trap properties. For example, in (Fraser and Wotawa, 2007) LTL rewriting was used for the analysis, and in all but very small models the performance improvement is significant. This monitoring is much simpler if only state expressions are allowed as test goals (e.g., (Hamon et al., 2004)). However, the actual improvement is expected to depend on the order in which test goals are selected. In the worst case, each test goal results in a distinct test case, such that a test case covers no other test goals, or only those of test cases that were previously generated. In the best case, the first test case created achieves full coverage.

In previous papers on monitoring test case generation with model checkers (e.g., (Fraser and Wotawa, 2007; Hamon et al., 2004)), the order of test goal selection was assumed to be performed nondeterministically, or not relevant at all if monitoring was not applied. In practice, this usually means that the trap properties are used in the order in which they are created.

4 Experiments on the Effects of the Test Case Generation Order

Intuitively, the order in which test cases are generated has an immediate effect on the size of the resulting test suite. In the best case a single test case might achieve full coverage, while in the worst case one test case has to be generated for each test goal. These two scenarios are the border cases, and therefore the question remains what the effect of the order is in practice. To answer this question, this section presents the results of two sets of experiments.

4.1 Initial Experiments

To analyze the effects of the order, we conducted a set of experiments on example models. Car Control (CA) is a simplified model simulating a car controller. The Safety Injection System (SIS) example was introduced by Bharadwaj and Heitmeyer (1999a) and has since been used frequently for studying automated test-case generation. Cruise Control (CC) is based on (Kirby, 1987) and has also been used several times for automated test-case generation, e.g., (Ammann et al., 2001, 1998). Windscreen Wiper (WP) is a windscreen wiper controller model provided by Magna Steyr, which has four Boolean and one 16 bit integer input variables, three Boolean and one 8 bit integer output variables, and one Boolean, two enumerated and one 8 bit integer internal variables. The system controls the windscreen heating, speed of the windscreen wiper and provides water for cleaning upon user request. The models are specified

using the input language of the model checker NuSMV (Cimatti et al., 1999).

Test suites were created for the example models with and without monitoring for different coverage criteria. We used the criteria *Simple Transition* (T), *Guard* (G), and *CompleteGuard* (CG) as described by Heimdahl et al. (2001). In addition, *Condition* (C) coverage was used as a weakened variant of CompleteGuard (i.e., Multiple Condition) coverage. Finally, the *Mutation* approach presented by Ammann and Black (1999) was used as well; here transitions of the model are represented as temporal logic formulas and then mutated.

Table 1
Trap property statistics.

Example	Trap Properties						
	Total	Infeasible	T	G	C	CG	Mutation
SIS	531	257	15	30	76	106	304
CA	834	369	16	32	112	284	390
Cruise	942	508	13	26	98	458	348
Wiper	1821	332	89	178	470	694	390

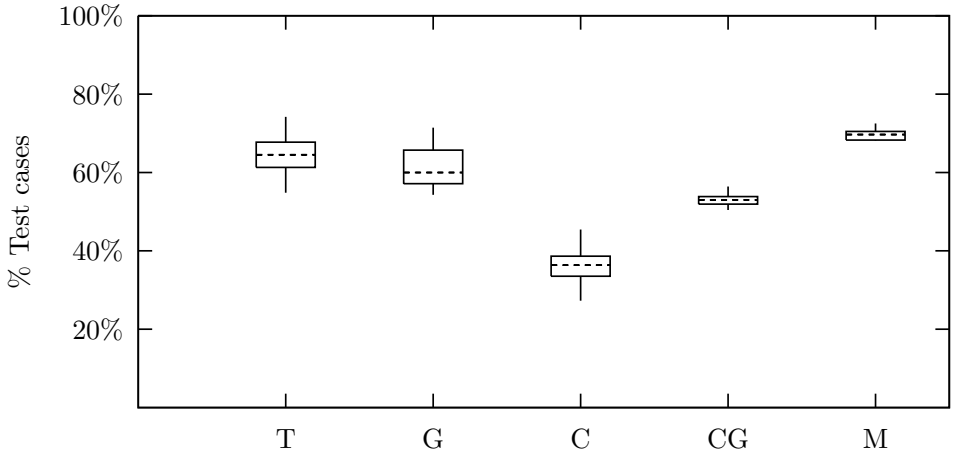

Fig. 1. WP: Reduction of number of test cases by monitoring with random selection vs. no monitoring.

Figures 1–4 illustrate how the different random orders affect the outcome in terms of the number of test cases as box plots (showing minimum, maximum, median, and 1st and 3rd quartile). Values represent the size as a percentage in comparison to the size without monitoring. As can be seen, the space for optimization is very limited: depending on the coverage criterion, an improvement of 5–10% seems possible in the best case compared to a bad random selection. Interestingly, the more trap properties there are, the less optimization is possible.

Intuitively, automatically generated trap properties for structural coverage criteria are equally distributed across the model's behavior, which might be a rea-

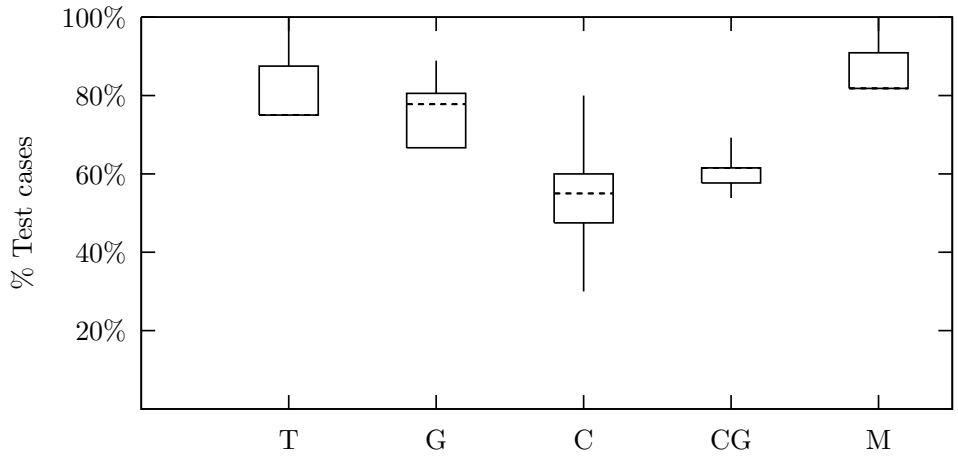

Fig. 2. SIS: Reduction of number of test cases by monitoring with random test goal selection.

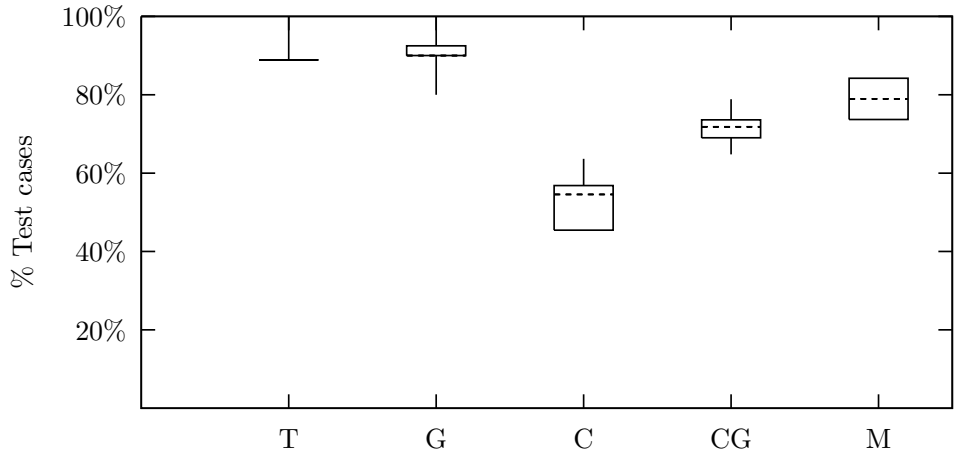

Fig. 3. CC: Reduction of number of test cases by monitoring with random test goal selection.

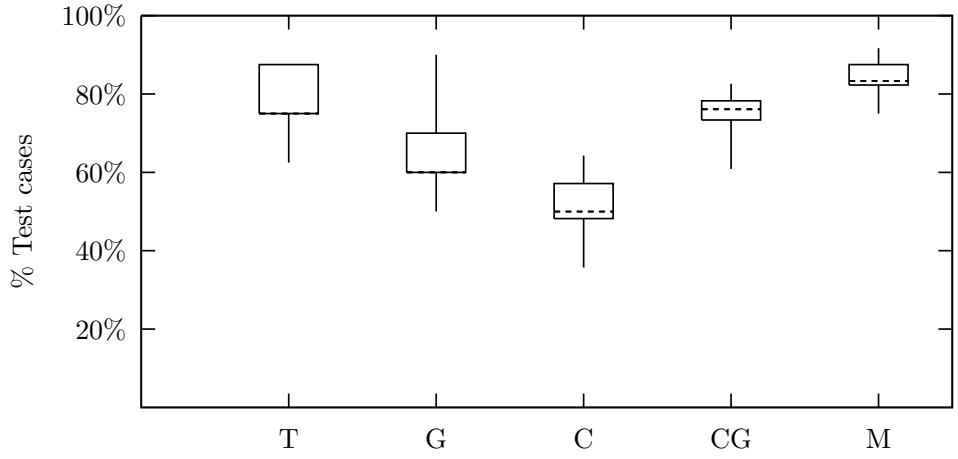

Fig. 4. CA: Reduction of number of test cases by monitoring with random test goal selection.

son for the small variation in the observed reduction. Therefore, we repeated the experiment, but used a requirement property based coverage criterion introduced by Tan et al. (2004) in contrast to the previously used structural coverage criteria. While the random choice for structural coverage criteria results in a standard deviation of 4% regarding the size, it is only 1.7% for the 342 property coverage trap properties on the windscreen wiper application. Consequently, the observed small variation is not unique to structural coverage criteria.

4.2 Second Set of Experiments

The initial set of experiments suggested that the actual order in which test cases are selected only has a minor effect on the size of the resulting test suites. There are threats to the validity of the initial experiments: Test case generation was only performed using the symbolic model checker NuSMV (Cimatti et al., 1999) which implements the classical counterexample generation technique for symbolic model checking (Clarke et al., 1995); using different techniques to derive test cases likely has an influence on the results. In order to validate the results we performed a set of additional experiments using three different SCR (Software Cost Reduction method (Heitmeyer, 2002)) specifications: The Safety Injection System (SIS) is a simplified version of the system described in (Courtois and Parnas, 1993), it monitors water pressure and adds coolant to the reactor core when the pressure falls below some threshold. The complete specification of SIS can be found in (Bharadwaj and Heitmeyer, 1999b). The Bombrel specification contains a simplified subset of the bomb release requirements of a U.S. Navy attack aircraft (Alspaugh et al., 1992) and it describes conditions under which the aircraft's OFP is required to issue a bomb release pulse. The Cruise Control System (CCS) (Jeffords and Heitmeyer, 2001) describes the core logics of an automotive cruise control, it monitors several quantities in its environment, e.g., the position of the cruise control lever and the automobiles speed, and uses this information to control a throttle.

SCR specifications consist of different types of tables, and coverage criteria can be defined for the logical expressions contained in the tables. For example, modified condition/decision coverage (MCDC) criterion (Chilenski and Miller, 1994) requires that each literal (condition) is shown to independently affect the value of the condition (decision) it is part of. There are several different flavors of MCDC; we use the variant masking MCDC (Chilenski and Richey, 1997), and MCDC with event expansion. We also use coverage criteria specific to SCR specifications and their tables:

- 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.
- Boundary coverage (DS and BOUND): Disequality split ($\geq$ becomes $>$ or $=$) and boundary coverage.

Each of these coverage criteria can automatically be represented as a set of trap properties. Such sets were generated for the different specifications; these are summarized in Table 2. Test cases were generated using several different model checkers:

- The symbolic model checker NuSMV (Cimatti et al., 1999) (only the symbolic model checker was used, although NuSMV also supports bounded model checking)
- The symbolic model checker Cadence SMV [2] (only the symbolic model checker was used, although Cadence SMV also supports bounded model checking)
- The explicit state model checker SPIN (Holzmann, 1997), supporting depth first search (DFS) and breadth first search (BFS)
- The symbolic and bounded model checker SAL (de Moura et al., 2004). The bounded model checker uses the SAT solver MiniSAT (Eén and Sörensson, 2003). The default depth of 10 was used for bounded model checking on the CCS and Bombrel examples, while the depth was increased to 111 for the SIS example.

The test suites were analyzed with regard to the coverage achieved by each test case. This information allows one to determine a minimal test suite that achieves full coverage, as described in Section 3: The problem can be represented as a set covering problem, and a greedy algorithm is used to derive a minimal test suite; i.e., a test suite where removing any test case will lead to at least one of the test goals to be unsatisfied. Note that this does not guarantee an optimal test suite in terms of the number of test cases, but unfortunately determining the optimal test suite is not feasible. Using the same greedy algorithm but picking the worst test case (i.e., the one that achieves the least coverage) was used to create a "maximal" test suite. The maximal test suite represents a worst case scenario, where the worst possible test goals is chosen at every step. Although both minimal and maximal test suites are not optimal, they provide suitable boundaries as to what effects the ordering can possibly have.

Finally, we simulated different test goal orders by randomly picking unsatisfied test goals until full coverage was achieved. This experiment was repeated 100000 times for each of the specifications and model checkers, and the results were statistically analyzed.

Table 2 describes the different example models in terms of the number of test

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

goals (trap properties) they result in. A number of test goals are infeasible; such test goals are not a problem when using model checkers to derive test cases, as an infeasible test goal simply means that a trap property is satisfied by the test model. In the experiments presented in this section we only considered the feasible test goals, as no matter in which order the test goals are selected each of the infeasible test goals has to be considered once.

Table 2
Trap property statistics.

Example	Trap Properties							
	Total	Infeasible	T	SM	MCDC	MCDC(ee)	DS	BOUND
SIS	131	13	12	19	40	44	8	8
CCS	140	19	12	15	51	62	0	0
Bombrel	139	3	13	21	39	60	4	2

Table 3 lists the total test suite length and generation time of the test case generation process for the models and trap properties shown in Table 2, using different model checkers. The number of test cases for each model equals the number of feasible trap properties, which can be derived from Table 2 (total number of trap properties - number of infeasible trap properties).

Table 3
Test suite statistics.

MC	SIS		CCS		Bombrel	
	Length	Time (s)	Length	Time (s)	Length	Time (s)
NuSMV	2967	30.4	924	40.7	1634	457.1
Cad. SMV	2967	52.7	53	2.4	1689	110.9
SPIN/BFS	2930	98.1	577	111.3	558	315.7
SPIN/DFS	88129	91.6	3281	118.3	870158	501.5
SAL/SMC	2628	82.3	581	72.2	1664	160.4
SAL/BMC	8919	1204.7	918	71.1	991	82.9

The results of the analysis with regard to different orders are summarized in Tables 4–6. The greedy algorithm clearly creates the smallest test suites with regard to the number of test cases. At the same time, a minimal number of test cases does not guarantee minimal test suite length; in fact some of the random orders achieve test suites of smaller total length. The same result holds for the generation time; random order can sometimes outperform the time it would need to generate the minimal test suite. All random orders perform better than the maximal test suites. However, the main observation here is that the range between minimal and maximal size is rather small compared to the total number of test goals, and the random order performs very close

to the minimal size. Note that in order to generate a minimal test suite *all* test cases have to be generated first, while for random order it is assumed that this is not done.

Table 4
Minimization statistics: SIS.

MC	# Test Cases			Total Length			Creation Time (s)		
	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
NuSMV	18	32	25.4	492	799	660.6	4.7	8.1	6.6
Cad. SMV	23	34	27.2	582	880	698.3	10.3	15.2	12.3
SPIN/BFS	15	26	22.3	496	779	658.0	14.4	23.9	20.3
SPIN/DFS	9	27	12.9	9326	25511	14683.3	7.3	21.9	10.5
SAL/SMC	13	27	16.3	428	764	509.3	11.5	22.0	14.5
SAL/BMC	16	33	18.8	1425	2690	1672.9	177.5	394.4	211.3

Table 5
Minimization statistics: CCS.

MC	# Test Cases			Total Length			Creation Time (s)		
	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
NuSMV	13	20	15.8	103	165	126.9	4.0	5.4	7.0
Cad. SMV	8	11	8.4	42	53	43.6	1.7	2.4	1.8
SPIN/BFS	18	30	20.3	96	145	106.3	17.2	28.1	18.8
SPIN/DFS	13	28	14.6	411	708	468.1	12.9	27.3	14.4
SAL/SMC	16	31	19.9	89	151	105.7	9.7	19.1	12.1
SAL/BMC	14	27	17.3	119	192	141.9	8.2	16.2	10.1

Table 6
Minimization statistics: Bombrel.

MC	# Test Cases			Total Length			Creation Time (s)		
	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
NuSMV	30	50	35.9	435	676	504.0	103.6	171.0	122.5
Cad. SMV	106	114	106.6	1566	1638	1570.6	91.6	97.5	92.1
SPIN/BFS	21	36	25.8	119	181	142.2	75.4	103.4	86.3
SPIN/DFS	30	42	34.1	300030	338825.3	420042	177.6	199.4	243.0
SAL/SMC	47	83	53.7	611	1244	715.9	54.8	109.3	64.1
SAL/BMC	27	84	36.2	263	749	339.4	19.3	59.9	25.7

Figure 5- 13 give more insight into how the different random orders affect the outcome in terms of number of test cases, total length and creation time

as box plots again (showing minimum, maximum, median, and 1st and 3rd quartile). The number of test cases is always significantly lower for SPIN using DFS, which is because this leads to much longer test cases, which can also be observed in the figures on the test suite length.

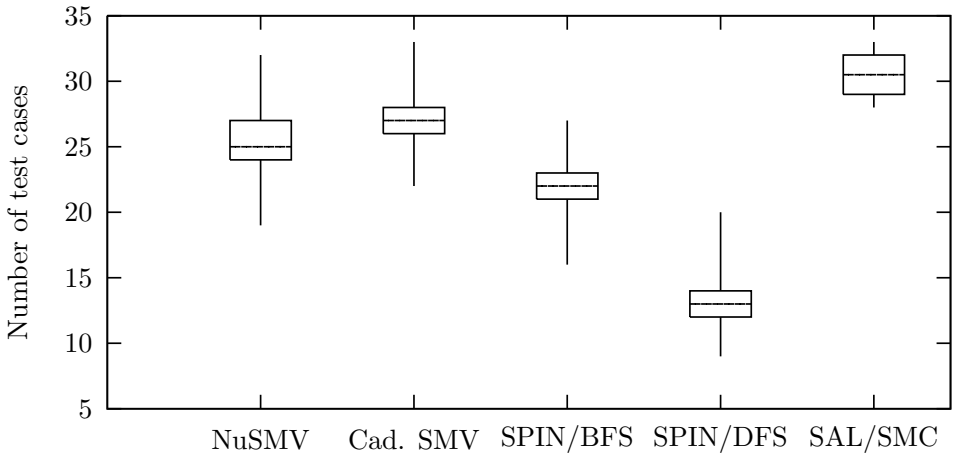

Fig. 5. SIS: Number of test cases (Maximum = 118) resulting for 100000 different random orders with monitoring.

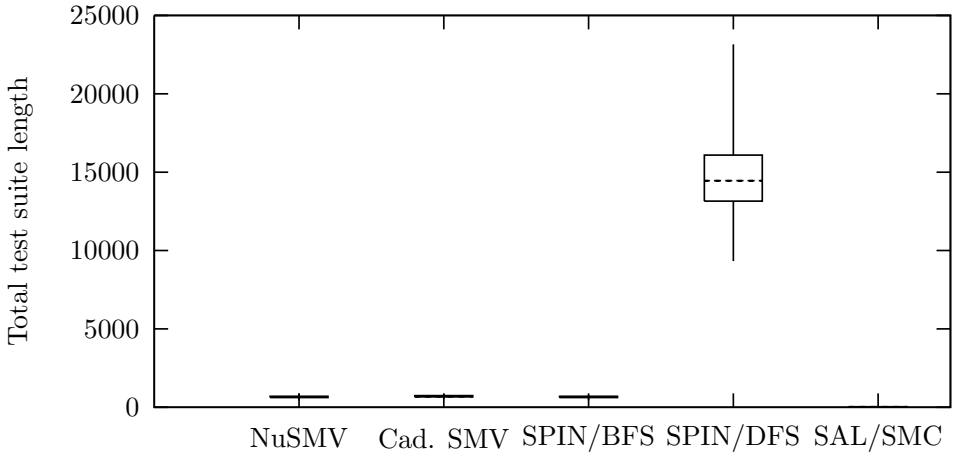

Fig. 6. SIS: Test suite length for 100000 different random orders with monitoring.

4.3 Discussion

In general, the monitoring of the test case generation achieves a significant reduction of the number of test cases. Although our experiments do not verify this, it seems natural that the actual fault sensitivity of the test suites will be reduced by this process just like for traditional test suite minimization (Heimdahl and Devaraj, 2004; Jones and Harrold, 2003; Rothermel et al., 1998) – the less testing is performed, the less likely it is that faults will be detected. Unfortunately, limited resources often make it necessary to apply such reduction techniques.

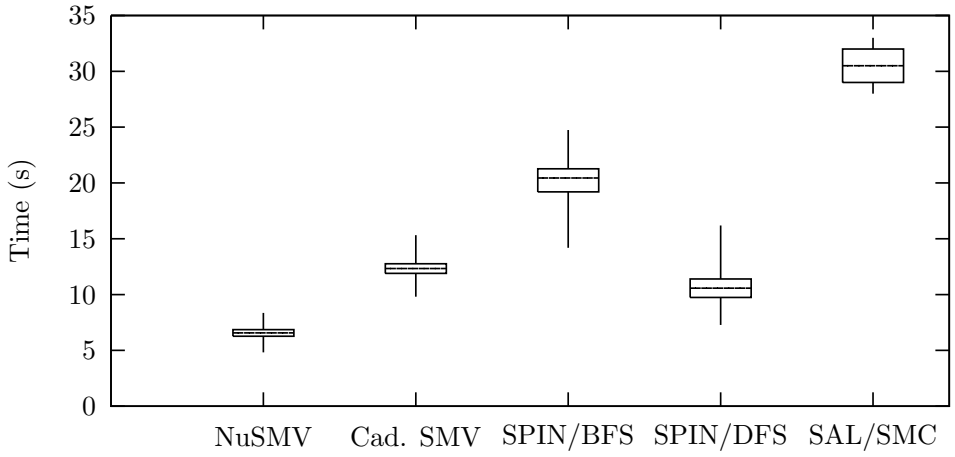

Fig. 7. SIS: Creation time for 100000 different random orders with monitoring.

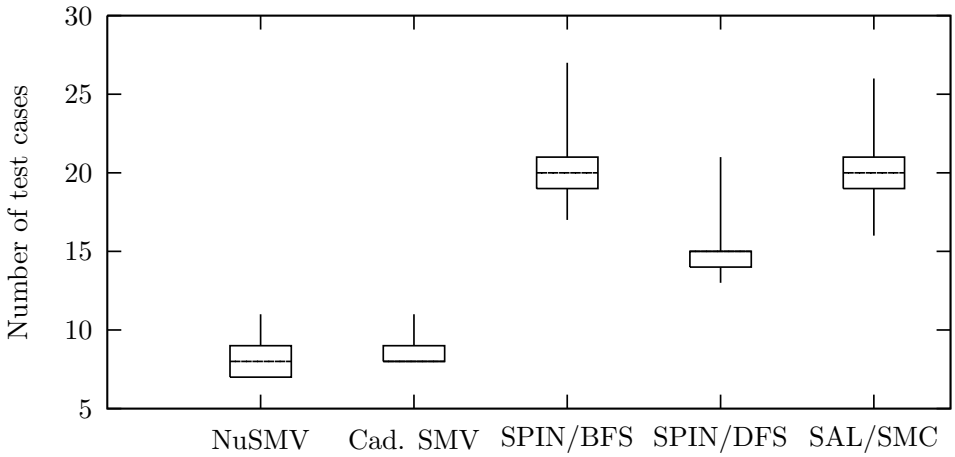

Fig. 8. CCS: Number of test cases (Maximum = 120) resulting for 100000 different random orders with monitoring.

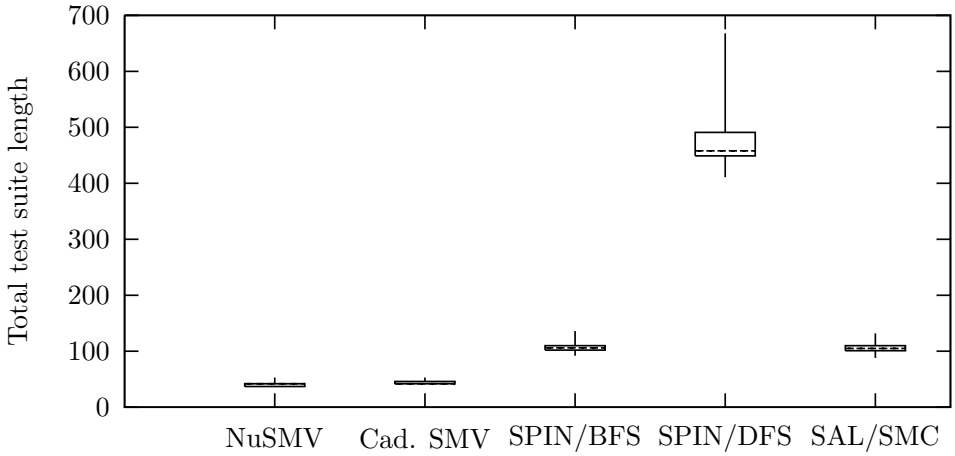

Fig. 9. CCS: Test suite length for 100000 different random orders with monitoring.

The worst case of a possible reduction was observed in the Bombrel example using Cadence SMV to generate test cases. Here, the largest test suite that can

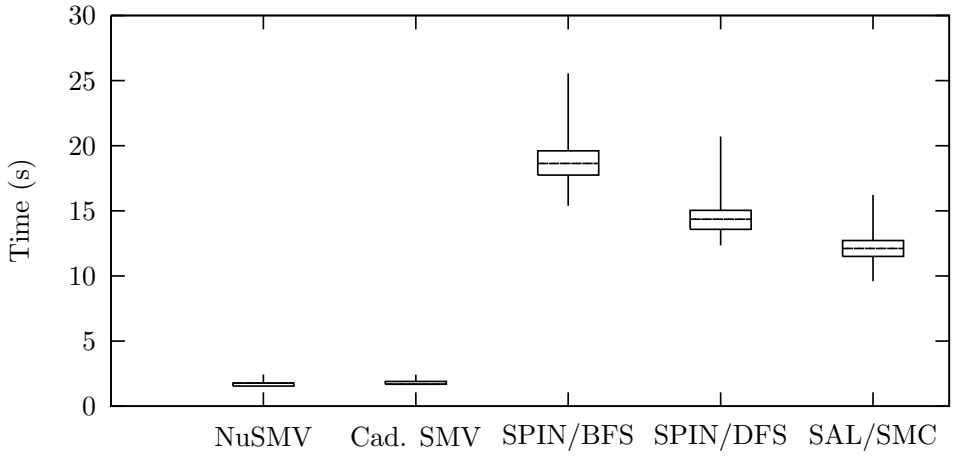

Fig. 10. CCS: Creation time for 100000 different random orders with monitoring.

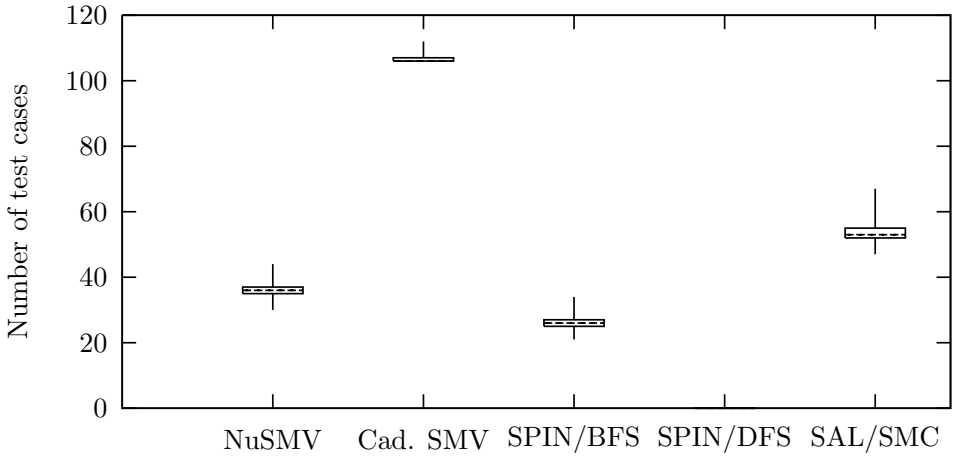

Fig. 11. Bombrel: Number of test cases (Maximum = 136) resulting for 100000 different random orders with monitoring.

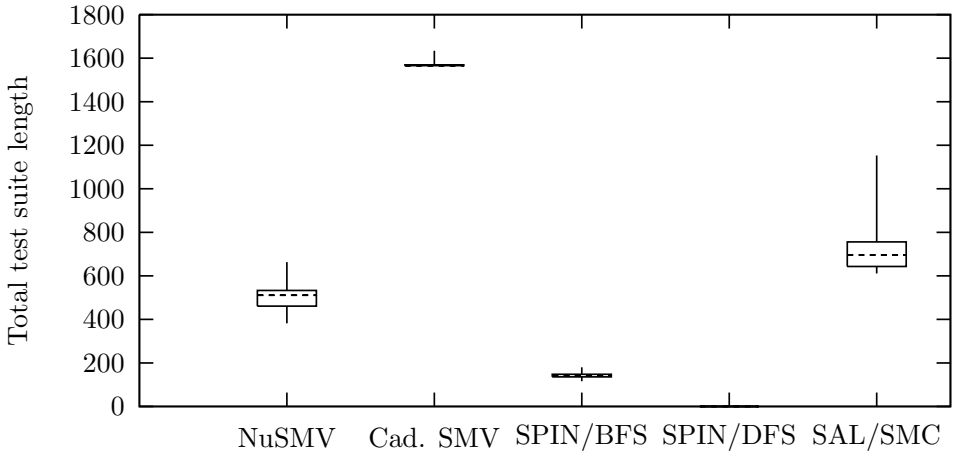

Fig. 12. Bombrel: Test suite length for 100000 different random orders with monitoring. SPIN/DFS is omitted because the scale would distort the figure; refer to Table 6 for the numbers.

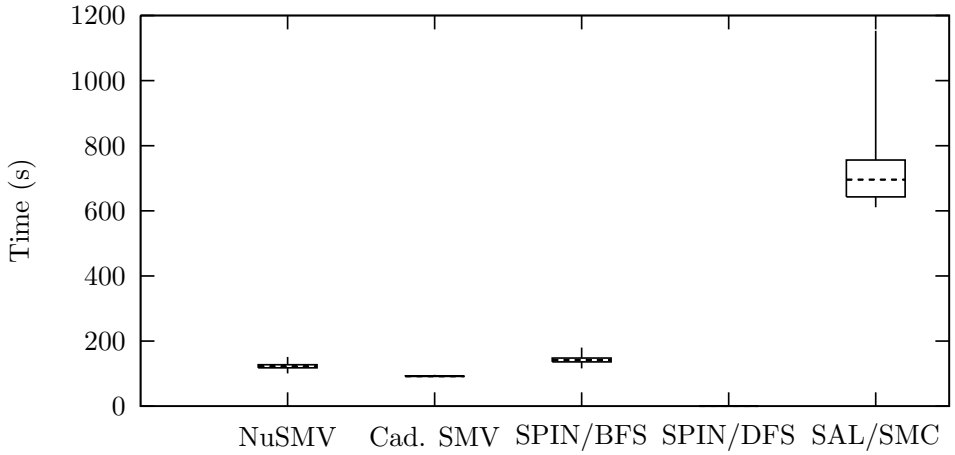

Fig. 13. Bombrel: Creation time for 100000 different random orders with monitoring.

be derived with monitoring is only 16.2% smaller than the original test suite. However, on average the worst possible reduction is 64.1% of the original test suite size. This means that even if the worst case occurs this will in practice achieve a significant reduction.

In contrast, the largest possible reduction on average for all examples is 79.4%. That means there is an average range of 15.3% of the total number of test cases between the best and worst possible result for all our examples. If a smaller test suite size is advantageous but not critically important, then we can conclude that any ordering will be sufficient if monitoring is applied. The average case of all random orderings is a reduction of 75.6%, which is close to the best case. Considering how little variance we observed with our random samples it also seems that it is very difficult to actually find a bad order, while the majority of orders perform very well and very close to the minimum.

When comparing the reduced size to the size of a full test suite any ordering promises a significant improvement. However, when performance and time are critical one might prefer to view the results from a different angle: On average, the minimal test suites for the SIS example are only 51,9% and the random test suites are 68,4% of the sizes of the test suites using the worst possible order; for CCS the sizes are reduced to 57,9% and 67,2% respectively, and for Bombrel the sizes are reduced to 61,9% and 71,0% respectively. The values for time and total length are similar to the values for the sizes. When viewed this way, there is potential to optimize the test case generation by prioritizing test goals.

Applying monitoring to the test case generation process does not guarantee a minimal test suite, because a new test case can always cover several other test goals for which other test cases have already been created, and thus can serve to replace previously generated test cases. This means that some of the test cases can possibly be removed using test suite minimization heuristics

once a test suite has been generated, even when using monitoring. However, the monitoring might prevent the generation of one or more test cases that are important for a small test suite. Even though a resulting test suite can be minimized it might still be larger than a minimized test suite derived from a complete test suite. Consequently, when the objective is to derive the smallest possible test suite for a given criterion it might be necessary to avoid monitoring and apply the minimization after the test case generation is finished.

As a threat to validity we have to note that the minimum and maximum were calculated with the greedy algorithm described above; therefore, these values are not optimal values. For example, this can be observed for the Bombrel example and the bounded model checker SAL, where the greedy algorithm suggests a minimum size of 27, while one of the 100000 random samples achieves a size of 26 test cases.

Our general conclusion is that a random ordering will be sufficient in many cases. The results achieved by our random sampling show that this will in general lead to very good reductions. If, however, performance is critical, then risking a bad ordering is not a good idea, and the range from the average random case to the best case has some promise for optimization.

5 Prioritizing Test Goals

In the experiments described in the previous section we observed that a significant reduction of test suite size and generation time can be achieved regardless of the order in which test goals are selected. However, random order still results in more test cases than a greedy post-creation minimization algorithm. Therefore we would like to investigate whether a presorting of the test goals can improve the results of the test case generation. As we are assuming that the time for test case generation is critical, the presorting should be as simple as possible. For example, this means that in this paper we try to avoid an in-depth subsumption analysis as suggested by Hong and Ural (2005).

In this section we evaluate three simple metrics that only consider the test goals, i.e., they do not consider the test model or any domain specific knowledge. The basic idea underlying any prioritization of test goals is that some test goals will result in longer test cases, and others in shorter test cases. Intuitively, a longer test case will cover more test goals at once, therefore aiming to create longer test cases earlier is expected to reduce the number of distinct test cases necessary to satisfy a coverage criterion. Table 7 lists the correlation coefficients between test case length and number of test goals covered by a test case for the different models and model checkers. The correlation is usually very high. Interestingly, bounded model checking results in test cases

with a lower but still high correlation. The Bombrel example also has lower correlation when using symbolic model checking, but the correlation is still positive.

Table 7
Correlation between test case length and number of satisfied test goals.

MC	SIS	CCS	Bombrel
NuSMV	0.91	0.97	0.44
Cad. SMV	0.89	0.98	0.21
SPIN/BFS	0.91	0.92	0.95
SPIN/DFS	0.95	0.95	0.62
SAL/SMC	0.90	0.85	0.18
SAL/BMC	0.76	0.56	0.38

The problem is that the length of a test case is not known a priori. It is therefore necessary to estimate the length of a test case given its test goal. In the case of model checker based testing this means a heuristic needs to estimate the length of a counterexample for a given trap property. In this section we present three different heuristics to estimate the counterexample length.

Another assumption is that it is cheaper to create one long test case than to create several short test cases. This might not always the case: For example, bounded model checking can get very computationally expensive for large bounds, while creating short test cases is cheap. In that case, the costs can be reduced by applying the metrics found in this section inversely; i.e., pick the probably shortest test case first.

5.1 Complexity Estimation

A simple intuition is to expect that complex properties result in complex counterexamples. For example, special border cases are described with many propositions, and expoundment results in more complex properties when using less common transition guards. Furthermore, a complex property is intuitively more difficult to cover by other test cases. Therefore, we define a simple heuristic to estimate the complexity of a property as follows:

Definition 6 (Property Complexity) *The complexity $c(f)$ of a temporal logic property f is inductively defined as follows, where f and g represent temporal logical formulas, and a and b are literals:*

$$
\begin{aligned}
c(\neg f) &= \bar{c}(f) \\
c(f \wedge g) &= c(f) + c(g) \\
c(f \vee g) &= min\{c(f, c(g)\} \\
c(\Box f) &= c(f) \\
c(f &= \tau \times c(f) \\
c(\Diamond f) &= \tau \times c(f) \\
c(f_1 \mathsf{U} f_2) &= c(f_1) + \\
&\quad \tau \times c(f_2)
\end{aligned}
\qquad
\begin{aligned}
c(true) &= 0 \\
c(false) &= 1 \\
c(a) &= 1 \\
c(a = b) &= 4 \\
c(a \neq b) &= 1 \\
c(a \leq, \geq b) &= 2 \\
c(a <, > b) &= 3
\end{aligned}
$$

The complexity of a property estimates how difficult it is to satisfy a property. The definition of $\bar{c}(f)$ is omitted in Definition 6 for space reasons; it is defined analogously but estimates the complexity to violate a property. That means that the calculations for $\vee$ and $\wedge$ are exchanged, as well as the value assignments for $true$ and $false$, $=$ and $\neq$, etc. The intuition behind the chosen numerical values is that equality of two values is more difficult to achieve than inequality, etc; a similar approach has been used to estimate the feasibility of paths in extended finite state machines (Derderian et al., 2008). The parameter τ represents a constant factor that is used to weigh temporal operators. We chose a value of $\tau = 0.5$ for our experiments.

Example: As an example, consider the temporal logic formula $\phi := \Box\,(a > b \wedge c = d)$. The weight of ϕ is the sum of the weights of $a > b$ and $c = d$, which is $3 + 4 = 7$.

Table 8
Correlation between test case length and complexity metric.

MC	SIS	CCS	Bombrel
NuSMV	0.03	0.27	0.34
Cad. SMV	0.03	0.27	0.35
SPIN/BFS	0.03	0.39	0.31
SPIN/DFS	0.06	0.36	0.18
SAL/SMC	0.06	0.39	0.36
SAL/BMC	0.10	-0.14	0.35

To evaluate this estimation, the complexity was calculated for all trap properties used in conjunction with the example models described in the previous section and compared to the actual counterexample length. Table 8 shows the correlation between the complexity estimation and the resulting test length for each of the models and test criteria. The correlation is positive in almost all cases, which means that the complexity estimation is feasible. However, the correlation is not very large in general, and there also are cases where it

is negative. It can be seen in Table 8 that the model checking algorithm and thus the test case generation algorithm used also has an influence. Furthermore the model under consideration also has an influence on the correlation. For example, the SIS example has a very low correlation.

Table 9
Correlation between test case length and complexity estimation.

Model	T	G	C	CG	Mut.
CA	0.48	0.43	-0.07	0.09	0.14
SIS	0.34	0.21	-0.13	0.08	0.13
CC	0.55	0.49	0.07	0.32	0.31
WP	0.84	0.84	0.22	0.59	0.50

Table 9 lists the correlation for the initial set of experiments and shows that the actual correlation depends not only on the model and model checkign algorithm but also the underlying coverage criterion. For example, transition and guard coverage, which are trap properties of the type $\Box(x \rightarrow y)$ have a higher correlation than trap properties for condition coverage, which are of the type $\Box(guard \wedge \neg condition)$.

There is only insignificant variation in the results when changing the constant values used in the metric (0-4 for atomic expressions), as long as the relative ordering is not changed. The constant value used for temporal operators only has minor effect in the considered coverage criteria.

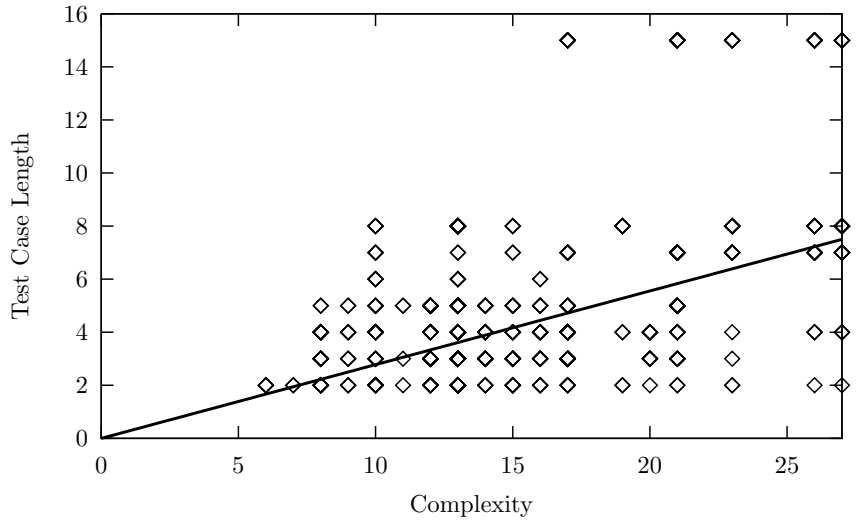

Fig. 14. Evaluation of complexity metric on WP with CG coverage, correlation = 0.59.

Figure 14 illustrates the correlation between complexity estimation and length for the windscreen wiper application and the complete guard coverage criterion. Note that Figure 14 does not reflect how often pairs of the same complexity and length occurred.

Consequently, the metric can be used to heuristically select trap properties, but it is not an admissible heuristic. Figure 14 shows that the prediction sometimes completely fails (e.g., short test cases for high complexity). This is not unexpected, as it is possible to write very complex properties that can be easily satisfied, while very simple properties might lead to very long test cases.

5.2 Distance Estimation

A slightly more complex metric can be defined using a distance estimation. Given a state and a property, this metric estimates the number of atomic propositions that have to change their value in order for the property to become true/false. The distance estimation in Definition 7 is similar to heuristics used in directed model checking (Edelkamp et al., 2001).

Definition 7 (Distance Estimation) *The distance estimation $h(f, s)$ for a temporal logic property f and a state s is inductively defined as follows:*

$$
\begin{aligned}
h(true, s) &= 0 \\
h(false, s) &= 1 \\
h(a, s) &= \begin{cases} 0 & \text{if } a \text{ is true in } s \\ 1 & \text{if } a \text{ is false in } s \end{cases} \\
h(\neg f, s) &= \bar{h}(f, s) \\
h(f \wedge g, s) &= h(f, s) + h(g, s) \\
h(f \vee g, s) &= min\{h(f, s), h(g, s)\} \\
h(\,\Box\, f, s) &= h(f, s) \\
h(fs) &= \tau \times h(f, s) \\
h(\,\Diamond\, f, s) &= \tau \times h(f, s) \\
h(f_1 \,\mathsf{U}\, f_2, s) &= h(f_1, s) + \tau \times h(f_2, s)
\end{aligned}
\qquad
\begin{aligned}
\bar{h}(true, s) &= 1 \\
\bar{h}(false, s) &= 0 \\
\bar{h}(a, s) &= \begin{cases} 1 & \text{if } a \text{ is true in } s \\ 0 & \text{if } a \text{ is false in } s \end{cases} \\
\bar{h}(\neg f, s) &= h(f, s) \\
\bar{h}(f \wedge g, s) &= min(\bar{h}(f, s), \bar{h}(g, s)) \\
\bar{h}(f \vee g, s) &= \bar{h}(f, s) + \bar{h}(g, s) \\
\bar{h}(\,\Box\, f, s) &= \bar{h}(f, s) \\
\bar{h}(fs) &= \tau \times \bar{h}(f, s) \\
\bar{h}(\,\Diamond\, f, s) &= \tau \times \bar{h}(f, s) \\
\bar{h}(f_1 \,\mathsf{U}\, f_2, s) &= \bar{h}(f_1, s) + \tau \times \bar{h}(f_2, s)
\end{aligned}
$$

Example: As an example, consider again the temporal logic formula $\phi := \Box\,(a > b \wedge c = d)$. Assuming the current state is $s = \{a = 2, b = 5, c = True, d = True\}$, then the distance $h(\phi, s)$ results as the sum of the distances to $a > b$ and $c = d$, which is $3 + 0 = 3$.

The distance measurement was evaluated using the same set of trap properties and models as for the evaluation of the complexity estimation. The initial state of the model was used, as the distance metric needs a state to estimate the distance from, and $\bar{h}(f, s)$ was used to estimate the distance to a violation.

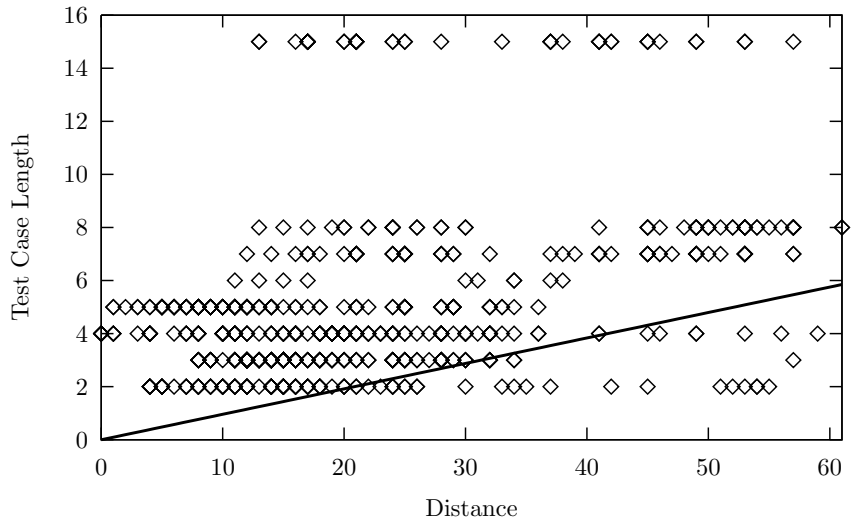

Fig. 15. Evaluation of distance metric on WP with CG coverage, correlation = 0.39.

Table 10
Correlation between test case length and distance metric.

MC	SIS	CCS	Bombrel
NuSMV	0.29	0.43	0.45
Cad. SMV	0.29	0.43	0.48
SPIN/BFS	0.26	0.54	0.36
SPIN/DFS	0.21	0.46	0.42
SAL/SMC	0.28	0.54	0.48
SAL/BMC	0.36	0.03	0.30

The results are summarized in Table 10. The correlation is quite similar to that of the complexity metric; in some cases it is better.

Figure 15 illustrates the correlation for the complete guard criterion and the windscreen wiper application again. The distance estimates are more diverse than the complexity estimates. Again the correlation is not overly large, but it shows that it is feasible and we should expect better results than in the average random case.

5.3 Dependency Measure

The third metric we evaluate simply represents the number of variables a trap property depends on. Intuitively, the more variables a trap property depends on, the more complicated it will be to create a test cases that satisfies the test goal, and thus the test case should be longer.

Definition 8 (Dependency Value) *The dependency value $d(f)$ for a temporal logic property f is defined as the number of different variables on which the variables occurring in the atomic propositions of f depend on.*

Table 11
Correlation between test case length and dependency metric.

MC	SIS	CCS	Bombrel
NuSMV	-0.26	-	0.19
Cad. SMV	-0.26	-	0.19
SPIN/BFS	-0.27	-	0.56
SPIN/DFS	-0.29	-	-0.08
SAL/SMC	-0.27	-	0.20
SAL/BMC	-0.19	-	0.47

Table 11 shows the correlation between the test case length and its dependency value. The dependency value for all trap properties in the cruise example is constant, therefore the correlation coefficient results in a division by zero. This table again illustrates that these metrics are very dependent on the models they are applied to. For the SIS example the correlation is negative with all model checkers, while it is better for the Bombrel example.

5.4 Evaluation

To further evaluate the presented metrics, we created test suites for the same criteria as used in Section 4.2, and compared the results with the random selection. The metrics were applied such that the most complex or distant trap property is always chosen next for test case generation. As described, this should reduce the total number of test cases, as longer test cases tend to cover more test goals.

Tables 12–14 summarize the results of these experiments. The results are in line with what is to be expected from the correlation analysis: The dependency value slightly reduces the test suite sizes for the Bombrel example, but not for the other two examples. Complexity and distance generally perform slightly better than the average random case, and clearly better than the worst case. The creation time is slightly reduced in almost all cases; again the dependency value sometimes increases the time for the examples with negative correlation. The distance metric in general achieves the highest reduction in the creation time. As the metrics we used are computationally very simple (linear in the number of subformulas of a property) the time to calculate the complexity-, distance-, and dependency-values can be neglected.

Table 12
Prioritization results: SIS.

MC	# Test Cases			Total Length			Creation Time		
	C	Dep.	Dist.	C	Dep.	Dist.	C	Dep.	Dist.
NuSMV	25	27	24	719	637	717	6.5	6.8	6.2
Cad. SMV	26	27	26	721	666	722	11.6	12.0	11.6
SPIN/BFS	20	22	19	658	622	656	19.1	19.7	18.6
SPIN/DFS	12	17	14	13157	18286	16416	9.8	13.9	11.5
SAL/SMC	15	18	15	466	486	467	13.2	13.9	13.1
SAL/BMC	20	23	19	1810	2026	1700	225.9	252.0	215.0

Table 13
Prioritization results: CCS.

MC	# Test Cases			Total Length			Creation Time		
	C	Dep.	Dist.	C	Dep.	Dist.	C	Dep.	Dist.
NuSMV	16	16	17	135	129	146	5.9	5.7	6.5
Cad. SMV	9	9	9	46	44	46	1.9	2.0	1.9
SPIN/BFS	18	22	18	96	110	96	17.2	20.8	17.2
SPIN/DFS	15	15	15	480	454	480	14.8	14.8	14.8
SAL/SMC	17	22	17	94	113	94	10.3	13.7	10.3
SAL/BMC	14	19	15	112	156	122	8.2	11.5	8.8

Table 14
Prioritization results: Bombrel.

MC	# Test Cases			Total Length			Creation Time		
	C	Dep.	Dist.	C	Dep.	Dist.	C	Dep.	Dist.
NuSMV	34	34	33	460	510	454	116.1	117.9	113.2
Cad. SMV	106	106	106	1566	1566	1566	91.6	91.6	91.6
SPIN/BFS	27	25	25	155	139	139	98.7	87.6	86.7
SPIN/DFS	34	32	32	340034	310046	320032	199.7	183.9	189.2
SAL/SMC	52	53	51	638	698	632	58.6	62.5	57.8
SAL/BMC	35	34	35	331	324	338	24.9	24.2	25.0

In several cases the dependency metric achieves the shortest test suites, although it also sometimes performs worse than both the random average and the other metrics. In the majority of cases the total test suite length is slightly reduced, although there are several exceptions. This was to be expected, as

the metrics aim to reduce the number of test cases, not their total length.

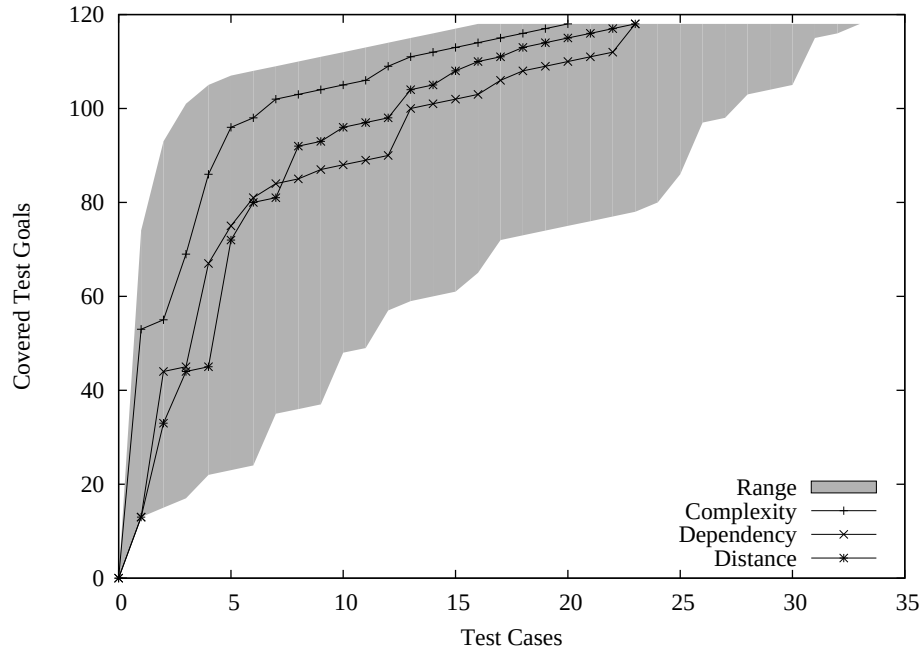

Fig. 16. Rate at which test goals are covered, SIS model, SAL bounded model checker.

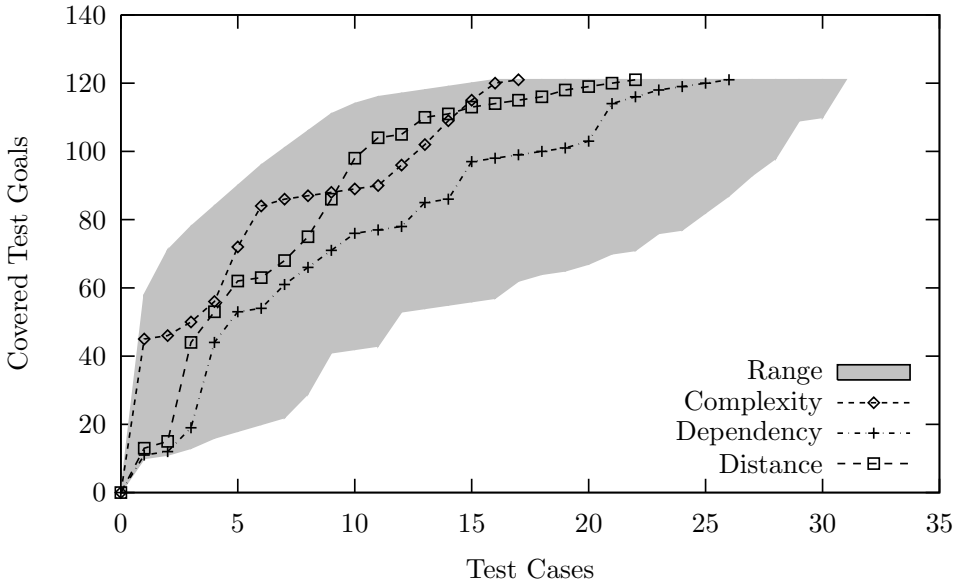

Fig. 17. Rate at which test goals are covered, CCS model, SAL symbolic model checker.

Figures 16- 18 illustrate the number of satisfied test goals as a function of the number of test case generated. The shaded areas represent the possible range in which this function can possibly lie. The upper border of this area is determined by the greedy minimization algorithm, and the lower border by using the greedy algorithm trying to satisfy as little as possible test goals. As

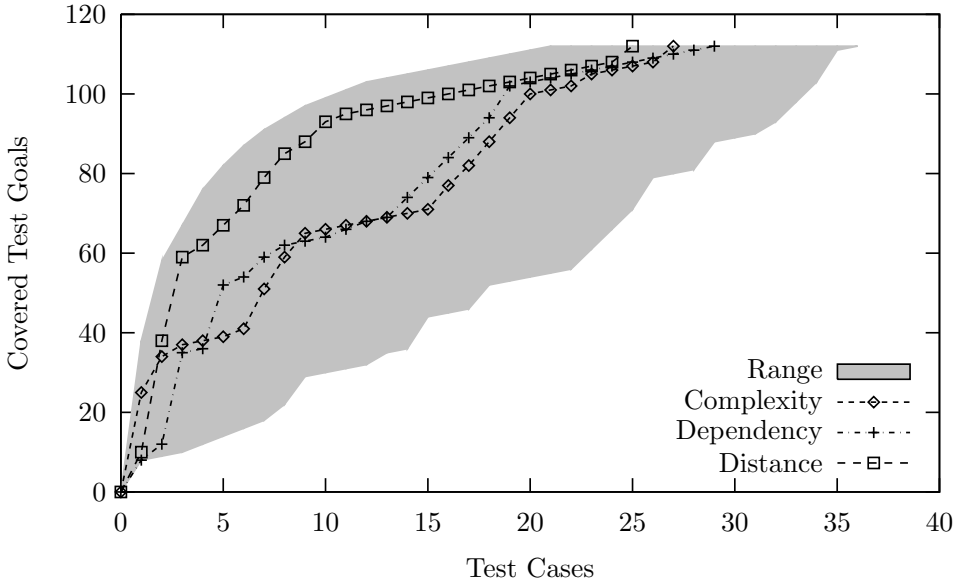

Fig. 18. Rate at which test goals are covered, Bombrel model, SPIN model checker (BFS).

can be seen all metrics make a couple of bad choices at some points, but generally perform close to the optimal curve. Again these figures show that the best prioritization technique depends on the model. For the SIS and CCS examples the complexity metric performs best, even though the length/coverage correlation was better for the distance metric. For the Bombrel example the distance metric performs best, and the dependency metric almost outperforms the complexity metric.

On average over all examples and model checkers, the complexity metric reduces the test suite size in comparison to the average random case by 3.9%, and the distance metric by 4.2%. The dependency metric reduces the test suite size for the Bombrel example by 5.8% compared to the average random case, but on average over all examples the size is increased by 6.2%. In general we can conclude that the prioritization will result in minor improvements, and given the simplicity of our metrics it seems worth the effort to avoid running into a worst case ordering. Of course it is conceivable that domain and model specific knowledge can lead to better prioritization.

6 Conclusions

This paper presented the results of a set of experiments conducted in order to see what effects monitoring of the test case generation has. In general, monitoring removes already satisfied test goals whenever a newly created test case is added, thus avoiding that test cases are unnecessarily created. This is related to test suite minimization, but has several differences: First, traditional

test suite minimization is a post-processing step that assumes an existing full test suite, while monitoring is performed during test case generation, which can be useful for both online and offline testing. Although monitoring decreases the size of a test suite significantly, it may but does not have to result in a minimal test suite. This means that further minimization might be possible.

A main objective of our experiments was to see the effects of the order in which test goals are selected. The evaluation lead to the conclusion that a significant reduction can usually be achieved with any ordering, and the range between best possible to worst possible ordering, i.e., orderings that lead to smallest/largest test suites, is not overly large. That means that in many cases it is not necessary to take an influence on the order in which test cases are generated.

When test cases generation is computationally expensive, there is still potential for optimization by prioritizing the test goals. Test case generation with model checkers, as used in our experiments, is an example of such an expensive technique. Although with the models in our experiments the test case generation time was not overly large is important to note that these models are manually abstracted versions. When using the full numerical domains instead of domain abstractions this changes the picture immediately and drastically increases the time necessary to generate a single test case.

Experimentation with three very simply metrics showed that prioritization can lead to smaller test suites. Given the right metric for the right model, the reduction compared to the mean value of all random orderings in our sample sets is in the order of 5%. It is conceivable that better prioritization can be performed when including domain knowledge or information about the models; for example, subsumption analysis (Hong and Ural, 2005; Marré and Bertolino, 2003) could be a complementary technique that could support finding better orders. It is expected that the impact of the prioritization will be larger if there is not one distinct test case per test goal, but if an iterative method is used where test cases can also be extended. This has for example been argued by Hamon et al. (2004), who iteratively extend test cases with a model checker.

References

Alspaugh, T. A., Faulk, S. R., Britton, K. H., Parker, R. A., Parnas, D. L., Shore, J. E., 1992. Software requirements for the A-7E aircraft. Tech. Rep. NRL-9194, Naval Research Lab., Wash., DC.

Ammann, P., Black, P. E., 1999. A Specification-Based Coverage Metric to Evaluate Test Sets. In: HASE '99: The 4th IEEE International Symposium

on High-Assurance Systems Engineering. IEEE Computer Society, Washington, DC, USA, pp. 239–248.

Ammann, P., Ding, W., Xu, D., 2001. Using a Model Checker to Test Safety Properties. In: Proceedings of the 7th International Conference on Engineering of Complex Computer Systems (ICECCS 2001). IEEE, pp. 212–221.

Ammann, P., Offutt, J., 2008. Introduction to Software Testing. Cambridge University Press, New York, NY, USA.

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

Bharadwaj, R., Heitmeyer, C. L., 1999a. Model Checking Complete Requirements Specifications Using Abstraction. Automated Software Engineering 6 (1), 37–68.

Bharadwaj, R., Heitmeyer, C. L., Jan. 1999b. Model checking complete requirements specifications using abstraction. Automated Software Engineering: An International Journal 6 (1), 37–68.

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

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

Chilenski, J., Richey, L. A., 1997. Definition for a masking form of modified condition decision coverage (MCDC). Tech. rep., Boeing.

Chilenski, J. J., Miller, S. P., September 1994. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, 193–200.

Chvatal, V., 1979. A Greedy Heuristic for the Set-Covering Problem. Mathematics of Operations Research 4 (3).

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

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

Clarke, E. M., Emerson, E. A., Sistla, A. P., 1983. Automatic Verification Of Finite State Concurrent System Using Temporal Logic Specifications: A Practical Approach. In: POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM Press, New York, NY, USA, pp. 117–126.

Clarke, E. M., Grumberg, O., McMillan, K. L., Zhao, X., 1995. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In: Proceedings of the 32st Conference on Design Automation (DAC). ACM

Press, pp. 427–432.
Courtois, P.-J., Parnas, D. L., 1993. Documentation for safety critical software. In: Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93). Baltimore, MD, pp. 315–323.
de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A., Jul. 2004. SAL 2. In: Alur, R., Peled, D. (Eds.), Computer-Aided Verification, CAV 2004. Vol. 3114 of Lecture Notes in Computer Science. Springer-Verlag, Boston, MA, pp. 496–500.
Derderian, K., Hierons, R., Harman, M., Guo, Q., 2008. Estimating the feasibility of transition paths in Extended Finite State Machines. Submitted for publication.
Edelkamp, S., Lafuente, A. L., Leue, S., 2001. Directed Explicit Model Checking with HSF-SPIN. In: SPIN '01: Proceedings of the 8th International SPIN Workshop on Model Checking of Software. Springer-Verlag New York, Inc., New York, NY, USA, pp. 57–79.
Eén, N., Sörensson, N., 2003. An Extensible SAT-solver. In: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Vol. 2919 of Lecture Notes in Computer Science. Springer, pp. 502–518.
Emerson, E. A., Halpern, J. Y., 1982. Decision Procedures and Expressiveness in the Temporal Logic of Branching Time. In: STOC '82: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing. ACM Press, New York, NY, USA, pp. 169–180.
Fraser, G., Wotawa, F., 2007. Using LTL Rewriting to Improve the Performance of Model-Checker Based Test-Case Generation. In: A-MOST '07: Proceedings of the 3rd International Workshop on Advances in Model-based Testing. ACM Press, New York, NY, USA, pp. 64–74.
Fraser, G., Wotawa, F., 2008. Ordering Coverage Goals in Model Checker Based Testing. In: A-MOST '08: Proceedings of the 4th International Workshop on Advances in Model-based Testing. To Appear.
Garey, M. R., Johnson, D. S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA.
Gargantini, A., 2007. Using Model Checking to Generate Fault Detecting Tests. In: Proceedings of the International Conference on Tests And Proofs (TAP). Vol. 4454 of Lecture Notes in Computer Science. Zurich, Switzerland, pp. 189–206.
Gargantini, A., Heitmeyer, C., 1999. 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. Vol. 1687. Springer, pp. 146–162.
Gregg Rothermel, Mary Jean Harrold, J. v. R. C. H., 2002. Empirical studies of test-suite reduction. Software Testing, Verification and Reliability 12 (4), 219–249.

Hamon, G., de Moura, L., Rushby, J., 2004. Generating Efficient Test Sets with a Model Checker. In: Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04). pp. 261–270.

Harrold, M. J., Gupta, R., Soffa, M. L., 1993. A Methodology for Controlling the Size of a Test Suite. ACM Trans. Softw. Eng. Methodol. 2 (3), 270–285.

Heimdahl, M. P. E., Devaraj, G., 2004. Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In: ASE. IEEE Computer Society, pp. 176–185.

Heimdahl, M. P. E., Rayadurgam, S., Visser, W., 2001. Specification Centered Testing. In: Proceedings of the Second International Workshop on Automated Program Analysis, Testing and Verification (ICSE 2001).

Heitmeyer, C. L., 2002. Encyclopedia of Software Engineering. Vol. 2. John Wiley & Sons, Ch. Software Cost Reduction.

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

Hong, H. S., Ural, H., 2005. Using Model Checking for Reducing the Cost of Test Generation. In: Formal Approaches to Software Testing. Vol. 3395 of Lecture Notes in Computer Science. Springer Verlag Gmbh, pp. 110–124.

Jeffords, R. D., Heitmeyer, C. L., 2001. An algorithm for strengthening state invariants generated from requirements specifications. In: 5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada. IEEE Computer Society, pp. 182–193.

Jones, J. A., Harrold, M. J., 2003. Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Softw. Eng. 29 (3), 195–209.

Kirby, J., 1987. Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System. Tech. Rep. TR-87-07, Wang Institute of Graduate Studies.

Lichtenstein, O., Pnueli, A., 1985. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In: POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM Press, New York, NY, USA, pp. 97–107.

Marré, M., Bertolino, A., 2003. Using spanning sets for coverage testing. IEEE Trans. Softw. Eng. 29 (11), 974–984.

McMillan, K. L., 1993. Symbolic Model Checking. Kluwer Academic Publishers, Norwell, MA, USA.

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

Queille, J.-P., Sifakis, J., 1982. Specification and verification of concurrent systems in CESAR. In: Proceedings of the 5th Colloquium on International Symposium on Programming. Springer-Verlag, London, UK, pp. 337–351.

Rothermel, G., Harrold, M. J., Ostrin, J., Hong, C., 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: ICSM '98: Proceedings of the International Conference on Soft-

ware Maintenance. IEEE Computer Society, p. 34.

Tallam, S., Gupta, N., 2005. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In: PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, New York, NY, USA, pp. 35–42.

Tan, L., Sokolsky, O., Lee, I., 2004. Specification-Based Testing with Linear Temporal Logic. In: Proceedings of IEEE International Conference on Information Reuse and Integration (IRI'04). pp. 493–498.

Vardi, M. Y., Wolper, P., 1986. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In: Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86). IEEE Computer Society, pp. 332–344.

Wong, W. E., Horgan, J. R., London, S., Mathur, A. P., 1995. Effect of test set minimization on fault detection effectiveness. In: ICSE '95: Proceedings of the 17th Int. Conference on Software Engineering. ACM Press, pp. 41–50.

Zhong, H., Zhang, L., Mei, H., 2006. An experimental comparison of four test suite reduction techniques. In: ICSE '06: Proceeding of the 28th international conference on Software engineering. ACM Press, pp. 636–640.