An Evaluation of Specification Based Test Generation Techniques using Model Checkers

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

Test case generation can be represented as a model checking problem, such that model checking tools automatically generate test cases. This has previously been applied to testing techniques such as coverage criteria, combinatorial testing, mutation testing, or requirements testing, and further criteria can be used similarly. However, little comparison between the existing techniques has been done to date, making it difficult to choose a technique. In this paper we define existing and new criteria in a common framework, and evaluate and compare them on a set of realistic specifications. Part of our findings is that because testing with model checkers represents the test case generation problem in a very flexible way best results can be achieved by combining several techniques from different categories. A best effort approach where test cases are only created for uncovered test requirements can create relatively small test suites that cover many or all different test techniques.

1. Introduction

Software testing is an important but difficult task. Because of its complexity, automation is desirable to reduce the effort and increase software quality. Among the many techniques that have been proposed for test case generation, the use of model checkers is a promising approach which can fully automatically derive test cases from a model under certain restrictions using a variety of different criteria.

Model checkers are automatic verification tools that are able to provide counterexample sequences to properties violated by a model. The idea of testing with model checkers is to represent the test case generation problem as a model checking problem such that the counterexamples provided by a model checker can be used as test cases.

For many testing techniques the representation as a model checking problem is straightforward. Therefore, several techniques have been proposed in the past, e.g., coverage criteria, mutation testing, combinatorial testing, or requirements based testing. This leaves a test engineer with the problem of which of all these techniques to choose.

In this paper we show for 17 different test criteria how to define them as a model checking problem. The majority of criteria is based on the hierarchy of criteria given in [2] for logical predicates. Some of these criteria have been used previously in different frameworks, while for others the mapping to a model checking problem is new. The criteria are then evaluated and compared on a set of example specifications. Interestingly, even very simple techniques can achieve high coverage of more complex techniques. Furthermore, there is no clear winner in terms of a single best technique, but instead of focusing on a single technique the flexibility of the model checking approach allows one to combine several different techniques. This makes it possible to apply a best effort strategy, where infeasible test requirements of a strict criterion are automatically replaced by weaker ones. By creating test cases only for uncovered test requirements the number of test cases stays reasonably small even when combining many techniques.

2. Test Case Generation with Model Checkers

This section summarizes how test cases can be generated systematically with a model checker, and defines a range of testing techniques together with their description as a model checking problem.

A model checker is an automatic verification tool that

[*] 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).

[†] Supported by the PRIN Italian MIUR project *D-ASAP: Architetture Software Adattabili e Affidabili per Sistemi Pervasivi*

takes as input an automaton based model and a temporal logic property. The state space of the automaton is explored in order to determine whether the property holds. If a property violation is found, then a counterexample is generated, which is an execution sequence that leads to the property violation. Some model checkers also support witness generation to show how a property holds. Under certain constraints (e.g., determinism and observability) counterexamples and witnesses can be interpreted as test cases which provide test data as well as the expected result (test oracle).

2.1. Transition system specifications

We assume that the model from which the test cases shall be derived is given as a formal specification suitable for the transition system framework introduced by Heimdahl et al. [15]. Examples of such specification languages used previously for test case generation with model checkers are SCR [12], RSML^{-e} [15], or ASM [13], and most state based specifications can be mapped to this framework.

The system state is uniquely determined by the values of n variables $\{x_1, x_2, \ldots, x_n\}$. Each variable x_i has a domain D_i, and consequently the reachable state space of a system is a subset of $D = D_1 \times D_2 \times \ldots \times D_n$. The set of initial values for the variables is defined by a logical expression ρ. The valid transitions between states are described by the transition relation, which is a subset of $D \times D$. The transition relation is defined separately for each variable using logical conditions. For variable x_i, the condition $\alpha_{i,j}$ defines the possible pre-states of the j-th transition, and the condition $\beta_{i,j}$ represents the j-th post-states. The j-th simple transition $\delta_{i,j}$ for a variable x_i is a conjunction of $\alpha_{i,j}$, $\beta_{i,j}$ and the guard condition $\gamma_{i,j}$: $\delta_{i,j} = \alpha_{i,j} \wedge \beta_{i,j} \wedge \gamma_{i,j}$.

The disjunction of all simple transitions for a variable x_i is a complete transition δ_i. The transition relation Δ is the conjunction of the complete transitions of all the variables $\{x_1, \ldots, x_n\}$, resulting in a basic transition system:

Definition 1 (Basic Transition System) *[15] A transition system M over variables $\{x_1, \ldots, x_n\}$ is a tuple $M = (D, \Delta, \rho)$, with $D = D_1 \times D_2 \times \ldots \times D_n$, $\Delta = \bigwedge_{i=1}^{n} \delta_i$, and the initial state expression ρ. For each variable x_i there is a transition relation δ_i, which is given by the disjunction of all the simple transition conditions $\delta_{i,j}$ defined for x_i, where $\delta_{i,j} = \alpha_{i,j} \wedge \beta_{i,j} \wedge \gamma_{i,j}$. The conditions $\alpha_{i,j}$, $\beta_{i,j}$, and $\gamma_{i,j}$ are pre-state, post-state, and guard of the j-th simple transition of variable x_i, respectively.*

Definition 2 (Test Case) *A test case $t := \langle s_0, \ldots s_l \rangle$ for transition system $M = (D, \Delta, \rho)$ is a finite sequence such that $\forall\ 0 \leq i < l : \Delta \models (s_i, s_{i+1})$ and $s_0 \in D$.*

2.2. Test case generation as a model checking problem

Automated test case generation requires formalization of the test objective (e.g., satisfaction of a coverage criterion), which can be seen as a set of test requirements (e.g., one test requirement for each coverable item). Each test requirement can be formalized as a temporal logical *test predicate*.

In this paper we use the temporal logic CTL [9] (Computation Tree Logic). CTL formulas consist of atomic propositions, logical operators, and temporal operators preceded by path quantifiers (**A**, **E**). '**X**' (*next*) expresses that a condition has to be true in the next state. '**G**' (*always*) requires that a condition holds at all states of a path, and '**F**' (*eventually*) requires a condition to eventually hold at some time in the future. '**U**' is the *until* operator, where 'a **U** b' means that a has to hold from the current state up to a state where b is true. The path quantifiers '**A**' ("all") and '**E**' ("some") require formulas to hold on all or some paths, respectively.

Definition 3 (Test predicate) *A test predicate ϕ is a temporal logical predicate that is satisfied by a test case $t = \langle s_0, \ldots s_l \rangle$ if there is $0 \leq i < l$ such that $t^i \models \phi$.*

Here, t^i denotes the subpath of t starting at state s_i, and $t \models \phi$ denotes satisfaction of test predicate ϕ by test case t. A test predicate is *infeasible* if it cannot be satisfied by any possible test case.

Definition 4 (Coverage Criterion) *A coverage criterion C is a rule for generating test predicates on a transition system M. $C(M)$ denotes the set of test predicates obtained by applying C to M.*

Definition 5 (Coverage Satisfaction) *A test suite T satisfies a coverage criterion C on transition system M if and only if for each feasible test predicate $\phi \in C(M)$, there exists a test $t \in T$ such that $t \models \phi$.*

Definitions 4 and 5 satisfy the basic testing adequacy criteria axioms presented in [23]. In addition to coverage satisfaction we use the percentage of satisfied feasible test predicates as *coverage value* to quantify the degree of coverage.

Model checking allows generation of finite paths for logical predicates, either showing satisfaction (*witness*) or violation (*counterexample*). To do so, the test predicates have to be formalized as temporal logic formulas. In most cases counterexample generation is exploited because it is supported by all model checking techniques. In order to force the model checker to generate a counterexample the test predicates are formalized in a negated fashion, such that the counterexample satisfies the original test predicate. Such properties are known as *trap properties*:

Definition 6 (Trap property) *A trap property τ for test predicate ϕ is a temporal logic property, such that any counterexample t to τ, i.e., $t \not\models \tau$ is a test case that satisfies ϕ.*

In the simplest case, a trap property for test predicate ϕ can be derived by stating that ϕ is never satisfied:

$$\mathbf{AG}\ \neg\phi \tag{1}$$

A counter example of (1) contains a state in which ϕ is satisfied. However, if the test predicate ϕ refers only to a transition guard, then the test obtained from the counter example does not execute the transition in the case the transition guard is true just in the last state of the counter example. Execution of the transition might be necessary in order to allow observation of whether the transition guard evaluated to the correct value and it brought the system to the right next state. For this reason, we propose to use as trap property the following formula, which leads to an additional transition at the end of the counterexample:

$$\mathbf{AG}\ (\phi \rightarrow \mathbf{AX}\ \mathit{False}) \tag{2}$$

The implication on *False* in the next state serves as a trick to force the counterexample generation: A counter example of (2) contains a state in which the left side of the implication ϕ is true, and the transition is executed, i.e. (2) is still a valid trap property and guarantees the transition observability.

A test criterion may generate infeasible test predicates. While in the general case feasibility is not decidable, a model checker for finite state systems can decide whether a test predicate is infeasible: Trap properties for infeasible test predicates do not result in counterexamples.

2.3. Logic expression coverage criteria

In this section we describe coverage criteria for logical expressions, using the nomenclature for logical expressions (i.e., a logical *predicate* consists of atomic *clauses* connected by logical operators) and coverage criteria given in [2]. The logical coverage criteria are applied to the guard conditions of simple transitions.

Predicate Coverage (PC) Predicate coverage for a simple transition α, β, γ requires γ to evaluate to true and false, and is described by the following two test predicates:

$$\phi_1 = \alpha \wedge \gamma \tag{3}$$
$$\phi_2 = \alpha \wedge \neg\gamma \tag{4}$$

Clause Coverage (CC) Clause coverage for a simple transition α, β, γ requires two test predicates for each clause C in the guard γ, such that C evaluates to both true and false:

$$\phi_1 = \alpha \wedge C \tag{5}$$
$$\phi_2 = \alpha \wedge \neg C \tag{6}$$

Complete Clause Coverage (CoC) Complete clause coverage requires that all possible valuations for the clauses of a logical predicate are covered. For a simple transition α, β, γ complete clause coverage requires a test predicate for every possible combination of truth values for the clauses C_i in the guard γ, where v_i can be either *True* or *False*:

$$\phi = \alpha \wedge \bigwedge_i (C_i = v_i) \tag{7}$$

General Active Clause Coverage (GACC) General active clause coverage requires that for each clause in a logical predicate there exists a state such that the clause *determines* the value of the predicate, and the clause has to evaluate to true and false. As shown in [2], a clause C determines a predicate P if the following xor-expression is true, where $P_{C,x}$ denotes P with C replaced with x:

$$P_{C,\mathit{True}} \oplus P_{C,\mathit{False}}$$

Consequently, GACC for a simple transition α, β, γ is achieved by two test predicates for each clause in γ:

$$\phi_1 = \alpha \wedge (\gamma_{C,\mathit{False}} \oplus \gamma_{C,\mathit{True}}) \wedge C \tag{8}$$
$$\phi_2 = \alpha \wedge (\gamma_{C,\mathit{False}} \oplus \gamma_{C,\mathit{True}}) \wedge \neg C \tag{9}$$

Correlated Active Clause Coverage (CACC) As a stricter variant of GACC, CACC adds the requirement that not only the clause but also the predicate itself has to evaluate to true and false. Here we hit a small problem of the temporal logic encoding: We do not know whether C will cause the predicate to evaluate to true or to false, consequently we need test predicates that refer to the values in the corresponding other test predicate; this is not directly possible with CTL.

A solution is to add a Boolean helper variable ψ to the specification (alternatively we could use four instead of two test predicates, one for each combination of truth values for C and the guard). The initial value of ψ is non-deterministically chosen (e.g., by the model checker), but cannot change during the execution.

$$\phi_1 = \alpha \wedge (\gamma_{C,\mathit{False}} \oplus \gamma_{C,\mathit{True}}) \wedge C \wedge \gamma = \psi \tag{10}$$
$$\phi_2 = \alpha \wedge (\gamma_{C,\mathit{False}} \oplus \gamma_{C,\mathit{True}}) \wedge \neg C \wedge \gamma \neq \psi \tag{11}$$

As ϕ_1 and ϕ_2 are correlated, both test predicates need to be contained in a single trap property. In the simplest case this is achieved by a disjunction of the two corresponding trap properties, but in practice an implication is usually preferable because of specifics of some counterexample generation algorithms (e.g., $\mathbf{AG}\ (\phi_1 \rightarrow \mathbf{AX}\ \mathbf{AG}\ \neg\phi_2)$).

A drawback of this solution is that a test case for a given clause is only generated if both ϕ_1 and ϕ_2 are feasible. However, this drawback can be compensated by combining test predicates for several criteria in a best effort approach, as described below.

Restricted Active Clause Coverage (RACC) As a further requirement to CACC, RACC requires that for every considered clause (major clause) the remaining clauses (minor clauses) have to have the identical truth values in both cases. As shown in [19], we need additional helper variables ψ_i to express the corresponding test predicates:

$$\phi_1 = \alpha \wedge \bigwedge_{j \neq i} (C_j = \psi_j) \wedge C_i = \psi_i \wedge \gamma = \psi_\gamma \tag{12}$$

$$\phi_2 = \alpha \wedge \bigwedge_{j \neq i} (C_j = \psi_j) \wedge C_i \neq \psi_i \wedge \gamma \neq \psi_\gamma \tag{13}$$

As for CACC these two test predicates are not independent, and therefore result in a single trap property. The number of necessary helper variables equals the maximum number of clauses in the transition guards of the specification. As these are Boolean variables and the number of clauses in a predicate is usually not very big, this does not affect scalability in general. GACC, CACC, and RACC are three different flavors of the important MCDC [7] criterion.

General Inactive Clause Coverage (GICC) In contrast to GACC, GICC requires a clause to *not* determine the outcome of the predicate. This simply means that we have to consider the negation of the expression that describes determination. In addition, we can now distinguish between four different cases: The clause can evaluate to true and false, and in both cases the predicate can evaluate to true and false. Consequently, there are four test predicates for each clause C in the guard γ of a simple transition α, β, γ:

$$\phi_1 = \alpha \wedge \neg(\gamma_{C,False} \oplus \gamma_{C,True}) \wedge C \wedge \gamma \quad (14)$$

$$\phi_2 = \alpha \wedge \neg(\gamma_{C,False} \oplus \gamma_{C,True}) \wedge \neg C \wedge \gamma \quad (15)$$

$$\phi_3 = \alpha \wedge \neg(\gamma_{C,False} \oplus \gamma_{C,True}) \wedge C \wedge \neg\gamma \quad (16)$$

$$\phi_4 = \alpha \wedge \neg(\gamma_{C,False} \oplus \gamma_{C,True}) \wedge \neg C \wedge \neg\gamma \quad (17)$$

Restricted Inactive Clause Coverage (RICC) The restricted variant of GICC again adds the requirement that the values for the minor clauses have to be identical for the case where the predicate evaluates to true and for the case where the predicate evaluates to false:

$$\phi_1 = \alpha \wedge \bigwedge_{j \neq i} (C_j = \psi_j) \wedge C_i = \psi_i \wedge \gamma = True \quad (18)$$

$$\phi_2 = \alpha \wedge \bigwedge_{j \neq i} (C_j = \psi_j) \wedge C_i \neq \psi_i \wedge \gamma = True \quad (19)$$

$$\phi_3 = \alpha \wedge \bigwedge_{j \neq i} (C_j = \psi_j) \wedge C_i = \psi_i \wedge \gamma = False \quad (20)$$

$$\phi_4 = \alpha \wedge \bigwedge_{j \neq i} (C_j = \psi_j) \wedge C_i \neq \psi_i \wedge \gamma = False \quad (21)$$

Similar to RACC and CACC the pairs of test predicates are not independent, and therefore each pair (ϕ_1, ϕ_2) and (ϕ_3, ϕ_4) results in a single trap property. GICC and RICC are two flavors of the RCDC [21] extension to MCDC.

Mutation (M) In general, mutation describes the systematic application of small changes to programs or specifications in order to evaluate test sets or generate test cases. Mutation can be used to derive test cases with model checkers (e.g., [3, 11]).

Let $Mutants(P)$ be the set of mutant expressions derived from expression P with a given set of mutation operators. As shown in [11], a mutant M and the original expression P evaluate to different values if $M \oplus P$. Consequently, for a simple transition (α, β, γ) we get one test predicate for each mutant M in $Mutants(\gamma)$:

$$\phi = \alpha \wedge (M \oplus \gamma) \quad (22)$$

2.4. Transition system coverage

The criteria described so far applied logical coverage criteria to the guard conditions of simple transitions. Now we turn to criteria that view pairs of transitions.

Transition Pair Coverage (TP) Transition pair coverage requires that all sub-paths of length 2 are covered. To express this as test predicates we need to use temporal operators. For each pair of simple transitions $(\alpha_1, \beta_1, \gamma_1)$ and $(\alpha_2, \beta_2, \gamma_2)$ there is a test predicate:

$$\phi = (\alpha_1 \wedge \gamma_1) \wedge \mathbf{EX}\ (\alpha_2 \wedge \gamma_2) \quad (23)$$

All Definitions (AllDef) Data-flow testing with model checkers has been introduced by Hong et al. [16] in the context of control flow graphs. Recently this has been transferred to abstract state machines by Cavarra [6], and can therefore be used for any specification that fits the transition system definition given earlier: A variable is *defined* in a simple transition (α, β, γ) if it occurs in the next state expression β, and it is *used* if it occurs in the guard condition γ. A definition clear path from a variable definition to a variable use is a path where no guard for a simple expression of the variable evaluates to true.

AllDef requires for each variable definition that a definition-clear path to *any* use is covered. Let $def(x) = \alpha_d \wedge \gamma_d$ be a definition of x at simple transition $(\alpha_d, \beta_d, \gamma_d)$, $use(x) = \alpha_u \wedge \gamma_u$ be a use of x at simple transition $(\alpha_u, \beta_u, \gamma_u)$, $defs(x) = \bigvee_i (\alpha_i \wedge \gamma_i)$ the disjunction of all definitions of x, and $uses(x) = \bigvee_i (\alpha_i \wedge \gamma_i)$ the disjunction of all uses of x. AllDef coverage for variable x and definition $def(x)$ is covered by the following test predicate:

$$\phi = def(x) \rightarrow \mathbf{EX}\ \mathbf{E}[\neg defs(x)\ \mathbf{U}\ uses(x)] \quad (24)$$

All Definition-Use Pairs (AllDU) The All-Definitions coverage criterion requires for each variable x and for each definition $def(x)$ that a definition-clear path to *every* use $use(x)$ is covered:

$$\phi = def(x) \rightarrow \mathbf{EX}\ \mathbf{E}[\neg defs(x)\ \mathbf{U}\ use(x)] \quad (25)$$

2.5. Combinatorial coverage

As the number of possible combinations for input variables is usually very large, combinatorial testing aims to provide high coverage with few test cases by requiring only all pairs or n-tuples of input values to be covered. In the case of transition system specifications, we can extend this type of coverage to all system variables. Combinatorial testing with model checkers has been considered in [5, 18].

State Coverage (SC) For each variable V and value v:

$$\phi = (V = v) \quad (26)$$

Pairwise Coverage (PWC) For each variable V_1 and value v_1 and V_2 with value v_2:

$$\phi = (V_1 = v_1 \wedge V_2 = v_2) \quad (27)$$

3-Way Coverage (3WC) For each variable V_1 and value v_1 and V_2 with value v_2 and V_3 with value v_3:

$$\phi = (V_1 = v_1 \wedge V_2 = v_2 \wedge V_3 = v_3) \quad (28)$$

We have presented the basic case here, which can result in a large number of test predicates. The number of test predicates can be reduced; for more details we refer to [5].

2.6. Requirements coverage

The traditional use of model checkers is to verify properties on specifications. In many cases such properties also exist for the specifications that are used to generate test cases. As these properties often represent user requirements it makes sense to use them for test case generation.

Property Coverage (VC) [1] Tan et al. [20] derived a property coverage criterion based on vacuity. A property is vacuously satisfied, if the model checker reports that the property is satisfied regardless of whether the model really fulfills what the specifier originally had in mind or not. For example, the property $\mathbf{AG}\ (x \rightarrow \mathbf{AX}\ y)$ is vacuously satisfied by any model where x is never true. A non-vacuous pass is a useful test case as it illustrates property satisfaction in an 'interesting' way.

A clause C of a property θ is vacuously satisfied if it can be replaced by any other sub-formula without changing the truth value of the formula on any model. Tan et al. [20] make use of the fact that to show vacuity it is sufficient to replace a sub-formula with $True$ or $False$, depending on its *polarity*. The polarity of a C is positive, if it is nested in an even number of negations in θ, otherwise it is negative. Let $pol(C)$ be a function such that $pol\,(C) = False$ if C has positive polarity in θ and $pol\,(C) = True$ otherwise.

Property coverage requires for every clause C of a property θ that there is a test case that non-vacuously satisfies C, i.e., there has to be a state where $\theta_{C,\, pol\,(C)}$ evaluates to false. This results in the following test predicate:

$$\phi = \neg(\theta_{C,\, pol\,(C)}) \quad (29)$$

Unique First Cause Coverage (UFC) Whalen et al. [22] extend the idea of GACC to temporal logic properties. A clause c is the unique first cause of a formula A, if in the first state along a path where A is satisfied, it is satisfied *because* of c. For example, in a sequence $\langle(\neg a, \neg b), (a, \neg b), (\neg a, b)\rangle$, the property $\mathbf{AF}\ (a \vee b)$ is true because a is true in the second state and b is true in the third state, but a is its unique first cause. Test predicates can be derived by applying a set of rules [22] which we cannot reproduce here because of space limitations.

[1] Denoted as vacuity coverage (VC) to distinguish it from predicate coverage (PC).

3. Experimental Evaluation

Given the choice of criteria presented in the previous section it is not an easy task to choose a suitable criterion for a given task. This choice will be influenced by several factors: How many test cases does it take to satisfy the criterion? How many test predicates are infeasible, thus consuming time during test case generation without actually contributing to a test suite? How good are the test cases generated for a given criterion expected to be at detecting faults, based on coverage and mutation score measurement? In order to offer some guidance for the choice we have performed a set of experiments on four different example specifications.

Specifications: The Safety Injection System[4] (SIS) specification models the control of coolant injection in a nuclear power plant. SIS is an example of a classical reactive systems which monitors some integer inputs and controls few critical outputs: in order to adequately test them, one should choose the right values of the inputs in possible big intervals. The Cruise Control (CCS) specification models a simple automotive cruise control based on [17]. LC is a logic controller specification [14]. CCS and LC do not have integer inputs and the internal logic of the controller is the most critical part to be tested. Finally, Windscreen Wiper (WIPER) 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. 23 informal requirements have been formalized in temporal logics; as these are the only realistic requirement properties available to us the criteria VC and UFC are only evaluated here.

Tools: For each of the four specifications we generated test suites for all of the criteria described in the previous section using the symbolic model checker NuSMV [8]. The specifications are given in NuSMV syntax, and trap properties were automatically derived from NuSMV specifications using a set of Python scripts. Test suites were generated calling NuSMV on each trap property using NuSMV's command line interface.

Experiments: The first step of our experiments consisted of generating a test suite for each test criterion for each specification, which also reveals infeasible test predicates. Then the resulting test suites were minimized using a simple heuristic, and for all test suites we then calculated the coverage with regard to all other criteria.

A test suite is minimal with regard to an objective if removing any test case from the test suite will lead to the objective no longer being satisfied. The motivation for using minimized test suites for analysis is to reduce the bias caused by the underlying test case generation technique;

Table 1. Number of trap properties, infeasible trap properties, and minimal number of test cases.

Criterion	CCS			SIS			LC			WIPER			% Infeasible
	Σ	Inf.	Tests	Σ	Inf.	Tests	Σ	Inf.	Tests	Σ	Inf.	Tests	Avg./Spec
PC	24	0	7	26	1	4	39	1	1	164	0	15	1.6%
CC	88	0	2	60	2	3	92	0	2	472	1	11	0.9%
CoC	264	141	22	72	19	6	148	67	5	752	103	56	34.7%
GACC	88	13	16	60	6	6	92	11	3	472	22	36	10.3%
CACC	44	13	10	30	6	4	46	11	2	236	22	26	20.7%
RACC	44	13	10	30	8	5	46	11	2	236	22	26	22.4%
GICC	176	88	3	120	69	4	184	107	2	944	457	17	53.5%
RICC	88	44	4	60	38	2	92	61	2	472	239	10	57.6%
M	476	125	15	295	111	5	764	127	5	3128	716	37	25.9%
TP	132	98	21	156	54	18	380	235	14	6642	4977	133	61.4%
AllDef	13	0	8	13	0	5	28	2	1	46	40	4	23.5%
AllDU	156	110	32	81	45	7	186	0	0	2196	1961	50	53.8%
SC	24	3	2	13	0	2	26	8	2	34	0	6	10.8%
PWC	654	78	12	140	2	3	538	294	3	990	56	64	18.4%
3WC	1396	580	24	192	4	8	3080	2840	3	4080	860	267	39.2%
VC	0	0	0	0	0	0	0	0	0	113	18	10	15.9%
UFC	0	0	0	0	0	0	0	0	0	113	27	16	23.9%

e.g., generating one test case for each test predicate using depth-first search is likely to result in a test suite consisting of long test cases, which will intuitively achieve higher cross coverage than a test suite generated with breadth first search. The symbolic model checker NuSMV generates short, but not necessarily the shortest possible test cases.

The problem of finding the optimal (minimal) subset is NP-hard, which can be shown by a reduction to the minimum set covering problem. In this experiment, we used a simple greedy heuristic to the minimum set covering problem for test suite minimization: The heuristic selects the test case that satisfies the most test predicates and remove all test predicates satisfied by that test case. This is repeated until all test predicates are satisfied.

A single test case will usually cover more than one test requirement, and so it is not strictly necessary to generate test cases for all test predicates. By monitoring the test predicates during test case generation it is possible to generate test cases only for those test predicates that are not already covered; this can greatly reduce the test suite size. Minimization requires existing, full test suites while monitoring checks test predicates on the fly during test case generation. On the other hand, monitoring does not guarantee minimal test suites. While the monitored test suites do not add much to a discussion using full and minimized test suites as they simply lie somewhere in between, we are interested in this approach when combining several techniques or using a best effort approach. To get an intuition of how many calls to a model checker are necessary to satisfy the different techniques we also created test suites using monitoring and different combinations of test criteria.

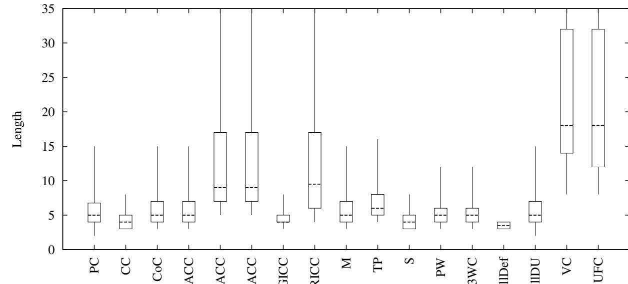

Figure 1. Test case length, Wiper example.

4. Results

Table 1 lists statistics about the specifications and criteria used in the evaluation. For each specification and criterion the table shows how many trap properties were generated in total (Σ), how many of them were infeasible (Inf.), and the minimal number of test cases out of those generated for a criterion necessary to satisfy the criterion. The total number of test cases generated for a criterion (i.e., without minimization) equals the total number of trap properties minus the number of infeasible trap properties, as each feasible trap property results in a test case. The number of trap properties for GACC is twice the number for CACC and RACC because for the latter criteria one trap property combines two test predicates. However, the number of infeasible trap properties is not twice as high because a trap property for CACC and RACC is already infeasible if one of the combined test predicates is infeasible (same for GICC/RICC).

Table 2. Cross coverage for minimized test suites (in percent %).

	PC	CC	CoC	GACC	CACC	RACC	GICC	RICC	M	TP	AllDef	AllDU	SC	PWC	3WC	VC	UFC
PC	-	98	71	88	80	80	89	93	90	41	86	48	98	85	66	17	14
CC	83	-	69	68	52	51	97	83	67	21	62	29	99	78	58	17	13
CoC	100	100	-	100	100	100	100	100	98	48	92	52	100	96	82	21	18
GACC	100	100	92	-	100	100	100	98	98	42	92	50	100	92	77	20	17
CACC	100	100	89	98	-	100	100	98	97	46	94	55	100	92	77	20	17
RACC	100	100	90	98	100	-	100	98	97	48	94	56	100	92	79	20	17
GICC	85	100	72	72	59	58	-	91	70	25	73	34	100	83	63	18	13
RICC	91	100	80	80	70	70	99	-	80	35	77	41	100	90	73	17	14
M	100	100	91	98	100	100	99	100	-	48	92	53	100	93	78	20	18
TP	100	100	85	94	90	90	97	99	95	-	98	76	99	96	86	18	15
AllDef	92	95	65	76	67	66	88	87	78	39	-	40	96	80	66	12	08
AllDU	75	75	60	69	66	66	73	73	71	56	75	-	74	69	54	24	18
SC	80	95	60	60	47	47	85	76	59	19	58	26	-	77	56	14	11
PWC	92	100	85	82	73	73	98	97	81	35	78	39	100	-	88	18	15
3WC	93	100	90	84	79	79	99	100	83	41	79	43	100	100	-	20	17
VC	93	97	71	79	65	64	93	96	84	32	100	59	97	85	52	-	79
UFC	96	99	78	86	80	80	96	99	89	35	100	61	100	91	61	100	-

Some criteria are particularly susceptible to infeasible test predicates, e.g., CoC, M, TP, AllDU, 3WC. The criteria AllDef and AllDU can result in very long test predicates, therefore some of the infeasible test predicates result from a limitation in the length of properties that NuSMV accepts.

Figure 1 illustrates the test case length (minimum, 1st quartile, median, 3rd quartile, and maximum) for the Wiper example. Some outliers for maximum length are truncated to improve readability. The results for the other specifications are similar and omitted for space reasons.

To save space we present the cross coverage only for the minimized test suites averaged for all four specifications in Table 2. Each row represents the test suites created for the criterion specified in the first column, and the remaining columns list the average coverage values of these test suites for the other criteria. Besides offering a direct comparison of the criteria this table also illustrates subsumption relations between different criteria. A criterion x subsumes another criterion y, if any test suite that satisfies x also satisfies y. For example, PWC subsumes SC, and 3WC subsumes PWC and SC, or GACC subsumes PC and CC, and CoC subsumes PC, CC, and all ACC and ICC variants. Subsumption is sensitive to infeasible test predicates; e.g., CACC does not achieve 100% GACC in the table because there exist clauses that can be covered with GACC but not with CACC. Interestingly, GACC achieves 100% coverage for CACC and RACC in our experiments, which supports the view that GACC is the preferred ACC variant. For more details about the subsumption relations between the presented criteria we refer to [2, p.113].

Table 3 shows the numbers of test cases for several different combinations of criteria, and the overall coverage considering all criteria (except VC and UFC). The combinations shown in the table were chosen out of the large number of possible combinations because they represent typical or useful combinations. Monitoring was applied during test case generation, and therefore the experiment was repeated 10 times with different random orderings of the test predicates within a criterion, and the results are averaged. ACC denotes the combination of RACC, CACC, GACC, CC, and PC, and ICC combines the ICC variants.

Table 3. Test cases and total coverage for combined criteria.

Combination	Number of test cases				Cov.
	CCS	SIS	LC	WIPER	
PC+CC	8.3	4.7	3.3	21.7	81.2%
ICC	10.1	5.6	2.3	22.3	85.0%
ACC	13.7	6.7	4.0	29.0	87.2%
ACC+ICC	17.7	7.0	5.2	41.7	87.3%
SC+PWC+3WC	34.7	12.0	5.3	315.0	89.7%
AllDef+D/CC+PWC	17.0	11.0	5.0	77.7	90.2%
TP+AllDef/DU	33.6	27.8	21.0	174.7	93.7%
AllDU+A/ICC+PWC	40.0	14.6	7.3	111.6	94.5%
M+AllDU+TP+PWC	45.0	30.7	20.0	215.3	97.9%
___+ACC	44.3	31.0	21.0	219.0	98.1%
All criteria	62.7	33.0	21.0	413.0	100%

5. Discussion

Based on the data collected in our experiments this section contains general conclusions. In particular, we try to

highlight advantages and disadvantages of the various criteria, and discuss how the criteria can be used in conjunction.

5.1. Which criterion is preferable?

There is no definite answer to the question which criterion is preferable, as this question is influenced by many different factors: For example, the testing process might be regulated by a standard posing requirements on the test criteria, and the resources available for test case generation or execution might be limited. Consequently, we analyze the different criteria in terms of how complex they are and how well they fare in satisfying other criteria.

First of all, the number of test predicates a criterion represents gives an indication of how difficult it is to satisfy the criterion, and how many test cases result from the criterion. Table 1 shows that criteria that are intuitively simpler to satisfy result in fewer test cases and also in fewer infeasible test predicates. PC, SC, and AllDef result in the least number of test predicates, and many of the other criteria result in significantly more test predicates. For combinatorial coverage the number of test predicates quickly increases with the size of combinations considered, therefore it does not come as a surprise that 3WC results in the largest number of test predicates, only sometimes topped by M. This is also because we applied combinatorial testing to all system variables, not only input variables as is often done. CC and the various ACC and ICC flavors are all in a similar range, GICC being an exception because there are four test predicates for every condition. CoC, TP, M, and AllDU coverage usually result in the largest number of test predicates, together with t-way coverage.

The number of infeasible test predicates is important for several reasons: Even when using monitoring or just extending existing test suites, for each infeasible test predicate it has to be attempted to generate a test case in order to detect whether it is infeasible. In addition, only exhaustive verification can determine infeasibility. A test criterion producing a large number of infeasible test predicates could therefore become inapplicable to large specifications if the number of inconclusive model checker runs is too high. In contrast, if non-exhaustive techniques are used (e.g., bounded model checking or explicit state model checking with limited memory) infeasibility cannot always be proven. TP and AllDU result in the largest number and percentage of infeasible test predicates (61,4% and 71,8% on average, respectively), while the simplest criteria (PC, CC, AllDef, SC) result in few infeasible test predicates (1,6%, 0,9%, 23,7%, 10,5% on average). The ICC criteria result in more infeasible test predicates than the corresponding ACC criteria. For 3WC the large number of infeasible test predicates is in large part because we considered all system variables, while traditionally combinatorial testing only considers input variables and results in only few infeasible test predicates. While it is commonly agreed that CoC results in too many test cases to be useful this is not the case in our experiments; however, the number of infeasible test predicates is rather high (34.7% on average).

Considering the length of the test cases, Figure 1 shows that CACC, RACC, and RICC have significantly longer test cases; this is because of the combined test predicates. The very long test cases of VC and UFC are influenced by the underlying requirement properties. The shortest test cases result for AllDef, SC, and CC. In general, the length of test cases for combinatorial coverage is lowest, and the logical criteria PC, CC, CoC, GACC, and GICC all have comparable length. The time to generate a test case (not shown here) mostly reflects the length; interestingly it takes significantly longer to generate test cases for AllDef, AllDU, VC, and UFC than for all other criteria, which is due to the complexity of these test predicates.

Finally, considering the cross coverage Table 2 reveals that the simple criterion PC can achieve quite good coverage with regard to most other criteria, even though it results in very few test cases. This supports the view that typically coverage criteria consist of many 'easy' test requirements and only few 'difficult' ones. Other simple criteria (CC, SC) do not fare quite so well, and are also covered by most other criteria. VC and UFC are very difficult to cover for non-requirements based criteria, while VC and UFC have quite good coverage of all other criteria, but this depends on the underlying properties. TP is also very difficult to cover by other criteria, while it achieves high coverage of all criteria except VC and UFC. Mutation achieves very high coverage on all criteria that consider single transition guards, but coverage of criteria considering several transitions or requirements is not so good. Some criteria show a higher coverage of RICC than GICC, this is because RICC has more infeasible test predicates. PWC and 3WC are also not very easy to cover, but their own coverage on other criteria is worse than that of most criteria based on transition guards. Finally, AllDef and AllDU are difficult to cover for transition guard based criteria, but their own coverage is often even worse than that of combinatorial criteria. Consequently, we recommend to use a mix of criteria from all groups.

Simply looking at the coverage values would result in a preference for criteria that simply result in larger test suites. In practice, resources for testing are usually limited, and so test cases should preferably maximize coverage while being as short as possible. To this extent we consider the ratio of the mutation score to the total test suite length (but only summarize the results here due to space restrictions). In most cases, AllDef has the highest ratio of mutants to total length. In addition, this criterion results in a reasonable number of test predicates out of which only few are infeasible. Therefore, this criterion is a very useful criterion and a good starting point; its only drawback is the com-

plexity of the test predicates. PC, CC, and SC also have very high ratios of killed mutants to total length but have the advantage of much simpler test predicates. The ICC criteria have a higher ratio than the ACC criteria, with GACC and GICC being the best criteria out of these groups. M and GACC have similar ratios. In this comparison 3WC performs worst; considering the large number of feasible and infeasible test predicates this criterion seems only recommendable as an addition if there are sufficient resources. TP also has a low ratio because of the large number of test cases, but it seems that TP is a stricter criterion than mutation so in this case the comparison is probably not fair.

5.2. Combining techniques

One of the main advantages of using model checkers for test case generation is that it is very simple to combine different techniques – it is just a matter of deriving new test predicates, but the framework is identical for all criteria.

When combining different techniques it will often make sense to just add new test cases for uncovered test predicates. An example application would be a *best effort* approach as described in [2]: Many coverage criteria are related through subsumption relations. If a test predicate turns out to be infeasible, the best effort approach is to turn to the next weaker variant of test predicate. This can be applied with model checkers by starting with a complex criterion, and then moving on to the next weaker version, only generating test cases for uncovered test predicates.

Table 3 shows several useful combinations of criteria. The number of test cases resulting from the combination is reasonable in all cases: In fact, generating test cases for *all* criteria results in test suites that are smaller than the full test suites created for most criteria (cf. Table 1). PC+CC is a combination often used in practice, and the coverage is clearly inferior to ACC or ICC criteria. Of all categories the combination of TP, AllDef, and AllDU achieves the highest coverage. Combinations from the different categories achieve the best results: M+AllDU+TP+PWC achieves 97.9% coverage, and adding the ACC criteria brings the coverage up to 99%.

An interesting observation when combining several criteria concerns the order in which test predicates are considered: The numbers in Table 3 are based on test case generation starting with the most complex criterion of a combination. When starting with the simplest criterion the number of unique test cases in the resulting test suite is slightly larger without a significant increase in coverage.

5.3. Threats to validity

Different model checking techniques can result in different counterexamples, which might also influence our findings. We only used the symbolic model checker NuSMV for our experiments. However, as NuSMV creates short test cases we expect the results for bounded model checking and explicit model checking with breadth first search to be similar. Depth first search results in very long test cases, which typically means that less test cases are necessary to satisfy all test predicates of a criterion. While changing some of the experimental results this should have no influence on our general conclusions. A further limitation of our experiments is that the specifications used are realistic but still on the small side; however, we expect that our results in terms of infeasible test predicates, cross coverage, etc. generalize also to larger specifications. Finally, our conclusions make the common assumption that high coverage on a specification is likely to increase fault detection ability, but we did not measure the effects on actual faults.

5.4. Related work

We considered state based specifications in this paper, but model checkers are also useful in other contexts. For example, Hong et al. [16] use model checkers to derive test cases for control flow graphs. Furthermore, we only considered techniques where the test objectives are represented as temporal logic properties but the specification is not altered. Some of the criteria in this paper have been used previously: Heimdahl et al. [15] proposed coverage criteria similar to PC, CoC, and RACC in their transition system framework, while for example the other ACC and ICC flavors are new. Mutation has been previously used (e.g., [3]), and the test predicates we used in this paper are based on [11]. The criteria in this paper are all based on logical expressions in the transition system framework, but it is also possible to define further criteria on the specifics of the actual specification language (e.g., [12, 13]). For a detailed overview of testing with model checkers we refer to [10].

An evaluation of the specification coverage criteria TP, M, and Full Predicate coverage (similar to GACC) was performed by Abdurazik et al. [1]. Many of the weaker logical coverage criteria are subsumed by these three criteria, therefore a combination of these criteria is likely to result in thorough test suites. This result is in line with our findings.

6. Conclusions

In this paper we have represented 17 different testing techniques that can be represented in temporal logic in a common framework, allowing us to unite criteria presented in different frameworks and scenarios and some other criteria not previously used in conjunction with model checker based test case generation. The large number of techniques suitable for such an approach make it difficult to pick a suitable set for a concrete application, therefore we performed a set of experiments to compare the criteria.

The evaluation shows that there is no single superior criterion. We have categorized the criteria into different categories, and it is useful to combine criteria from all of these categories. Such a combination does not necessarily result in large test suites if only test cases for uncovered test pred-

icates are added. In addition to combining techniques from different categories it is also useful to combine related techniques from the same category, as this allows a best effort approach to cope with infeasible test predicates.

As some criteria are complex and result in many feasible and infeasible test predicate, the choice of criteria will often be guided by the available resources. If resources are limited it is reasonable to start with simple criteria that have few test predicates (e.g., AllDef, PC, CC, SC), and usually only few of them are infeasible. This gives high coverage at small costs, and the remaining resources can be used to cover more difficult test objectives. For example, GACC and GICC result in reasonably small test suites, have only few infeasible test predicates, and are generally preferable to their stricter variants. The strictest criteria such as M, TP, AllDU, 3WC all result in many infeasible test predicates, but many of the feasible test predicates are difficult to cover by other criteria, therefore intuitively resulting in 'good' test cases. Even using all criteria in combination is feasible for test case generation, as monitoring can ensure that the test suite size stays reasonably small.

Acknowledgements Many thanks to Paul Ammann for comments on an earlier version of this paper.

References

[1] A. Abdurazik, P. Ammann, W. Ding, and J. Offutt. Evaluation of Three Specification-Based Coverage Testing Criteria. In *Proceedings of the 6th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS 2000)*, pages 179–187. IEEE Computer Society, September 2000.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[3] 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.

[4] R. Bharadwaj and C. L. Heitmeyer. Model Checking Complete Requirements Specifications Using Abstraction. *Automated Software Engineering*, 6(1):37–68, 1999.

[5] A. Calvagna and A. Gargantini. A Logic-Based Approach to Combinatorial Testing with Constraints. In *Tests and Proofs*, volume 4966 of *LNCS*, pages 66–83. Springer, 2008.

[6] A. Cavarra. Data flow analysis and testing of abstract state machines. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *Proceedings of First International Conference Abstract State Machines, B and Z (ABZ 2008)*, volume 5238 of *LNCS*, pages 85–97. Springer, 2008.

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

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

[9] 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.

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

[11] A. Gargantini. Using Model Checking to Generate Fault Detecting Tests. In *Proceedings of the International Conference on Tests And Proofs (TAP)*, volume 4454 of *LNCS*, pages 189–206, 2007.

[12] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests From Requirements Specifications. In *ESEC/FSE'99: 7th European Software Engineering Conf. and Foundations of Software Engineering Symposium*, volume 1687 of *LNCS*, pages 146–162. Springer, 1999.

[13] A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, 2001.

[14] I. Grobelna. Formal verification of logic controller specification using NuSMV model checker. In *X International PhD Workshop OWD'2008*, 2008.

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

[16] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 151–161. Springer, 2002.

[17] J. Kirby. Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System. Technical Report TR-87-07, Wang Inst. of Graduate Studies, 1987.

[18] 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.

[19] S. Rayadurgam and M. P. Heimdahl. Generating MC/DC Adequate Test Sequences Through Model Checking. In *Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop*, pages 91–96, 2003.

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

[21] S. A. Vilkomir and J. P. Bowen. Reinforced Condition/Decision Coverage (RC/DC): A New Criterion for Software Testing. In *Proceedings of the 2nd Int. Conf. on Formal Specification and Development in Z and B (ZB 02)*, pages 291–308, London, UK, 2002. Springer.

[22] M. W. Whalen, A. Rajan, M. P. Heimdahl, and S. P. Miller. Coverage Metrics for Requirements-Based Testing. In *ISSTA'06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 25–36, New York, NY, USA, 2006. ACM Press.

[23] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.