COMPETENCE NETWORK SOFTNET AUSTRIA

Testing with model checkers: A survey

SNA-TR-2007-P2-04

Gordon Fraser, Franz Wotawa, Paul E. Ammann

SNA TECHNICAL REPORT

NOVEMBER 2007

Competence Network Softnet Austria, Inffeldgasse 16c, A-8010 Graz, Austria,
Phone +43-316-873-5711, Fax +41-316-873-5706
sekretariat@soft-net.at, http://www.soft-net.at/

SNA TECHNICAL REPORT

SNA TECHNICAL REPORT SNA-TR-2007-P2-04, NOVEMBER 2007

Testing with model checkers: A survey

Gordon Fraser,[1] **Franz Wotawa**[2] **Paul Ammann**[3]

Abstract. About a decade after the initial proposal to use model checkers for the generation of test cases we take a look at the achievements in this field of research. Model checkers are formal verification tools, capable of providing counterexamples to violated properties. Normally, these counterexamples are meant to guide an analyst when searching for the root cause of a property violation. They are, however, also very useful as test cases. Many different approaches have been presented, many problems have been solved, yet many issues remain. This survey paper reviews the state of the art in testing with model checkers.

[1]Institute for Software Technology, Technische Universität Graz, Inffeldgasse 16c, A-8010 Graz, Austria. E-mail: fraser@ist.tugraz.at

[2]Institute for Software Technology, Technische Universität Graz, Inffeldgasse 16b/2, A-8010 Graz, Austria. E-mail: fwotawa@ist.tugraz.at

[3]Department of Information and Software Engineering, MS 4A4, George Mason University, Fairfax, VA 22030-4444, U.S.A.

1 INTRODUCTION

Testing remains the most important method to verify the quality of software. Automation is necessary, because manual testing takes a lot of effort and is error prone. Callahan et al. [1] and Engels et al. [2] initially proposed the use of model checkers for the automated generation of test cases.

Testing with model checkers is a model-based testing technique. The model-based approach to software testing encompasses the creation of an abstract model, which is used to automatically create test cases. At the same time the model tells us the expected outcome, thus solving the test oracle problem. Research has resulted in numerous different approaches, differing in how models are derived, what formalisms are used to represent the models, how test cases are selected in the model, and many other factors. A recent taxonomy of model-based testing techniques [3] gives an overview of available approaches.

A model checker is a tool used for formal verification. It takes as input an automaton based model of a system and a temporal logic property, and then effectively explores the entire state space of the system in order to determine whether the model violates the property or not. If a property violation is encountered, then a counterexample is returned to illustrate the violation to the analyzer.

While formal verification can prove property violation or satisfaction, it is usually not sufficient in practice. The proof only shows that a given model fulfills a property, while the actual implementation is also influenced by its environment, e.g., platform, compiler, etc. Furthermore, formal verification usually only applies to models of limited size, which means that only abstract models of complex programs can be verified. Consequently, software testing is necessary.

The idea of testing with model checkers is to interpret counterexamples as test cases. The main challenge is to force the model checker to systematically create sets of such counterexamples, which can then be used as a complete test suite.

A decade of research on testing with model checkers has resulted in a multitude of different techniques of how to derive test cases. Many issues related to this test case generation have been tackled, yet there are still some show stoppers, like the dreaded state explosion problem.

Testing with model checkers is mostly applied to reactive systems [4], where the software size is within bounds (e.g., embedded software), and it is feasible to assume the existence of a suitable model. However, model checkers have also been used on other types of systems. In general, anything can be tested with model checkers if there is a suitable model, that is sufficiently abstract to allow verification in realistic time.

This paper takes a look at achievements have been made for testing with model checkers. It is organized as follows: First, Section 2 reviews theoretical background and preliminaries of model checkers and testing with model checkers. Section 3 introduces a running example that is used at several points throughout the paper. Section 4 takes a look at the available techniques of how to use a model checker to create test suites with regard to different coverage criteria, Section 5 considers approaches based on requirement properties, and Section 6 reviews mutation based methods to generate test cases. The main issues in testing with model checkers are discussed in Section 7. Section 8 discusses the use of model checkers to analyze existing test suites, and Section 9 reviews applications of model checkers in the context of software testing that do not fit into the previous sections. Publicly available tools that can be used to generate test cases with model checkers are described in Section 10. Finally, the paper is concluded with a brief discussion of outstanding research issues in Section 11.

2 PRINCIPLES

Basically, a model checker is a tool intended for formal verification. It takes as input an operational specification of the system that is considered. Then, it takes a temporal logic property and analyzes the entire state space of the model in order to determine whether the model violates the property or not. If the state space exploration shows no property violations, then correctness with regard to the property is proved. A basic feature of model checkers is the ability to generate witnesses and counterexamples for property satisfaction or violation, respectively. When a typical model checker detects that a property is violated, it returns a counterexample that illustrates the property violation. A human analyzer can use this counterexample to identify and fix the design fault. For testing purposes, the counterexample can be interpreted as a test case.

2.1 Model Checking Preliminaries

The formalism commonly used to describe model checking and to define the semantics of temporal logics is the Kripke structure.

Definition 1 (Kripke Structure) *A Kripke structure K is a tuple $K = (S, S_0, T, L)$:*

- *S is a set of states.*
- *$S_0 \subseteq S$ is an initial state set.*
- *$T \subseteq S \times S$ is a total transition relation, that is, for every $s \in S$ there is a $s' \in S$ such that $(s, s') \in T$.*
- *$L : S \rightarrow 2^{AP}$ is a labeling function that maps each state to a set of atomic propositions that hold in this state.*

AP is a countable set of atomic propositions.

An infinite execution sequence of this model is a *path*. As paths are infinite, deadlocks cannot directly be modeled with Kripke structures. It is, however, possible to model a deadlock as a self-loop to a state. A Kripke structure defines all possible paths of a system.

Definition 2 (Path) *A path $p := \langle s_0, s_1, ... \rangle$ of Kripke structure K is an infinite sequence such that $\forall\ i \geq 0 : (s_i, s_{i+1}) \in T$ for K.*

Let $Paths(K, s)$ denote the set of paths of Kripke structure K that start in state s. We use $Paths(K)$ as an abbreviation to denote $\{Paths(K, s) \mid s \in S_0\}$.

As infinite paths are not usable in practice, model checking uses finite sequences, commonly referred to as *traces*. If necessary, we can interpret a finite sequence as an infinite sequence where the final state is repeated infinitely.

Definition 3 (Trace) *A trace $t := \langle s_0, ...s_n \rangle$ of Kripke structure K is a finite sequence such that $\forall\ 0 \leq i < n : (s_i, s_{i+1}) \in T$ for K. There can be a dedicated state s_i such that $s_i = s_n$ and $i \neq n$, which is a loopback state, and $\langle s_0, \ldots s_{i-1}, (s_i \ldots s_n)^\omega \rangle$ is a path of K.*

A trace t is either a finite prefix of an infinite path or a path that contains a loop, if a loopback state is given. The latter is called a lasso-shaped sequence, and has the form $t := t_1(t_2)^\omega$, where t_1 and t_2 are finite sequences. The sequence t_2 is repeated infinitely often, denoted with ω, the infinite version of the Kleene star

operator used for ω-languages. Lasso shaped sequences are used in practice to show violation of liveness properties, which requires infinite sequences. For example, the model checker NuSMV [5] interprets all identical states in a trace as possible points of loopback. The number of transitions a trace consists of is referred to as its *length*. For example, trace $t := \langle s_0, s_1, ..., s_n \rangle$ has a length of $length(t) = n$.

Temporal logics are modal logics with special operators for time. Time can either be interpreted to be linear or branching. The most common logics are the linear time logic LTL [6] (Linear Temporal Logic), and the branching time logic CTL [7] (Computation Tree Logic). CTL*, introduced by Emerson and Halpern [8], is the superset of these logics. Most current model checkers support either LTL or CTL, or sometimes both. Other temporal logics that are used in model checking are Hennessy-Milner Logic [9] (HML), Modal μ-calculus [10], and different flavors of CTL such as timed, fair, or action CTL.

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

$$\begin{aligned} \phi \quad ::= \quad & true \mid false \mid a \in AP \mid \neg\, \phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \\ & \phi_1 \rightarrow \phi_2 \mid \phi_1 \equiv \phi_2 \mid \phi_1\ \mathsf{U}\ \phi_2 \mid \bigcirc \phi \mid \Box \phi \mid \Diamond \phi \end{aligned}$$

A property ϕ satisfied by path π of model K is denoted as $K, \pi \models \phi$, which is also abbreviated as $\pi \models \phi$ if K is obvious from the context. A path π of model K violating property ϕ is denoted as $K, \pi \not\models \phi$ or $\pi \not\models \phi$. The semantics of LTL is expressed for infinite paths of a Kripke structure. π^i denotes the suffix of the path π starting from the i-th state, and π_i denotes the i-th state of the path π, with $i \in \mathbb{N}_0$. The initial state of a path π is π_0.

Definition 5 (LTL Semantics) *Satisfaction of LTL formulas by a path $p \in Paths(K)$ of a Kripke Structure $K = (S, S_0, T, L)$ is inductively defined as follows, where $a \in AP$:*

$$\begin{aligned}
& K, \pi \models true && \textit{for all } \pi && (1)\\
& K, \pi \not\models false && \textit{for all } \pi && (2)\\
& K, \pi \models a && \textit{iff} \quad a \in L(\pi_0) && (3)\\
& K, \pi \models \neg\phi && \textit{iff} \quad K, \pi \not\models \phi && (4)\\
& K, \pi \models \phi_1 \wedge \phi_2 && \textit{iff} \quad K, \pi \models \phi_1 \wedge K, \pi \models \phi_2 && (5)\\
& K, \pi \models \phi_1 \vee \phi_2 && \textit{iff} \quad K, \pi \models \phi_1 \vee K, \pi \models \phi_2 && (6)\\
& K, \pi \models \phi_1 \rightarrow \phi_2 && \textit{iff} \quad K, \pi \not\models \phi_1 \vee K, \pi \models \phi_2 && (7)\\
& K, \pi \models \phi_1 \equiv \phi_2 && \textit{iff} \quad K, \pi \models \phi_1 \textit{ iff } K, \pi \models \phi_2 && (8)\\
& K, \pi \models \phi_1\ \mathsf{U}\ \phi_2 && \textit{iff} \quad \exists i \in \mathbb{N}_0 : K, \pi^i \models \phi_2 \wedge \forall\, 0 \leq j < i : K, \pi^j \models \phi_1 && (9)\\
& K, \pi \models \bigcirc \phi && \textit{iff} \quad K, \pi^1 \models \phi && (10)\\
& K, \pi \models \Box \phi && \textit{iff} \quad \forall j \in \mathbb{N}_0 : K, \pi^j \models \phi && (11)\\
& K, \pi \models \Diamond \phi && \textit{iff} \quad \exists j \in \mathbb{N}_0 : K, \pi^j \models \phi && (12)
\end{aligned}$$

The temporal logic CTL was introduced by Clarke and Emerson [7]. It can be viewed as a subset of CTL*, introduced by Emerson and Halpern [8]. CTL* formulas consist of atomic propositions, logical operators, temporal operators (**F** , **G** , **U** , **R** , **X**) and path quantifiers (**A**, **E**). The operator **F** ("finally") corresponds to the eventually operator $\Diamond$ in LTL, **G** ("globally") corresponds to $\Box$. **U** ("until") corresponds to U, and **R** ("release") is the logical dual of **U** . **X** ("next") corresponds to the next operator $\bigcirc$. The path quantifiers **A** ("all") and **E** ("some") require formulas to hold on all or some paths, respectively. CTL* includes all possible combinations of temporal operators with formulas, where the temporal operators do not have to be preceded by path quantifiers. As CTL* model checking is complex, most model checkers use either CTL or LTL in practice. Consequently, we do not consider CTL* in detail. CTL is the subset of CTL* obtained by requiring that each temporal operator is immediately preceded by a path quantifier. Consequently, the syntax of CTL can be defined as follows:

Definition 6 (CTL Syntax) *The BNF definition of CTL formulas is given below:*

$$\begin{aligned}\phi \;:=\; & a \in AP \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \\ & \mathbf{AX}\ \phi \mid \mathbf{AF}\ \phi \mid \mathbf{AG}\ \phi \mid \mathbf{A}\ \phi_1 \mathbf{U}\ \phi_2 \mid \mathbf{A}\ \phi_1 \mathbf{R}\ \phi_2 \mid \\ & \mathbf{EX}\ \phi \mid \mathbf{EF}\ \phi \mid \mathbf{EG}\ \phi \mid \mathbf{E}\ \phi_1 \mathbf{U}\ \phi_2 \mid \mathbf{E}\ \phi_1 \mathbf{R}\ \phi_2\end{aligned}$$

As all temporal operators are preceded by a path quantifier in CTL, the semantics of CTL can be expressed by satisfaction relations for state formulas. $K, s \models \phi$ denotes a state formula ϕ that is satisfied in state s of Kripke structure K.

Definition 7 (CTL Semantics) *Satisfaction of CTL formulas by a state $s \in S$ of a Kripke Structure $K = (S, S_0, T, L)$ is inductively defined as follows, where $a \in AP$:*

$$\begin{aligned}
K, s \models a \quad & \textit{iff} \quad a \in L(s) \wedge s \in S \\
K, s \models \neg\phi \quad & \textit{iff} \quad \neg(K, s \models \phi) \\
K, s \models \phi_1 \vee \phi_2 \quad & \textit{iff} \quad (K, s \models \phi_1) \vee (K, s \models \phi_2) \\
K, s \models \phi_1 \wedge \phi_2 \quad & \textit{iff} \quad (K, s \models \phi_1) \wedge (K, s \models \phi_2) \\
K, s \models \mathbf{AX}\ \phi \quad & \textit{iff} \quad \forall \pi \in Paths(K, s) : K, \pi_1 \models \phi \\
K, s \models \mathbf{AF}\ \phi \quad & \textit{iff} \quad \forall \pi \in Paths(K, s) : \exists i : K, \pi_i \models \phi \\
K, s \models \mathbf{AG}\ \phi \quad & \textit{iff} \quad \forall \pi \in Paths(K, s) : \forall i : K, \pi_i \models \phi \\
K, s \models \mathbf{A}\phi_1 \mathbf{U}\ \phi_2 \quad & \textit{iff} \quad \forall \pi \in Paths(K, s) : \exists i : \forall j < i : K, \pi_j \models \phi_1 \wedge \\
& \qquad \forall k \geq i : K, \pi_k \models \phi_2 \\
K, s \models \mathbf{A}\phi_1 \mathbf{R}\ \phi_2 \quad & \textit{iff} \quad \forall \pi \in Paths(K, s) : \forall i : \forall j < i : K, \pi_j \not\models \phi_1 \rightarrow K, \pi_i \models \phi_2 \\
K, s \models \mathbf{EX}\ \phi \quad & \textit{iff} \quad \exists \pi \in Paths(K, s) : K, \pi_1 \models \phi \\
K, s \models \mathbf{EF}\ \phi \quad & \textit{iff} \quad \exists \pi \in Paths(K, s) : \exists i : K, \pi_i \models \phi \\
K, s \models \mathbf{EG}\ \phi \quad & \textit{iff} \quad \exists \pi \in Paths(K, s) : \forall i : K, \pi_i \models \phi \\
K, s \models \mathbf{E}\phi_1 \mathbf{U}\ \phi_2 \quad & \textit{iff} \quad \exists \pi \in Paths(K, s) : \exists i : \forall j < i : K, \pi_j \models \phi_1 \wedge \\
& \qquad \forall k \geq i : K, \pi_k \models \phi_2 \\
K, s \models \mathbf{E}\phi_1 \mathbf{R}\ \phi_2 \quad & \textit{iff} \quad \exists \pi \in Paths(K, s) : \forall i : \forall j < i : K, \pi_j \not\models \phi_1 \rightarrow K, \pi_i \models \phi_2
\end{aligned}$$

Commonly, three different types of verifiable properties are distinguished:

Safety Property: A safety property describes a behavior that may not occur on any path ("Something bad may not happen"). To verify a safety property, all execution paths have to be checked exhaustively. Safety properties are of the type $\Box \neg \phi$ or $\mathbf{AG}\ \neg \phi$, where ϕ is a propositional formula.

Invariance Property: An invariance property describes a behavior that is required to hold on all execution paths. It is logically complementary to a safety property. Invariance properties are of the type $\Box \phi$ or $\mathbf{AG}\ \phi$, where ϕ is a propositional formula.

Liveness Property: A liveness property describes that "something good eventually happens". With linear time logics, this means that a certain state will always be reached. For example, $\Box \phi_1 \rightarrow \Diamond \phi_2$ and $\mathbf{AG}\ \phi_1 \rightarrow \mathbf{AF}\ \phi_2$ are liveness properties.

The aim of model checking is to determine whether a given model fulfills a given property. Several different algorithms have been successfully used for this task, using different temporal logics and data structures. Once property violation or satisfaction is determined, a model checker can return an example of how this violation or satisfaction occurs. This is illustrated with a *counterexample* or *witness*, respectively. Satisfaction of LTL properties is defined using linear sequences. Consequently, witnesses and counterexamples for LTL formulas are also linear sequences. In contrast, CTL properties are state formulas. Therefore, the CTL model checking problem [11, p. 35] is to find the set of states that satisfy a given formula in a given Kripke structure. Special algorithms are used to derive trace examples for witness or counterexample states [12].

The first successful model checking approach is *explicit model checking*. There are different approaches based on LTL [13, 14] and CTL [15, 16] properties. In all approaches, the state space is represented explicitly, and searched by forward exploration until a violation of a property is found. For example, in LTL model checking the negation of a property is represented as an automaton that accepts infinite words (Büchi automaton). If the synchronous product of model and Büchi automaton contains any accepting path, then this path proves property violation (the path shows that the negation of the property is accepted by the model automaton, and therefore the property itself is violated). The counterexample is simply the path back to the initial state. The search algorithm can either be depth- or breadth-first search; recently heuristic search has also been considered. Breadth-first search always finds the shortest possible counterexamples, but the memory demands are significantly higher than for depth-first search. In CTL model checking, all states satisfying a given property are determined by recursively calculating the satisfied sub-formulas for each state. If all states are visited and no violation is detected, then the property is consistent with the model. *Directed model checking* [17] extends explicit model checking with heuristic search to increase the speed with which errors are found and counterexamples are generated. Such a technique is applicable if the aim of model checking is not a proof of correctness, but the generation of counterexample. As such, this idea is well suited for test case generation.

Symbolic model checking [18], the second generation of model checking, uses ordered binary decision diagrams (BDDs [19]) to represent states and function relations on these states efficiently. This allows the representation of significantly larger state spaces, but a large number of BDD variables has a negative impact on the performance, and the ordering of the BDD variables has a significant impact on the overall size. There are different heuristic approaches of how to order variables, as determining the optimal order is NP-complete [20].

Bounded Model Checking [21], the third generation of model checking, reformulates the model checking problem as a constraint satisfaction problem (CSP). This allows the use of propositional satisfiability (SAT) solvers to calculate counterexamples up to a certain upper bound. As long as the boundary is not too big,

this approach is very efficient. There are also approaches to extend bounded model checking to infinite state systems. Bounded model checking has been successfully applied to systems where traditional model checking fails. At the same time, there are many settings where a bounded model checker fails while a symbolic model checker is efficient. Therefore, bounded model checkers do not replace but supplement traditional model checking techniques.

The most commonly used model checkers in the context of testing are the explicit state model checker SPIN [22] (Simple Promela Interpreter), the Symbolic Analysis Laboratory SAL [23], which supports both symbolic and bounded model checking, the symbolic model checker SMV [24] as well as its derivative NuSMV [5], which supports symbolic and bounded model checking. Other popular model checkers include Murϕ [25] the process algebra based FDR2 [26], or COSPAN [27]; some of these have also been used for testing.

Many current model checkers such as NuSMV [5] or SAL [23] support CTL model checking in addition to or instead of LTL model checking. In CTL model checking, special algorithms are applied to construct linear traces from an initial state to explain a violating state [12]. However, only certain restricted subsets of branching time temporal logics such as $ACTL^{det}$ or LIN always result in linear counterexamples [28]. When using full CTL, linear counterexamples are not always sufficient as evidence for property violation or satisfaction. Most work on testing with model checkers only considers the linear subset when using CTL for properties. Therefore, we use the term *counterexample* to describe a linear trace that either shows an LTL property violation or violation of a CTL property that can be violated by a linear trace.

Recently, an algorithm to create tree-like counterexamples has been proposed by Clarke et al. [29]. In a related work, Wijesekera et al. [30] define a formal relation between test cases and counterexamples for full CTL. This relation is described in Section 7.8. A related approach has been presented by Meolic et al. [31]: Witness and counterexample automata represent the superset of all finite and linear witnesses/counterexamples for a limited subset of CTL.

2.2 Testing with Model Checkers

The idea of testing with model checkers is to interpret counterexamples as test cases. A suitable test case execution framework can extract from this the test data, and also the expected results (i.e., test oracle). Early work on testing with model checkers required manually specified *test purposes* to either formulate negated as *never-claims* [2], or to partition the execution tree [1]. Later, many different techniques were proposed to systematically and automatically derive complete sets of test cases. Most approaches follow the idea of never-claims and use counterexamples, but some also use witness traces instead of counterexamples; these two ideas are complementary, as a simple negation of the used properties is sufficient to switch from one to the other. This section only considers the basic idea of test case generation, systematic techniques are considered in later sections.

A test purpose describes the desired characteristics of a test case that should be created. For example, it could describe the final state of the test case, or a sequence of states that should be traversed. The test purpose is specified in temporal logic and then converted to a never-claim by negation; this asserts that the test purpose never becomes true. Model checking the never-claim on a model results in a counterexample, if the never-claim becomes false at some point. The counterexample illustrates how the never-claim is violated, and thus shows how the original test purpose is fulfilled. As will be shown in Sections 4 and 5, a popular approach is to automatically create never-claims based on coverage criteria. These never-claims are called *trap properties* [32], and for each item that should be covered one trap property is generated. A test purpose is not necessarily feasible, but fortunately infeasible test purposes are not a problem, because the

never-claim for an infeasible test purpose simply results in no counterexample.

The exact interpretation of counterexamples as test cases depends on the system under test. In many cases, testing with model checkers is applied to reactive systems, which read input values from sensors and set output values accordingly. The model therefore consists of a set of variables representing input, output, and possibly internal variables, as depicted in Figure 1. The system reacts to inputs by setting output values, such that a logical step in a counterexample can be mapped to an execution cycle of the system under test (see Figure 2(b)). Testing with model checkers is not limited to this specific type of automaton, but mapping from counterexamples to test cases will vary for different scenarios; for example when considering flow graphs [33].

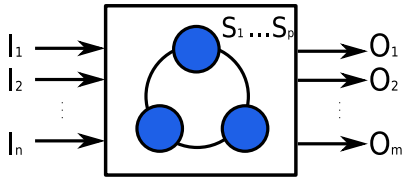
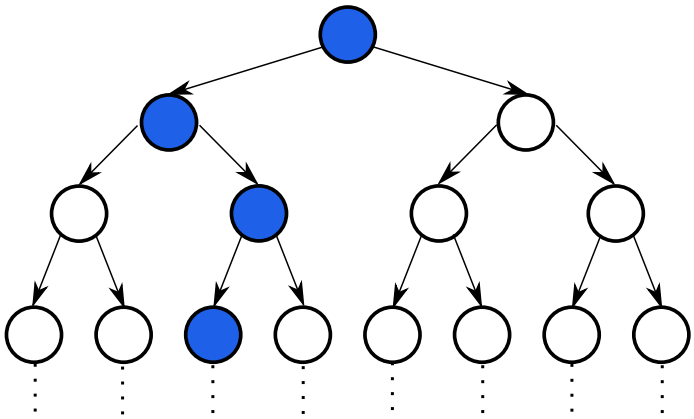
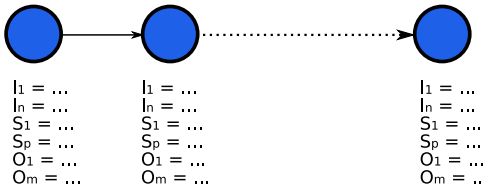


Figure 1: Reactive system model.



(a) A counterexample is a trace in the execution tree.



(b) Counterexample, usable as test case.

Figure 2: Counterexamples are execution paths, where each state assigns values to all variables.

In the reactive system scenario, counterexamples can directly be interpreted as test cases. Because test cases are always finite, it is necessary to distinguish between traces with or without loopback when mapping a trace to a test case.

Definition 8 (Test Case) *A test case $t := \langle s_0, ...s_n \rangle$ related to Kripke structure K is a finite sequence such that $\forall\ 0 \leq i < n : (s_i, s_{i+1}) \in T$ for K.*

The number of transitions a test case consists of is referred to as its *length*. E.g., test case $t := \langle s_0, s_1, ..., s_n \rangle$ has a length of $length(t) = n$. Test cases can easily be created from traces (Definition 3). If a trace does not contain a loopback state, then trace and test case are identical. If the trace does contain a loopback, i.e., it is a lasso-shaped sequence, then the lasso needs to be unfolded. Tan et al. [34] describe truncation strategies to create finite test cases from lasso-shaped sequences. When using a white-box testing technique, the complete internal state is known, and can be tracked in detail during test case execution. Therefore a test case can be terminated whenever the same state has been visited twice at the same position in the loop. When using a black-box testing approach, the loop part of the trace is repeated a finite number of times.

This interpretation of counterexamples as test cases can only be directly applied to deterministic systems. If there is nondeterminism, then a concrete counterexample contains only one possible choice for each nondeterministic choice. Applying such a test case to an implementation that makes a different but valid choice would falsely report a fault. There are considerations of how to extend model checker based testing to nondeterministic systems; see Section 7.7 for more details.

The result of the test case generation is a *test suite* (or *test set*). A test suite TS is a finite set of test cases.

Definition 9 (Test Suite) *A test suite TS is a finite set of n test cases. The* size *of TS is n. The overall* length *of a test suite TS is the sum of the lengths of its test cases t: $length(TS) = \sum_{t \in TS} length(t)$.*

In order to describe how a test case is executed, further definitions are necessary. In general, the test case execution depends on the relation between the model and the system under test (SUT). The most common scenario in the literature is that of reactive systems, which are executed in an infinite loop. This scenario is also assumed here; if the mapping from model to SUT is different, then test case execution has to be adapted to this change. A time step in the model can easily be mapped to an execution cycle in such a reactive system. In each cycle, the SUT receives stimuli from the environment via its *inputs*. Using the inputs, the SUT performs some computations and makes some changes, which can be observed via its *outputs*. A test case is executed via interaction with the system under test (SUT). For this execution, the inputs are provided by the tester. The resulting outputs are observed, and compared to the expected values. This observation leads to a *verdict*, which can be either *fail* or *pass*, expressing that a fault was found or not, respectively.

So far, a counterexample was only described as a sequence of states. According to the definition of a Kripke structure, each state can be mapped to a set of atomic propositions that hold in this state. In order to use counterexamples as test cases, we need to identify inputs and outputs at each state. For this, we use the concept of *modules*, as described by Boroday et al. [35] in the context of testing with model checkers, and originating from module checking theory [36]. In practice, the set of atomic propositions AP of a Kripke structure does not contain deliberate propositions; a system is defined by a set of variables [12, 35, 37] (see Figure 1). The labeling function for a state therefore results in a valuation of all variables in that state. The variables can be partitioned into input, output and internal variables.

Definition 10 (Module) *A module M is a triple $M = (K, I, O)$, where $K = (S, S_0, T, L)$ is a Kripke structure, and $I, O \subseteq AP$ are disjoint sets of input and output variables. The set of hidden variables is defined as $H = AP \backslash (I \cup O)$.*

Intuitively, a module works as follows: In every state s, M reads $L(s) \cap I$, stores internally $L(s) \cap H$, and outputs $L(s) \cap O$. $Inp(s) = L(s) \cap I$ denotes the *input* of the module at state s, and $Out(s) = L(s) \cap O$ denotes the *output* of the module at state s. A trace $t := \langle s_0, s_1, ...s_n \rangle$ of module M can be interpreted such that the output sequence $\langle Out(s_0), Out(s_1), ...Out(s_n) \rangle$ is produced in response to the input sequence $\langle In(s_0), In(s_1), ...In(s_n) \rangle$.

Informally, a test case $t := \langle s_0, s_1, ...s_n \rangle$ is therefore executed on an SUT I by providing values for all input-variables described by $Inp(s_i)$ to I and comparing the output-variables described by $Out(s_i)$ with the values returned by I, for every state s_i, $0 \leq i \leq n$. This actually allows two different interpretations: Synchronous languages [38] such as Lustre [39], Esterel [40], or Signal [41] assume that the implementation responds immediately, which is known as the *synchrony hypothesis*. This hypothesis can be verified by showing that the program execution time is always smaller than the time between two successive external inputs. Under this assumption the expected output is contained in the same state as the input within a test case. Alternatively, the expected output can be assigned to the successor state of the state containing the inputs. There is little difference between these two choices, as long as both model and test case execution framework agree on the interpretation. Formal testing theory assumes the existence of a formal model representing the implementation. The formal model for the implementation does not actually have to exist; this is known as the *test hypothesis* (see e.g., [42]). In our setting, execution of a test case is defined by interpreting the implementation as a module $M_I = (K_I, I, O)$.

Definition 11 (Test Case Execution) *Execution of test case* $t := \langle t_0, ...t_n \rangle$ passes *on implementation* $I = (K_I, I, O)$*, if* I *has a path* $p := \langle s_0, ...s_n, ... \rangle$*, i.e.,* $p \in Paths(I)$*, such that for all states* $t_i : 0 < i \leq n$*:* $s_i = t_i$*. If* I *does not have such a path, the test case* t fails *on* I*.*

As initially proposed by Engels et al. [2] and implemented in several different approaches (see Section 6), an alternative to converting test purposes to never-claims is to introduce deliberate errors in the model. The details of such an approach are discussed in Section 6; but with hindsight of this we refine the definition of a test case. A distinction is made between *positive* and *negative* test cases, depending on whether they contain a deliberate error or not.

What exactly leads to the detection of a fault depends on the type of test case. Positive test cases, suffixed with "$^+$", describe correct (positive) behavior. Negative test cases, marked with the suffix "$^-$", describe faulty (negative) behavior:

Definition 12 (Positive Test Case) *A positive test case* t^+ *detects a fault if its execution* fails*.*

Definition 13 (Negative Test Case) *A negative test case* t^- *detects a fault if its execution* passes*. A negative test case contains a transition* (t_i, t'_j) *which is not defined by the reference model Kripke structure* K*, i.e.,* $(t_i, t'_j) \notin T$*.*

A correct implementation is expected to pass positive test cases, therefore positive test cases are also referred to as *passing* tests. In contrast, negative test cases should not be passed by a correct implementation, hence such test cases are also known as *failing* tests. The majority of available approaches produce positive test cases. Therefore, if a test case is not specially denoted as t^- or t^+ in this paper, it is a positive test case.

3 A SIMPLE DEMONSTRATION MODEL

To illustrate the presented techniques, the following model serves as an example in this paper: The model represents a simplified controller of a car (CC). It has two Boolean inputs that represent the user's decision

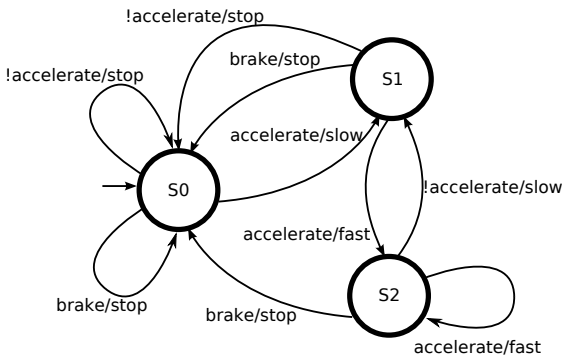


Figure 3: FSM of example model Car Controller (CC).

to accelerate or brake. Upon acceleration, the car starts moving, with either slow or fast velocity. Upon braking the car immediately stops. Figure 3 depicts this model as an FSM, where transitions are labeled with the input and the resulting velocity as output. Figure 4 shows the SMV source code of the model.

```
MODULE main
VAR
  accelerate: boolean;
  brake: boolean;
  velocity: { stop, slow, fast };

ASSIGN
  init(velocity) := stop;
  next(velocity) := case
     accelerate & !brake & velocity = stop : slow;
     accelerate & !brake & velocity = slow : fast;

     !accelerate & !brake & velocity = fast : slow;
     !accelerate & !brake & velocity = slow : stop;

     brake: stop;

     TRUE : velocity;
  esac;
```

Figure 4: CC represented as SMV model.

In addition to the model, several simple requirement properties are expressed in LTL:

1. Whenever the brake is activated, movement has to stop.

$$\square(\texttt{brake} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop}) \tag{13}$$

2. When accelerating and not braking, the velocity has to increase gradually, until it is fast.

$$\square(\neg\texttt{brake} \wedge \texttt{accelerate} \wedge \texttt{velocity} = \texttt{stop} \rightarrow \bigcirc \texttt{velocity} = \texttt{slow}) \quad (14)$$
$$\square(\neg\texttt{brake} \wedge \texttt{accelerate} \wedge \texttt{velocity} = \texttt{slow} \rightarrow \bigcirc \texttt{velocity} = \texttt{fast}) \quad (15)$$

3. When not accelerating and not braking, the velocity has to decrease gradually, until the car stops.

$$\square(\neg\texttt{brake} \wedge \neg\texttt{accelerate} \wedge \texttt{velocity} = \texttt{fast} \rightarrow \bigcirc \texttt{velocity} = \texttt{slow}) \quad (16)$$
$$\square(\neg\texttt{brake} \wedge \neg\texttt{accelerate} \wedge \texttt{velocity} = \texttt{slow} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop}) \quad (17)$$

4 COVERAGE BASED TEST CASE GENERATION

While manual specification of test purposes as proposed by Engels et al. [2] can lead to efficient test cases, it is usually advantageous to systematically create complete test sets according to some test objective. It is difficult to ensure complete coverage of all possible system behaviors with manually specified test purposes.

Coverage criteria are a means to measure how thorough a system is exercised by a given test suite. A coverage criterion is defined on some aspect of a program or specification, for example statements or code branches. Full coverage is achieved, if all items described by the coverage criterion are executed (= *covered*) by at least one test case. Coverage is usually quantified as the percentage of items that are covered.

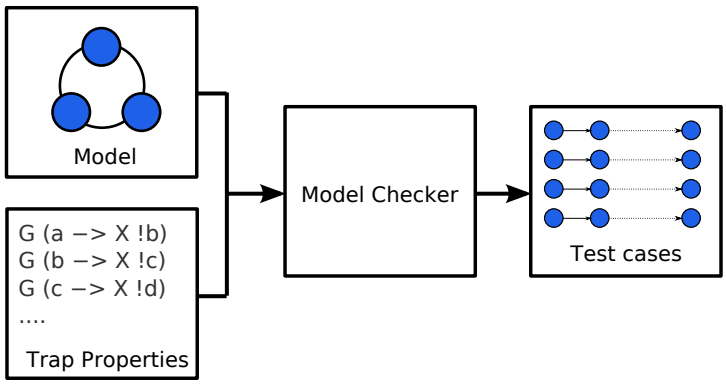


Figure 5: Coverage based test case generation.

Model checkers can be used to automatically derive test suites for maximum coverage of a given criterion. This process is illustrated in Figure 5. For each item that should be covered, a distinct never-claim (*trap property* [32]) is specified. The test suite is created by model checking all trap properties against a given model. Again, infeasible trap properties are detected if the model checker creates no counterexamples.

For example, in order to create a test suite that covers all states of the system variables, a trap property for each possible state a of every variable x is needed, claiming that the value is not taken: $\square\neg(x = a)$. A counterexample to such an example trap property is any trace that contains a state where $x = a$. In the car controller example introduced in Section 3, state coverage with respect to each variable could be achieved

with the following set of trap properties:

$\Box(\texttt{accelerate} \neq 0)$
$\Box(\texttt{accelerate} \neq 1)$
$\Box(\texttt{brake} \neq 0)$
$\Box(\texttt{brake} \neq 1)$
$\Box(\texttt{velocity} \neq \texttt{stop})$
$\Box(\texttt{velocity} \neq \texttt{slow})$
$\Box(\texttt{velocity} \neq \texttt{fast})$

While trap properties for state coverage are simple safety properties, trap properties can be deliberate temporal logic properties for which counterexamples exist; for example, they can be defined over transitions or sequences of transitions.

4.1 Coverage of SCR Specifications

The concept of trap properties was initially proposed by Gargantini and Heitmeyer [32] with regard to SCR (Software Cost Reduction method [43]) specifications. An SCR model is defined as a quadruple (S, S_0, E^m, T), where S is the set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of input events, and T is the transform describing the allowed state transitions. T is described with tables for events (predicates defined on a pair of system states implying that the value of at least one state variable has changed) and conditions (predicates defined on a system state) with regard to all variables controlled by the considered system.

SCR specifications consist of different types of tables. A condition table defines a variable as a function of a mode and a condition, and an event table defines a variable as a function of a mode and an event. Table 1 shows a mode table for the CC example model.

Table 1: SCR Mode Transition Table for `velocity` in CC example.

Current Mode	Event	New Mode
`stop`	$\texttt{accelerate} = 1\ AND\ \texttt{brake} = 0$	`slow`
`slow`	$\texttt{accelerate} = 1\ AND\ \texttt{brake} = 0$	`fast`
`fast`	$\texttt{accelerate} = 0\ AND\ \texttt{brake} = 0$	`slow`
`slow`	$\texttt{accelerate} = 0\ AND\ \texttt{brake} = 0$	`stop`
`fast`	`brake`	`stop`

The operational SCR specification is automatically converted to an SMV or SPIN model. SCR requirement properties can then be used as never-claims, as in [2]. The novel idea presented in [32] is to automatically create trap properties from the SCR tables. Each table is converted to an if-else construct for the model checker SPIN, or a case-statement for SMV. For each variable, a designated variable is added, which indicates which branch of the if-else/case construct is currently active. For each possible value of this special variable, a trap property is formulated claiming that this value is never taken. For example, the variable $\texttt{Case}Var$ represents the chosen case for variable Var. Resulting trap properties in LTL would be,

e.g., $\Box\neg(\texttt{Case}Var = 1)$, $\Box\neg(\texttt{Case}Var = 2)$, etc. The trap properties automatically result in a test suite for branch coverage of the SCR model. Considering Table 1, the resulting model will contain five cases. Consequently, there will be six trap properties, because the "no-change" case has to be considered as well. The trap properties are:

$$\{\Box\neg(\texttt{Case\_velocity} = i) \mid 0 \leq i \leq 6\}$$

4.2 Coverage of Transition Systems

Heimdahl et al. [44] proposed a framework for specification based test case generation, focusing on structural coverage criteria, independently of any specific formalism. This framework is instantiated in [37, 45, 46], where a general transition system definition is given. Other formalisms, such as SCR or RSML^{-e} [47], can be interpreted as such transition systems, and mapped to model checking specifications, e.g., SMV. The language RSML^{-e} is based on the Statecharts like language Requirements State Machine Language (RSML) [48], and adds support for interfaces between the environment and the control software.

In this framework, the system state is uniquely determined by the values of n variables $\{x_1, x_2, ..., x_n\}$. Each variable x_i has a domain D_i, and consequently the reachable state space of a system is a subset of $D = D_1 \times D_2 \times ... \times D_n$. 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 $\beta_{i,j}$ is the j-th post-state condition. A simple transition for a variable x_i is a conjunction of $\alpha_{i,j}$, $\beta_{i,j}$ and a 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 $\{x_1, ..., x_n\}$. Consequently, a basic transition system is defined as follows:

Definition 14 (Basic Transition System) *A transition system M over variables $\{x_1 ... x_n\}$ is a tuple $M = (D, \Delta, \rho)$, with $D = D_1 \times D_2 \times ... \times D_n$, $\Delta = \bigwedge_i \delta_i$, and the initial state expression ρ. For each variable x_i there is a transition relation δ_i, that is the disjunction of several simple transitions $\delta_{i,j} = \alpha_{i,j} \wedge \beta_{i,j} \wedge \gamma_{i,j}$, where $\alpha_{i,j}$, $\beta_{i,j}$, and $\gamma_{i,j}$ are pre-state, post-state, and guard conditions of the j-th simple transition of variable x_i.*

Figure 6 illustrates the RSML^{-e} specification for our example CC model. The corresponding simple transitions for the CC model are listed in Table 2.

Trap properties can be derived from the basic transition system model. Structural coverage criteria are defined with regard to the transition relation Δ. These coverage criteria are similar to common code based coverage criteria such as decision or condition coverage, but refer to transitions. In general, the conjunction of pre-state condition α and guard condition γ can be interpreted as a logical predicate, which allows the use of common logic based coverage criteria [49]. The post-state condition β is used as part of the trap properties to force the creation of relevant counterexamples.

Simple Transition Coverage requires that each simple transition of every variable is executed. A simple transition consists of pre-state, post-state, and guard conditions. Consequently, a trap property to create an according test case simply has to require that always when the pre-state condition and guard are true, the post-state condition may *not* be satisfied in the next state:

$$\Box\,\alpha \wedge \gamma \rightarrow \bigcirc\neg\beta$$

```
STATE_VARIABLE velocity:
  VALUES: { stop, slow, fast }
  INITIAL_VALUE: stop
  CLASSIFICATION: State

  EQUALS stop IF
    TABLE
      accelerate                 : * F F;
      brake                      : T F F;
      PREV_STEP(velocity) = stop : * * T;
      PREV_STEP(velocity) = slow : * T *;
    END TABLE

  EQUALS slow IF
    TABLE
      accelerate                 : T F;
      brake                      : F F;
      PREV_STEP(velocity) = stop : T *;
      PREV_STEP(velocity) = fast : * T;
    END TABLE

  EQUALS fast IF
    TABLE
      accelerate                 : T T;
      brake                      : F F;
      PREV_STEP(velocity) = slow : T *;
      PREV_STEP(velocity) = fast : * T;
    END TABLE

END STATE_VARIABLE
```

Figure 6: Simple car controller as RSML^{-e} specification.

In the CC example (Figure 4), simple transition coverage is achieved with the following trap properties:

$$\Box(\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} \wedge \neg\texttt{brake} \rightarrow \bigcirc\neg(\texttt{velocity} = \texttt{slow}))$$
$$\Box(\texttt{velocity} = \texttt{slow} \wedge \texttt{accelerate} \wedge \neg\texttt{brake} \rightarrow \bigcirc\neg(\texttt{velocity} = \texttt{fast}))$$
$$\Box(\texttt{velocity} = \texttt{fast} \wedge \neg\texttt{accelerate} \wedge \neg\texttt{brake} \rightarrow \bigcirc\neg(\texttt{velocity} = \texttt{slow}))$$
$$\Box(\texttt{velocity} = \texttt{slow} \wedge \neg\texttt{accelerate} \wedge \neg\texttt{brake} \rightarrow \bigcirc\neg(\texttt{velocity} = \texttt{stop}))$$
$$\Box(\texttt{brake} \rightarrow \bigcirc\neg(\texttt{velocity} = \texttt{stop}))$$
$$\Box(\texttt{velocity} = \texttt{fast} \wedge \texttt{accelerate} \wedge \neg\texttt{brake} \rightarrow \bigcirc\neg(\texttt{velocity} = \texttt{fast}))$$
$$\Box(\texttt{velocity} = \texttt{stop} \wedge \neg\texttt{accelerate} \wedge \neg\texttt{brake} \rightarrow \bigcirc\neg(\texttt{velocity} = \texttt{stop}))$$

Note that counterexamples to such trap properties end with the simple transition from α to β. If this is

Table 2: Simple transitions for `velocity`.

Transition	α	β	γ
1	$\texttt{velocity} = \texttt{stop}$	$\texttt{velocity} = \texttt{slow}$	$\texttt{accelerate} \wedge \neg\texttt{brake}$
2	$\texttt{velocity} = \texttt{slow}$	$\texttt{velocity} = \texttt{fast}$	$\texttt{accelerate} \wedge \neg\texttt{brake}$
3	$\texttt{velocity} = \texttt{fast}$	$\texttt{velocity} = \texttt{slow}$	$\neg\texttt{accelerate} \wedge \neg\texttt{brake}$
4	$\texttt{velocity} = \texttt{slow}$	$\texttt{velocity} = \texttt{stop}$	$\neg\texttt{accelerate} \wedge \neg\texttt{brake}$
5	$True$	$\texttt{velocity} = \texttt{stop}$	$\texttt{brake}$
6	$\texttt{velocity} = \texttt{fast}$	$\texttt{velocity} = \texttt{fast}$	$\texttt{accelerate} \wedge \neg\texttt{brake}$
7	$\texttt{velocity} = \texttt{stop}$	$\texttt{velocity} = \texttt{stop}$	$\neg\texttt{accelerate} \wedge \neg\texttt{brake}$

not an observable transition, then an additional postamble sequence is necessary. The *transition coverage* criterion described by Offutt et al. [50] is basically identical to the simple transition coverage criterion.

Simple Guard Coverage is similar to decision coverage in code based testing. Simple guard coverage requires that for each simple transition there exists a test case s where the guard evaluates to true in a state where the pre-state condition is true, and a test case t where the guard evaluates to false in a state where the pre-state condition is true. This criterion corresponds to the *predicate coverage* criterion [49] for logical expressions. For example, this could be expressed as a pair of trap properties:

$$\Box\, \alpha \wedge \gamma \rightarrow \bigcirc \neg\beta$$
$$\Box\, \alpha \wedge \neg\gamma \rightarrow \bigcirc \beta$$

Again, the post-state expression β is negated in order to force creation of suitable counterexamples for the case when the guard evaluates to true. When the guard evaluates to false, creation of a counterexample is forced by claiming β will be true in the next state. The CC example (Figure 4) requires fourteen trap properties to achieve simple guard coverage. To save space we only consider the first simple transition:

$$\Box\, (\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} \wedge \neg\texttt{brake} \rightarrow \bigcirc \neg(\texttt{velocity} = \texttt{slow}))$$
$$\Box\, (\texttt{velocity} = \texttt{stop} \wedge \neg(\texttt{accelerate} \wedge \neg\texttt{brake}) \rightarrow \bigcirc (\texttt{velocity} = \texttt{slow}))$$

Condition Coverage [50] requires that for each condition (clause) in a predicate there is a test case where the condition evaluates to true, and a test case where the condition evaluates to false. This criterion corresponds to the *clause coverage* criterion [49] for logical expressions, and can also be applied to guard conditions. It can be expressed as a pair of trap properties for clause c:

$$\Box \neg(\alpha \wedge c = true)$$
$$\Box \neg(\alpha \wedge c = false)$$

The trap property claims that there is no state where the pre-state condition α is true and the clause c takes evaluates to true or false. Resulting counterexamples do not necessarily execute the transition.

Again we consider the first simple transition of the example model:

$$\Box\,(\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} = true)$$
$$\Box\,(\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} = false)$$
$$\Box\,(\texttt{velocity} = \texttt{stop} \wedge (\neg\texttt{brake}) = true)$$
$$\Box\,(\texttt{velocity} = \texttt{stop} \wedge (\neg\texttt{brake}) = false)$$

Complete Guard Coverage is similar to the *multiple condition* criterion in code based coverage analysis, also known as *combinatorial coverage* [49]. A guard condition consists of several clauses (usually called conditions in code based coverage). Complete guard coverage requires that all possible combinations of truth values for the clauses of a guard are covered.

Let the clauses in a guard condition γ be $\{c_1, \ldots, c_l\}$, then complete guard coverage of γ requires a test case s for any given Boolean vector u of length l, such that for some i: $\bigwedge_{k=1}^{l}(c_k(s_i, s_{i+1}) = u_k)$. This means that for every u there has to be a trap property of the type:

$$\Box\,\neg(\alpha \wedge \bigwedge_{k=1}^{l}(c_k(s_i, s_{i+1}) = u_k))$$

The trap property claims that there is no state where the pre-state condition α is true and the clauses take on the values described by u_k; this results in a trace that leads to the chosen valuation for the guard condition. Note that this trace does not necessarily execute the transition.

Again we consider the first simple transition of the example model:

$$\Box\,(\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} = false \wedge (\neg\texttt{brake}) = false)$$
$$\Box\,(\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} = false \wedge (\neg\texttt{brake}) = true)$$
$$\Box\,(\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} = true \wedge (\neg\texttt{brake}) = false)$$
$$\Box\,(\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} = true \wedge (\neg\texttt{brake}) = true)$$

The number of trap properties quickly increases with the number of clauses. Consequently, if the number of clauses is too big, then complete guard coverage can result in too many test cases to be useful. As a more practical solution, the modified decision/condition coverage (MC/DC) criterion [51] has been proposed in the context of code coverage. MC/DC requires that each condition (clause) is shown to independently affect the value of the decision (predicate) it is part of. This informal definition is ambiguous, and allows three different interpretations. There are many variations of MC/DC in the literature; Ammann and Offutt [52] articulate the differences and provide a uniform framework of definitions. Following the nomenclature used in [52], the considered clause in a predicate will be called the *major* clause, and the remaining clauses *minor* clauses. In all interpretations of MC/DC, it takes a pair of test cases to cover a clause. As test case pairs for different clauses need not be disjoint, the size of an MC/DC test suite can be as small as $l + 1$ test cases for a predicate with l clauses. In the strictest variant (1), the values of all minor clauses are fixed while the value of the major clause is changed, which also has to result in a change of the value of the predicate. A slightly relaxed variant (2) still requires that the predicate takes on both values, but the values of the minor clauses do not need to be fixed. Finally, the decision coverage can also be relaxed, resulting in variant (3), which does not require the predicate to take on both values.

Clause-wise Guard Coverage is an adaptation of the strictest interpretation (1) of MC/DC to basic transition systems, introduced by Rayadurgam and Heimdahl [37]. The authors assume some mechanism that

calculates a pair of Boolean vectors u and v of equal length l for l clauses in guard γ, where only the m-th value differs. When the clauses c_i are assigned the values in u, then γ evaluates to true, and when assigned the values in v, then γ evaluates to false. The vectors u and v could, for example, be derived using constraint satisfaction techniques. These vectors can be used to formulate the following trap properties for the m-th clause of guard γ:

$$\Box\,\alpha \wedge \bigwedge_{k=1}^{l} (c_k = u_k) \rightarrow \bigcirc \neg\beta$$

$$\Box\,\alpha \wedge \bigwedge_{k=1}^{l} (c_k = v_k) \rightarrow \bigcirc \beta$$

The first trap property results in a test case where the guard evaluates to true, and the second trap property results in a test case where the guard evaluates to false as a consequence of a different value for c_m.

Considering the first simple transition of our example again, clause-wise guard coverage is achieved with the following set of trap properties (slightly rewritten to fit into the page width):

$$\Box\,((\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} \wedge (\neg\texttt{brake}) = true) \rightarrow \bigcirc\,(\neg\texttt{velocity} = \texttt{slow}))$$

$$\Box\,((\texttt{velocity} = \texttt{stop} \wedge \neg\texttt{accelerate} \wedge (\neg\texttt{brake}) = true) \rightarrow \bigcirc\,(\texttt{velocity} = \texttt{slow}))$$

$$\Box\,((\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} \wedge (\neg\texttt{brake}) = false) \rightarrow \bigcirc\,(\texttt{velocity} = \texttt{slow}))$$

Heimdahl et al. [53] also define *Clause-wise Transition Coverage*, which is identical to clause-wise guard coverage but defined in the context of the specification language RSML^{-e}. In a case study [53], a flight guidance system specified in RSML^{-e} at varying levels of abstraction is analyzed with regard to test case generation. The case study shows that the performance of the model checker is a critical factor, but if the model size is within bounds, then coverage based test case generation is feasible. Complex criteria such as clause-wise transition coverage result in better test cases than very simple criteria like state coverage or simple transition coverage. A high complexity of trap properties has a negative effect on the performance.

The drawback of the solutions described in [37] and [53] is that a mechanism that calculates appropriate valuations of all minor clauses is required. Furthermore, all clauses have to be independent, otherwise there might not exist a test case for every chosen valuation. As a solution, Rayadurgam and Heimdahl [54] describe a method to create pairs of test cases for MC/DC. The solution consists of altering the model such that there are auxiliary Boolean variables that store the values of clauses. The initial values of these auxiliary variables are nondeterministically assigned by the model checker to true or false. After that, these values are not changed anymore. This results in a vector u of l value assignments to the l clauses of a guard condition. Note that the model checker selected suitable valuations automatically, and no additional mechanism to calculate vectors is necessary. The vector chosen by the model checker is used to create a counterexample for clause c_m that represents two concatenated test cases: A test case where γ evaluates to true, and a test case where γ evaluates to false, where the values of all clauses except the major clause c_m are defined by u. The first case is covered by any counterexample to the following:

$$\Box\,(\neg(\alpha \wedge \gamma \wedge c_m \neq u_m \wedge \bigwedge_{k \neq m} (c_k = u_k)))$$

The second case is covered by any counterexample to the following:

$$\Box\,(\neg(\alpha \wedge \neg\gamma \wedge c_m = u_m \wedge \bigwedge_{k \neq m} (c_k = u_k)))$$

Combining these two trap properties to one trap property achieves that the same valuations for all clauses except the major clause are used. To reach the same decision point twice in a single run of a reactive system, a dedicated hard reset transition might be necessary. Resulting counterexamples can then be split at the hard reset transition into two test cases. Consequently, a trap property for a pair of MC/DC test cases for major clause c_m results in the following:

$$\square\,(\neg(\alpha \wedge \gamma \wedge c_m \neq u_m \wedge \bigwedge_{k \neq m} (c_k = u_k))) \vee \square\,(\neg(\alpha \wedge \neg\gamma \wedge c_m = u_m \wedge \bigwedge_{k \neq m} (c_k = u_k)))$$

As an example, consider the first transition of the CC model, and let `accelerate` be the major clause. This results in the following trap property, where the model checker performs the task of choosing suitable values for u_a and u_b:

$$\square\,(\neg(\texttt{velocity} = \texttt{stop} \wedge (\texttt{accelerate} \wedge \neg\texttt{brake}) \wedge \texttt{accelerate} \neq u_a \wedge (\neg\texttt{brake}) = u_b))) \vee$$
$$\square\,(\neg(\texttt{velocity} = \texttt{stop} \wedge \neg(\texttt{accelerate} \wedge \neg\texttt{brake}) \wedge \texttt{accelerate} = u_a \wedge (\neg\texttt{brake}) = u_b))$$

It is conceivable to modify this kind of trap property such that the considered transitions are actually executed. For example, the following trap property achieves that a counterexample will first take the considered transition (to make left part of implication false), and then reach a point where the guard evaluates to false (here, execution of a transition is forced by claiming $\bigcirc\beta$, which is false as γ is false):

$$\square\,((\alpha \wedge \gamma \wedge c_m \neq u_m \wedge \bigwedge_{k \neq m} (c_k = u_k)) \rightarrow \square\,(\alpha \wedge \neg\gamma \wedge \wedge c_m \neq u_m \wedge \bigwedge_{k \neq m} (c_k = u_k)) \rightarrow \bigcirc\beta)$$

Full Predicate Coverage [50] requires that each clause in each predicate is tested independently. In contrast to the previously discussed clause wise coverage criteria, the values of minor clauses may change as long as the value of the predicate is still determined by the considered clause. Consequently, full predicate coverage corresponds to interpretation (2) of MC/DC as described above. In [55] this criterion is further relaxed to *Uncorrelated Full Predicate Coverage*. While this still requires that for a clause it is shown for both possible truth values that it influences the predicate, it is not required that the actual value of the predicate differs; that is, it drops the requirement for decision coverage. This corresponds to interpretation (3) of MC/DC as described above. In [55], the Boolean derivative [56] is used to create two trap properties for each clause. The Boolean derivative dP/dc of predicate P for condition c is a predicate on the remaining conditions that is true if the value of c determines the value of P. Trap properties for condition a in a predicate P can be formulated by claiming that the derivative always implies c is false and in a second property that c is true. We apply the derivative to the guard condition γ as follows:

$$\square\,((\alpha \wedge d(\gamma)/dc) \rightarrow c)$$
$$\square\,((\alpha \wedge d(\gamma)/dc) \rightarrow \neg c)$$

Again we consider the first simple transition of the example model:

$$\square\,((\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate}) \rightarrow (\neg\texttt{brake}))$$
$$\square\,((\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate}) \rightarrow \neg(\neg\texttt{brake}))$$
$$\square\,((\texttt{velocity} = \texttt{stop} \wedge \neg\texttt{brake}) \rightarrow (\texttt{accelerate}))$$
$$\square\,((\texttt{velocity} = \texttt{stop} \wedge \neg\texttt{brake}) \rightarrow \neg(\texttt{accelerate}))$$

A natural extension of MC/DC is the *Reinforced Condition/Decision Coverage* (RC/DC) [57] criterion. The idea of RC/DC is that it is not sufficient to show for each clause that it independently affects the predicate's outcome; it is also necessary to show that each clause independently *keeps* the outcome. This means that the values of the minor clauses are fixed while the value of the major clause is altered, and the value of the decision does not change because of this. As it might not be possible to keep the values of all minor clauses fixed, this requirement can be relaxed.

Following the MC/DC definition given in [37], RC/DC needs the following additional trap properties, where u', $'v$ is a pair of Boolean vectors of equal length l for l clauses in guard γ, where only the m-th value differs. When the clauses take on the values described in these vectors, the guard condition γ shall evaluate to the same value in both cases. If γ evaluates to true in both cases, then the transition is taken and β evaluates to true in the next state. Therefore, the trap properties contain the negation of β:

$$\square\, \alpha \wedge \bigwedge_{k=1}^{l} (c_k = u_k) \rightarrow \bigcirc \neg\beta$$
$$\square\, \alpha \wedge \bigwedge_{k=1}^{l} (c_k = v_k) \rightarrow \bigcirc \neg\beta$$

If γ evaluates to false, then the negation of β has to be removed.

Considering the first simple transition of our example again, RC/DC is achieved with the following set of trap properties (slightly rewritten to fit into the page width):

$$\square\, ((\texttt{velocity} = \texttt{stop} \wedge \neg\texttt{accelerate} \wedge (\neg\texttt{brake}) = \mathit{false}) \rightarrow \bigcirc (\texttt{velocity} = \texttt{slow}))$$
$$\square\, ((\texttt{velocity} = \texttt{stop} \wedge \neg\texttt{accelerate} \wedge (\neg\texttt{brake}) = \mathit{true}) \rightarrow \bigcirc (\texttt{velocity} = \texttt{slow}))$$
$$\square\, ((\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} \wedge (\neg\texttt{brake}) = \mathit{false}) \rightarrow \bigcirc (\texttt{velocity} = \texttt{slow}))$$

Finally, *Transition Pair Coverage* is another related coverage criterion given in [50] that is also useful in the transition system context. In contrast to (simple) transition it requires that all feasible *pairs* of transitions are covered. As shown in [55], this results in trap properties with two levels of next statements. The following trap property covers the transitions $(\alpha_1, \beta_1, \gamma_1)$ and $(\alpha_2, \beta_2, \gamma_2)$:

$$\square\, (\alpha_1 \wedge \gamma_1 \rightarrow \bigcirc (\alpha_2 \wedge \gamma_2 \rightarrow \bigcirc \neg\beta_2))$$

As an example, a test case covering the pair of the first two transitions of our example is achieved with the following trap property (slightly rewritten to save space):

$$\square\, ((\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} \wedge \neg\texttt{brake}) \rightarrow$$
$$\bigcirc ((\texttt{velocity} = \texttt{slow} \wedge \texttt{accelerate} \wedge (\neg\texttt{brake})) \rightarrow \bigcirc \neg(\texttt{velocity} = \texttt{fast})$$

4.3 Control and Data Flow Coverage Criteria

The previous section presented coverage criteria in the context of basic transition systems. Such coverage criteria, however, are not limited to this specific kind of model, but can be applied to any system or specification that uses Boolean predicates. For example, Hong and Lee [33] use similar criteria (state coverage, transition coverage) to create test cases for the control flow of a program or EFSM model, and define trap properties to generate test cases for data flow coverage criteria. So far, the discussed coverage criteria only

considered the control flow of a model. Control flow criteria are based on logical expressions in the specification, which determine the branching during the execution. In contrast, data flow oriented coverage criteria consider how variables are defined and used during execution.

Test case generation with regard to coverage of data flow graphs is considered in [33, 58]. A flow graph G is defined as a tuple $G = (V, v_s, v_f, A)$, where V is a finite set of vertices, $v_s \in V$ is the start vertex, $v_f \in V$ is the final vertex, and A is a finite set of arcs. A vertex represents a statement and an arc represents possible flow of control between statements. The set of variables that is defined at a vertex v is denoted with $DEF(v)$, and the set of variables that is used at a vertex v is denotes with $USE(v)$. A flow graph can be interpreted as a Kripke structure $K(G) = (V, v_s, L, A \cup \{(v_f, v_f)\})$, where $L(v_s) = \{start\}$, $L(v_f) = \{final\}$, and $L(v) = DEF(v) \cup USE(v)$ for every $v \in V - \{v_s, v_f\}$.

```
v1:   input ( accelerate , brake , previous_velocity );
v2:    velocity = previous_velocity ;
v3:   if ( brake ) {
v4:      velocity = stop ;
      } else {
v5:      if ( accelerate ) {
v6:         if ( velocity == stop )
v7:            velocity = slow ;
            else
v8:            velocity = fast ;
         } else {
v9:         if ( velocity == fast )
v10:           velocity = slow ;
            else
v11:           velocity = stop ;
      }
v12:  output ( velocity );
```

Figure 7: Example implementation of CC example.

As an example, Figure 7 shows an implementation of our car controller example. The corresponding data flow graph is depicted in Figure 8, where vertexes are annotated with their DEF and USE sets. Hong et al. [59] show how model checkers can be used to derive test cases for different data flow coverage criteria using witness formulas. In this survey, we use the criteria by Rapps and Weyuker [60] to illustrate this approach; trap properties for further criteria are given in Hong et al. [59]. The basic idea of data flow criteria is to find *definition-use pairs* (du-pairs). A pair $(d(x, v), u(x, v'))$ is a du-pair, if there exists a path $\langle v, v_1, \ldots, v_n, v' \rangle$ from vertex v to v', such that x is not defined in any v_i for $1 \leq i \leq n$, or if $n = 0$ (this is called a definition-clear path).

Hong et al. [59] express du-pairs as WCTL-formulas. WCTL formulas are CTL formulas where only the temporal operators **EF** , **EX** , and **EU** occur, and for any sub-formula of the form $f_1 \wedge \ldots \wedge f_n$ all f_i except one at the most are atomic propositions. A WCTL formula for a du-pair ($d(x, v)$, $u(x, v)$) is expressed as follows:

$$wctl(d(x, v), u(x, v')) := \mathbf{EF}\ (d(x, v) \wedge \mathbf{EX}\ \ \mathbf{E}[\neg def(x)\mathbf{U}\ (u(x, v') \wedge \mathbf{EF}\ final)])$$

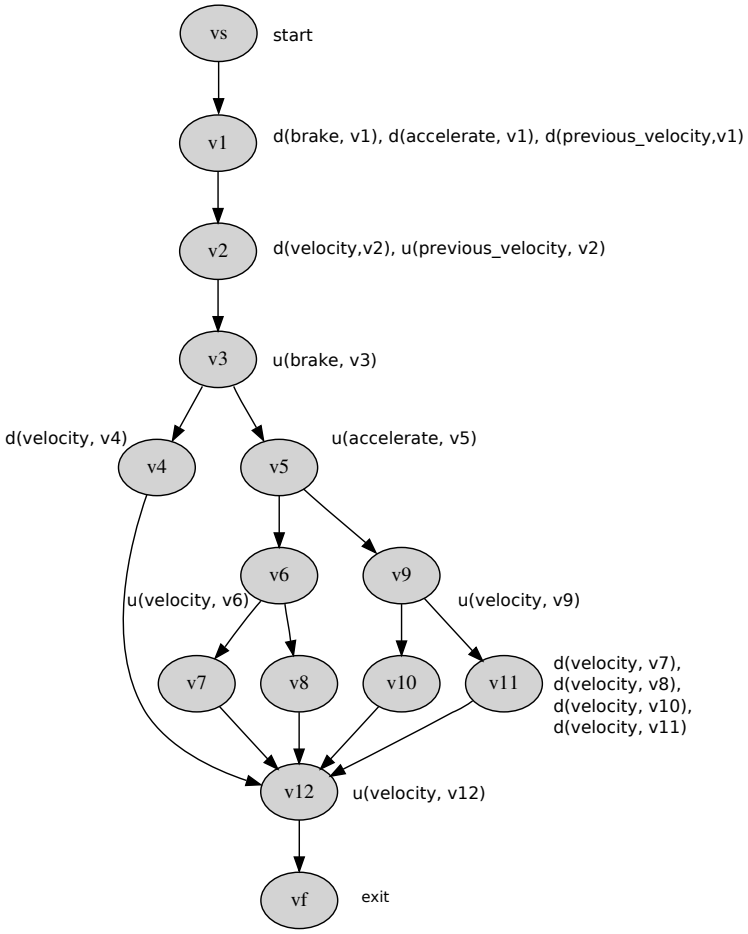


Figure 8: Data flow graph for example implementation.

In this formula, $def(x)$ is the disjunction of all definitions of x. This formula expresses that there exists a path from the initial state to $d(x, v)$, such that there exists a definition-clear path to $u(x, v')$. In addition, **EF** $final$ requires that the path continues to the final vertex, such that the path is a *complete* path. For example, the following formula results for the du-pair $(d(brake, v1), u(brake, v3))$:

$$\begin{aligned} wctl(d(brake, v1), u(brake, v3)) \quad := \quad & \mathbf{EF}\ (d(brake, v1) \wedge \mathbf{EX} \\ & \mathbf{E}[\neg d(brake, v1)\mathbf{U}\ (u(brake, v3) \wedge \mathbf{EF}\ final)]) \end{aligned}$$

An example witness to this formula is $\langle vs, v1, v2, v3, v4, v12, vf \rangle$. Any given $(d(x, v), u(x, v'))$ is a du-pair iff the Kripke structure representing the data flow graph satisfies $wctl(d(x, v), u(x, v'))$.

The *all-defs* coverage criterion requires that for every definition $d(x, v)$ there is a test case that uses a definition-clear path to some $u(x, v')$. Let $DEF(G)$ denote the set of definitions in the data flow graph G, and $USE(G)$ the set of uses in G. Then, a test suite T satisfies the all-defs coverage criterion iff it is a witness set for:

$$\{ \bigvee_{u(x,v) \in USE(G)} wctl(d(x, v), u(x, v')) \mid d(x, v) \in DEF(G) \}$$

Once more, a set of test cases can be created by using the model checker to derive witness sequences for this set of formulas, or equivalently, by calculating counterexamples to the negations of these formulas.

In our example data flow graph in Figure 8, we can identify the example set of du-pairs given below (note that other sets of du-pairs are also possible). A test suite satisfying all-defs can be created by calculating a witness for each $wctl(d(x,v),u(x,v'))$ or counterexample to $\neg(wctl(d(x,v),u(x,v')))$ for each $(d(x,v),u(x,v'))$ in this set:

$\{(d(brake,v1),u(brake,v3)),$
$(d(accelerate,v1),u(accelerate,v5)),$
$(d(previous\_velocity,v1),u(previous\_velocity,v2)),$
$(d(velocity,v2),u(velocity,v6)),$
$(d(velocity,v4),u(velocity,v12)),$
$(d(velocity,v7),u(velocity,v12)),$
$(d(velocity,v8),u(velocity,v12)),$
$(d(velocity,v10),u(velocity,v12)),$
$(d(velocity,v11),u(velocity,v12))\}$

The *all-uses* coverage criterion can be defined in a similar manner. A test suite satisfies the all-uses coverage criterion, if for every definition $d(x,v)$ and every use $u(x,v')$ there exists some definition-clear path with respect to x as part of a test case. Consequently, a test suite satisfies the all-uses coverage criterion if it is a witness set for the following set of formulas:

$$\{wctl(d(x,v),u(x,v')) \mid d(x,v) \in DEF(G), u(x,v') \in USE(G)\}$$

Obviously, the number of du-pairs is larger for the all-uses criterion than for the all-defs criterion. In our example graph, we get the following set of du-pairs:

$\{(d(brake,v1),u(brake,v3)),$
$(d(accelerate,v1),u(accelerate,v5)),$
$(d(previous\_velocity,v1),u(previous\_velocity,v2)),$
$(d(velocity,v2),u(velocity,v6)),$
$(d(velocity,v2),u(velocity,v9)),$
$(d(velocity,v4),u(velocity,v12)),$
$(d(velocity,v7),u(velocity,v12)),$
$(d(velocity,v8),u(velocity,v12)),$
$(d(velocity,v10),u(velocity,v12)),$
$(d(velocity,v11),u(velocity,v12))\}$

In this example, only the du-pair $(d(velocity,v2),u(velocity,v9))$ is added in comparison to the all-defs criterion. Theoretically, the worst case number of du-pairs can be $O(n^2)$ for a flow graph of size n.

Further data flow criteria are considered in [59]. Data flow coverage criteria are extended with control dependence information in [61]. In [62], data and control flow criteria are applied to Statecharts specifications. In [33], control and data flow coverage criteria are defined for extended finite state machines (EFSMs).

4.4 Coverage of Abstract State Machines

Countless specification formalisms have been defined in the past. In general, coverage criteria can be defined and used for testing for any specification language that is susceptible to model checking. For example, Abstract State Machines [63] (ASMs) are yet another formalism that has been considered in the context of coverage oriented test case generation [64, 65]. ASMs are semantically well defined pseudo-code over abstract structures. An ASM consists of states and a finite set of rules for guarded function updates. Rules are of the type *if* $condition$ *then* $updates$, where $condition$ is an arbitrary Boolean expressions, and $updates$ is a finite set of function updates that are executed simultaneously. There are many different types of functions, and basically a nullary function can be interpreted as a variable.

```
data Velocity = stop | slow | fast
instance AsmTerm Velocity

brake :: Dynamic Bool
brake = initVal "brake" False

accelerate :: Dynamic Bool
accelerate = initVal "accelerate" False

velocity :: Dynamic Velocity
velocity = initVal "velocity" stop

r1 :: Rule()
r1 = if (velocity == stop) && (accelerate == True) && (brake == False)
  then velocity := slow

r2 :: Rule()
r2 = if (velocity == slow) && (accelerate == True) && (brake == False)
  then velocity := fast

r3 :: Rule()
r3 = if (velocity == fast) && (accelerate == False) && (brake == False)
  then velocity := slow

r4 :: Rule()
r4 = if (velocity == slow) && (accelerate == False) && (brake == False)
  then velocity := stop

r5 :: Rule()
r5 = if (brake == True)
  then velocity := stop
```

Figure 9: ASM specification for CC example, each transition represented as a distinct rule..

Figure 9 shows the car controller example as an ASM specification, where each transition is represented as a distinct rule, following the style used in the tool ATGT (see Section 10), which automatically creates test cases from ASM specifications with a model checker.

Similarly to the previously described approaches, rules are suitable for trap property generation. *Rule coverage* is similar to simple transition coverage, and requires a test case where the guard condition evaluates to true, and one where the guard condition evaluates to false: **AG** $(\neg condition)$. For example, rule 1 in Figure 9 results in two trap properties, the first one lets the guard evaluate to true, the second one to false:

$$\Box\,(\texttt{velocity} = \texttt{stop} \wedge \neg\texttt{accelerate} \wedge \neg\texttt{brake})$$
$$\Box\,\neg(\texttt{velocity} = \texttt{stop} \wedge \neg\texttt{accelerate} \wedge \neg\texttt{brake})$$

In a similar style, other coverage criteria based on logical predicates can be applied to rule guards. For example, MC/DC is used by [64, 65].

In contrast to these control oriented coverage criteria, the *rule update coverage* requires each update function to be nontrivially executed at least once. This is a data flow coverage criterion, as it considers the value of a variable prior to a new assignment (i.e., definition). Rule update coverage for all five rules of our example specification results in the following trap properties, generated by ATGT:

$$\Box\,(\texttt{velocity} \neq \texttt{slow} \wedge (\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} \wedge \neg\texttt{brake}))$$
$$\Box\,(\texttt{velocity} \neq \texttt{fast} \wedge (\texttt{velocity} = \texttt{slow} \wedge \texttt{accelerate} \wedge \neg\texttt{brake}))$$
$$\Box\,(\texttt{velocity} \neq \texttt{slow} \wedge (\texttt{velocity} = \texttt{fast} \wedge \neg\texttt{accelerate} \wedge \neg\texttt{brake}))$$
$$\Box\,(\texttt{velocity} \neq \texttt{stop} \wedge (\texttt{velocity} = \texttt{slow} \wedge \neg\texttt{accelerate} \wedge \neg\texttt{brake}))$$
$$\Box\,(\texttt{brake} \wedge \texttt{velocity} \neq \texttt{stop})$$

Further coverage criteria are defined by Gargantini and Riccobene [64]; *Parallel rule coverage* requires combinations of updates to be executed in parallel, and *strong parallel rule coverage* requires all possible combinations of parallel update functions to be covered. This approach has been evaluated with the model checker SMV in [64], and with the model checker SPIN in [65].

5 REQUIREMENTS BASED TESTING

The majority of coverage based approaches use some structural coverage criterion based on a behavioral model of the SUT. Sometimes it is desirable to create test cases with respect to a given set of requirement properties. The approach described by Engels et al. [2] can be used for this, if requirement properties are used as test purposes. The drawback is that each requirement property only results in one test case. This test case is not necessarily a good exercise regarding the property. For example, consider the property $\Box\,(x \rightarrow \bigcirc y)$, which is quite a common type. A counterexample might not contain a state where x is true, which obviously is not a good test case for the property. A straight forward approach is to require the antecedent to become true in a test case. For example, this is achieved with antecedent coverage [66], where $\Box\,(x \rightarrow \bigcirc y)$ is reformulated to $\Box\,(x \rightarrow \bigcirc y) \wedge \Diamond\,(x)$. Further approaches are shown below.

It is not always possible to create useful counterexamples by directly negating requirement properties. For example, negation of a safety property might result in a counterexample which consists of only one state (the initial state) — which is not a useful test case.

An equivalence partitioning of the execution tree is suggested by Callahan et al. [1, 67]. For a single requirement property, two kinds of paths can be distinguished within the expanded computation tree: those

for which the property is fulfilled, and those where the property is violated. The idea of this partitioning is that all paths within a partition are assumed to be very similar. That way, only a small number of test cases, or in fact only a single test case, per partition is necessary. A complete cover of disjoint partitions on infinite paths in the computation tree can be created by combining properties and their negations conjunctively.

For example assume two requirement properties ϕ_1 and ϕ_2. There are four different possible partitions for these two properties: $\phi_1 \wedge \phi_2$, $\phi_1 \wedge \neg\phi_2$, $\neg\phi_1 \wedge \phi_2$, and $\neg\phi_1 \wedge \neg\phi_2$. Each such combination is a *coverage property*. This partitioning, called conjunctive complementary closure (CCC), creates partitions that are only disjoint when considering complete paths in the computation tree. Finite traces may fall into one or more partitions. Coverage properties can be used to validate existing test traces, determine to which partition a given test case belongs to, or to create a new test case for a partition.

5.1 Vacuity Based Coverage

Tan et al. [34] describe a method to derive trap properties from requirement properties. These trap properties achieve that such test cases are created that show how a property is non-vacuously fulfilled. Vacuity describes the problem that a property is satisfied in a way not intended. 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 $\square(x \rightarrow \bigcirc y)$ is vacuously satisfied by any model where x is never true. A vacuous pass of a property is an indication of a problem in either the model or the property.

Beer et al. [68] use *witness formulas* to detect vacuity for a subset of ACTL (CTL with only **A** quantified temporal operators). This method is extended to CTL* by Kupferman and Vardi [69]. More efficient algorithms are considered by Purandare and Somenzi [70]. In general, vacuity of a property is detected by checking a formula and its witness formula against the model.

Witness formulas are derived from properties by changing sub-formulas. The idea is that if a model satisfies a property and also a corresponding witness formula, then the property is satisfied vacuously. If the witness formula is not satisfied by the model, then the property is properly satisfied. The replacement of sub-formula ϕ with ψ in formula f is denoted as $f[\phi \leftarrow \psi]$. If a sub-formula can be replaced such that the model does not satisfy the resulting formula, then the sub-formula *affects* the formula:

Definition 15 (Affect) *[34] A sub-formula ϕ of f affects f in model M if there is a formula ψ such that the truth value of f and $f[\phi \leftarrow \psi]$ are different with respect to M.*

If a property f is vacuously satisfied by a model, then there exists a sub-formula ϕ in f that does not affect the property. This means that there exists no replacement ψ for ϕ such that $f[\phi \leftarrow \psi]$ is violated by a given model. Consequently, a property is satisfied vacuously iff the formula and its witness formula are both satisfied by the same model:

Definition 16 (Vacuity) *[34] A model M satisfies f vacuously with respect to a sub-formula ϕ if $M \models f$ and ϕ doesn't affect f in M. M satisfies f vacuously if there exists a sub-formula ϕ such that M satisfies f vacuously with respect to ϕ.*

The replacement formula ψ can be any formula. Fortunately it is not necessary to replace ϕ with every possible ψ in order to detect vacuity. Kupferman and Vardi [69] show that it is sufficient to replace ϕ with $true$ or $false$, depending on the *polarity* of ϕ in the formula f. The polarity of a sub-formula ψ is positive, if it is nested in even number of negations in f, otherwise it is negative. The polarity of a sub-formula ϕ is denoted as $\square(\phi)$. To avoid confusion with the LTL $\square$ operator, it is noted that $\square$ in this section always

refers to the polarity. Replacement of ϕ according to its polarity makes it feasible to determine vacuity using witness formulas.

Theorem 1 *[69] A model M satisfies the formula f vacuously if and only if $M \models \neg f[\phi \leftarrow \Box(\phi)]$ for some (occurrence of) atomic proposition ϕ, where $\Box(\phi) = false$ if a has positive polarity in f and $\Box(\phi) = true$ otherwise.*

The idea of property coverage is that a test case that covers a property according to the property coverage criterion should not pass on any model that does not satisfy the property.

Definition 17 (Property-Coverage Metrics) *[34] Given a property f, a test t covers a sub-formula ϕ of f if there is a mutation $f[\phi \leftarrow \Box(\phi)]$ such that every model M that passes t will not satisfy the formula $f[\phi \leftarrow \Box(\phi)]$.*

The property coverage can be measured by creating a set of witness formulas. The percentage of these witness formulas that are violated by at least one test case represents the property coverage value. Details of how coverage is measured on existing test cases is given in Section 8. Furthermore, test cases can be generated by using witness formulas as trap properties, following the approach shown in Figure 5. For every requirement property f there is a trap property for every sub-formula ϕ of the following type:

$$f[\phi \leftarrow \Box(\phi)]$$

For example, requirement 1 of the CC example model (Equation 13, $\Box\,(\texttt{brake} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop})$) results in the following trap properties (note that the polarity of `brake` is $false$ because of the implication):

$$\Box\,(false \rightarrow \bigcirc \texttt{velocity} = \texttt{stop})$$
$$\Box\,(\texttt{brake} \rightarrow \bigcirc true)$$

5.2 Unique First Cause Coverage

Whalen et al. [66] adapt the MC/DC criterion to apply to LTL requirement properties as a metric called *Unique-First-Cause Coverage*. While MC/DC only applies to states that fulfill certain requirements regarding the valuation of conditions in control flow branches, LTL properties define paths rather than states. Whalen et al. define MC/DC via sets of Boolean expressions for decision assignments, and then refine these sets with temporal operators.

Given a decision A, A^+ denotes the set of expressions necessary to show that all conditions in A positively affect the outcome of A; that is, where A evaluates to true as a consequence of a considered condition. A^- denotes the set of expressions necessary to show that all conditions in A negatively affect the outcome of A. In the following definition, x denotes a basic condition.

Definition 18 (Expressions for MC/DC)

$$
\begin{array}{lcl}
x^+ & = & \{x\} \\
x^- & = & \{\neg x\} \\
(A \wedge B)^+ & = & \{a \wedge B \,|\, a \in A^+\} \cup \{A \wedge b \,|\, b \in B^+\} \\
(A \wedge B)^- & = & \{a \wedge B \,|\, a \in A^-\} \cup \{A \wedge b \,|\, b \in B^-\} \\
(A \vee B)^+ & = & \{a \wedge \neg B \,|\, a \in A^+\} \cup \{\neg A \wedge b \,|\, b \in B^+\} \\
(A \vee B)^- & = & \{a \wedge \neg B \,|\, a \in A^-\} \cup \{\neg A \wedge b \,|\, b \in B^-\} \\
(\neg A)^+ & = & A^- \\
(\neg A)^- & = & A^+
\end{array}
$$

The set of expressions necessary to cover a decision is determined by recursively applying the above rules. For example, the expression $x \vee (y \wedge z)$ results in the following set to show positive affect: $\{(x \wedge \neg(y \wedge z)), (\neg x \wedge (y \wedge z))\}$. The set to show negative affect is: $\{(\neg x \wedge \neg(y \wedge z)), (\neg x \wedge (\neg y \wedge z)), (\neg x \wedge (y \wedge \neg z))\}$. A requirement for a test suite to satisfy MC/DC of a decision $x \vee (y \wedge z)$ is that each constraint in these two sets is satisfied by a test case.

As LTL formulas are defined on paths and not states, the above rules need to be extended to take temporal operators into consideration, resulting in the unique-first-cause coverage (UFC) criterion. A test suite satisfies UFC, if it achieves that every basic condition in a formula takes on all possible outcomes at least once, and each basic condition is shown to independently affect the formula's outcome. Assuming a formula A and a path π, a condition c is the unique first cause of A, if in the first state along π where A is satisfied, it is satisfied because of c. The following rules are defined in [66]:

Definition 19 (Expressions for UFC)

$$
\begin{array}{lcl}
\Box\,(A)^+ & = & \{A \,\mathsf{U}\, (a \wedge \Box\,(A)) \,|\, a \in A^+\} \\
\Box\,(A)^- & = & \{A \,\mathsf{U}\, a \,|\, a \in A^-\} \\
\Diamond\,(A)^+ & = & \{\neg A \,\mathsf{U}\, a \,|\, a \in A^+\} \\
\Diamond\,(A)^- & = & \{\neg A \,\mathsf{U}\, (a \wedge \Box\,(\neg A)) \,|\, a \in A^-\} \\
\bigcirc\,(A)^+ & = & \{\bigcirc\,(a) \,|\, a \in A^+\} \\
\bigcirc\,(A)^- & = & \{\bigcirc\,(a) \,|\, a \in A^-\} \\
(A \,\mathsf{U}\, B)^+ & = & \{(A \wedge \neg B) \,\mathsf{U}\, ((a \wedge \neg B) \wedge (A \,\mathsf{U}\, B)) \,|\, a \in A^+\} \cup \\
 & & \{(A \wedge \neg B) \,\mathsf{U}\, b \,|\, b \in B^+\} \\
(A \,\mathsf{U}\, B)^- & = & \{(A \wedge \neg B) \,\mathsf{U}\, (a \wedge \neg B) \,|\, a \in A^-\} \cup \\
 & & \{(A \wedge \neg B) \,\mathsf{U}\, (b \wedge \neg(A \,\mathsf{U}\, B)) \,|\, b \in B^-\}
\end{array}
$$

For example, the simple property $\phi = \Box\,(x \wedge y)$ results in the constraints $\phi^+ = \{(x \wedge y) \,\mathsf{U}\, ((x \wedge y) \wedge \Box\,(x \wedge y)\}$ and $\phi^- = \{((x \wedge y) \,\mathsf{U}\, (\neg x \wedge y)), ((x \wedge y) \,\mathsf{U}\, (x \wedge \neg y))\}$. These constraints can be used to create test cases with a model checker. As always, it is necessary to negate the constraints to be valid trap properties. This results in the following type of trap properties, where for each $f \in \phi^+, \phi^-$ one trap property is created:

$$\Box \neg f$$

Test cases can be derived by using the usual trap property based approach shown in Figure 5.

As another example, let ϕ be the requirement 1 of the CC example model (Equation 13): $\phi = \square(\texttt{brake} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop})$. For this property, the following two trap properties result for positive affect:

$$\neg(\texttt{brake} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop})\,\mathsf{U}\,(\neg\texttt{brake} \wedge \neg(\bigcirc \texttt{velocity} = \texttt{stop}) \wedge \square(\texttt{brake} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop}))$$

$$\neg(\texttt{brake} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop})\,\mathsf{U}\,(\texttt{brake} \wedge (\bigcirc \texttt{velocity} = \texttt{stop}) \wedge \square(\texttt{brake} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop}))$$

The following two trap properties result for negative affect:

$$\neg(\texttt{brake} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop})\,\mathsf{U}\,(\texttt{brake} \wedge \neg(\bigcirc \texttt{velocity} = \texttt{stop}))$$

$$\neg(\texttt{brake} \rightarrow \bigcirc \texttt{velocity} = \texttt{stop})\,\mathsf{U}\,(\texttt{brake} \wedge (\bigcirc \neg(\texttt{velocity} = \texttt{stop})))$$

LTL semantics are defined for infinite traces, while test cases are finite. Therefore, Whalen et al. [66] refine the rules to derive expression sets, which is mainly of interest for test suite analysis.

5.3 Dangerous Traces

There are several approaches based on mutation, where test cases are created with regard to requirement properties. Although mutation based approaches are considered in Section 6, we now consider the idea of *dangerous traces*, introduced by Ammann et al. [71]. In this approach, mutation is used to find scenarios where a dangerous action is either inevitable or possible as of the next state or at some point in the future.

An action is said to be dangerous if it can lead to a safety property violation. As a correct implementation may never violate safety properties, a fault model is used to describe how safety properties are violated. The fault model is used to create *mutants* of a behavioral model, where each mutant is a simple syntactically valid variation of the original model. The fault represented by the mutant model M' can result in four different types of dangerous traces:

- A trace is **AX** *dangerous*, if the additional transitions allowed by the mutant M' violate a property P in all next states after executing the mutated transition.
- A trace is **EX** *dangerous*, if there exists an additional transition allowed by the mutant M' which violates a property P in the next state.
- A trace is **AF** *dangerous*, if it can be extended with the next state from M' and other transitions from the combined model so that in future there always is a violation of P.
- A trace is **EF** *dangerous*, if it can be extended with the next state from M' and other transitions from the combined model so that in future there sometimes is a violation of P.

Ammann et al. [71] describe a method based on trap properties to generate test cases for all of these different types of dangerous traces. The use of a mutant model M' allows two different flavors of test cases to be produced for each dangerous trace: A failing test case that includes the faulty transition and leads to the property violation, of a passing test case, where instead of the faulty transition the correct transition is taken. To allow creation of both types of test cases, Ammann et al. [71] combine a model M with a mutant M', such that the combined model can take transitions from M or M'. A special variable `original` is used to

indicate whether a transition is part of the original model; it is false if the mutated transition is executed. This special variable allows definition of trap properties to derive test cases from the model/mutant combination. In accordance with the original paper [71], we here state the test requirements, which simply have to be negated to be used as trap properties.

For example, a failing test case for an **AX** dangerous trace requires a sequence along which `original` is true up to a state, where all next states where `original` is false violate P:

$$\mathbf{EF}\,(\texttt{original} \wedge \mathbf{EX}\,(\neg\texttt{original}) \wedge \mathbf{AX}\,(\neg\texttt{original} \rightarrow \neg P))$$

Creation of a passing test case is a little bit more tricky, and exploits the behavior of CTL counterexample creation algorithm. An **EX** `original` expression is added at the proper place in the test requirement in order for the counterexample to contain an original transition instead of a mutated one:

$$\mathbf{EF}\,(\texttt{original} \wedge \mathbf{EX}\,(\texttt{original}) \wedge \mathbf{EX}\,(\neg\texttt{original}) \wedge \mathbf{AX}\,(\neg\texttt{original} \rightarrow \neg P))$$

The test requirements for **EX** dangerous failing and passing traces are:

$$\mathbf{EF}\,(\texttt{original} \wedge \mathbf{EX}\,(\neg\texttt{original} \wedge \neg P))$$
$$\mathbf{EF}\,(\texttt{original} \wedge \mathbf{EX}\,(\texttt{original}) \wedge \mathbf{EX}\,(\neg\texttt{original} \wedge \neg P))$$

The test requirements for **AF** dangerous failing and passing traces are (partially abbreviated to fit in a line):

$$\mathbf{EF}\,(\texttt{original} \wedge \mathbf{EX}\,(\neg\texttt{original}) \wedge \mathbf{AX}\,(\neg\texttt{original} \rightarrow (\neg P \vee \mathbf{AF}\,(\neg P))))$$
$$\mathbf{EF}\,(\texttt{orig} \wedge \mathbf{EX}\,(\texttt{orig}) \wedge \mathbf{EX}\,(\neg\texttt{orig}) \wedge \mathbf{AX}\,(\neg\texttt{orig} \rightarrow (\neg P \vee \mathbf{AF}\,(\neg P))))$$

The test requirements for **EF** dangerous failing and passing traces are:

$$\mathbf{EF}\,(\texttt{original} \wedge \mathbf{EX}\,(\neg\texttt{original}) \wedge \mathbf{EF}\,(\neg P))$$
$$\mathbf{EF}\,(\texttt{original} \wedge \mathbf{EX}\,(\texttt{original}) \wedge \mathbf{EX}\,(\neg\texttt{original}) \wedge \mathbf{EF}\,(\neg P))$$

Given mutants of the CC example model (Figure 4), any of the requirement properties given in Section 3 can serve as P in the trap properties listed above.

The overall process for test case generation, depicted in Figure 10, consists of deriving a set of trap properties according to the desired dangerous traces for each considered safety property as well as creating a set of mutants. Each mutant is combined with the original model, and then checked against the trap properties.

5.4 Property Relevance

A related approach to dangerous traces was presented by Fraser and Wotawa [72]. Here, *property relevance* is introduced as a relationship between test cases and requirement properties. A failing test case is relevant to a property, if the erroneous behavior described by the test case violates the property. In contrast, positive test cases have to satisfy requirement properties as they are created from a correct model. Therefore, a passing test case is relevant to a property, if the test case execution *could* lead to a property violation on an erroneous implementation. The possible deviation is simulated according to a fault model or simple mutation. Based on the notion of property relevance, it is shown in [72] how any structural coverage criterion can be combined with property relevance:

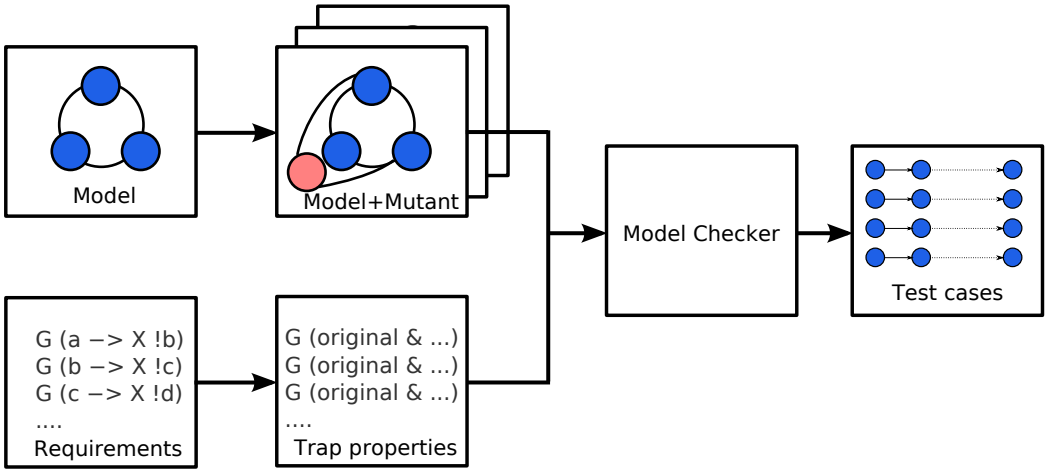


Figure 10: Dangerous traces for safety properties.

Definition 20 (X Property Relevance Coverage) *The property relevance coverage C_R of a test suite TS with regard to a set of properties* P *and a structural coverage criterion* X *represented as a set of trap properties* T *is defined as the ratio of trap properties that are covered such that the covering test case continues relevantly to a property, to the total number of possible property/trap property combinations:*

$$C_R = \frac{1}{|\mathsf{P}| * |\mathsf{T}|} \cdot |\{p, tr \mid p \in \mathsf{P} \wedge tr \in \mathsf{T} \wedge relevant\_covered(tr, p, \mathsf{TS})\}|$$

The predicate $relevant\_covered(a, b, TS)$ is true if there exists a test case $t \in TS$ such that t consists of two sub-sequences $t := t_1, t_2$ where t_1 covers a, i.e., $t_1 \not\models a$, and t_2 is relevant to b, i.e., $relevant(t_2, b)$.

For example, *Transition Property Relevance Coverage* requires that for each transition and each requirement property there is a test case that executes the transition and then proceeds relevant to the property. In [72], methods to measure property relevance and property relevant coverage are presented. These methods extend the general approach of coverage measurement with model checkers, which is described in Section 8. To simplify the measurement procedure, a weakened variant of property relevant coverage (Weak X Property Relevance Coverage) is defined, which relaxes the requirement on the order in which a structural item and a property have to be covered.

The approach taken in [72] to create property relevance test suites is to first create a complete test suite for the structural coverage criterion using a traditional trap property based approach. This test suite can then be optimized to simplify the second step, which is to extend each test case with a property relevant postamble. For this extension, the original model and a model that can take erroneous transitions are combined, such that they share the identical prefixes. The initial state of these models corresponds to the considered structural coverage test case. The model checker is now used to create a trace that shows how the erroneous model violates the requirement property; this is achieved by simply checking the property using the outputs of the erroneous model. As correct and erroneous model share the same inputs, the trace created by the correct model represents the correct behavior which could lead to the property violation. This trace is used to extend the existing test case. The erroneous model can be a mutant model, according to some fault model. As the number of mutants is potentially very high, Fraser and Wotawa [72] suggest a special kind of mutant which can nondeterministically choose exactly one erroneous transition along any execution trace.

6 MUTATION BASED TEST CASE GENERATION

In general, mutation describes the modification of a program according to some fault model. Mutation analysis describes the process of evaluating an existing test suite with regard to its ability to identify mutants. Mutation testing is the process of deriving test cases that identify as many mutants as possible. The idea of mutation is based on the *coupling effect* [73] and *competent programmer hypothesis* [74]. The former states that tests that detect simple faults are likely to also detect complex faults, while the latter states that programs are close to being correct.

Originally, mutation testing was applied to source code [73, 74]. Specification mutation was initially introduced by Budd and Gopal [75]. In the context of model checker based testing, specification mutation was introduced by Ammann and Black [76] for coverage analysis, and the use for test case generation was initially suggested by Ammann et al. [77]. There are related approaches where specifications are mutated; e.g., Srivatanakul et al. [78] apply mutation together with model checking. Here, however, only approaches where the aim is test case generation are considered.

A *competent specifier hypothesis* is assumed, which resembles the competent programmer hypothesis and states that specifications are close to what is actually desired. If specifications are interpreted as abstract programs, the coupling effect can be assumed as well.

6.1 Mutation Operators for Specifications

Although mutation can be applied to automaton models directly, the prevalent method is to mutate the textual representations of models, for example in the input language of the model checker used for test case generation.

In general, a mutation operator describes a syntactic change according to a fault model. The mutation operator can be applied to different locations in the specification, each application resulting in a specification mutant. Usually only first order mutants are considered, that is, mutants that differ from the original version by only one mutation.

Mutation operators for specifications are analyzed by Black et al. [79]. The examples given in [79] use the syntax of the model checker SMV [24], but can be applied to language that uses similar logical expressions. For example, the same mutation operators can also be applied to LTL or CTL properties.

The following mutation operators are defined and evaluated with regard to coverage in [79]. Example mutation operators are illustrated with the line `accelerate & !brake & velocity = stop: slow` of the CC example SMV code (Figure 4):

Logical Operator Replacement (LRO) : This mutation operator replaces a logical operator with another logical operator.

```
accelerate | !brake & velocity = stop:  slow
```

Relational Operator Replacement (RRO) : This mutation operator replaces a relational operator with another relational operator.

```
accelerate & !brake & velocity > stop:  slow
```

Expression Negation Operator (ENO) : This operator negates sub-expressions.

```
accelerate & !(!brake & velocity = stop ):  slow
```

Simple Expression Negation (SNO) : This operator negates an atomic condition in a decision.

```
!accelerate & !brake & velocity = stop:  slow
```

Operand Replacement Operator (ORO) Changes variables or constants with other syntactically valid operands. For example:

Variable Replacement Operator (VRO) : This operator replaces a variable reference with a reference to another variable of the same type.

```
accelerate & !accelerate & velocity = stop:  slow
```

Constant Replacement Operator (CRO) : This operator replaces a constant with a syntactically valid different constant.

```
accelerate & !brake & velocity = stop:  stop
```

Missing Condition Operator (MCO) : This operator removes a single condition from a decision.

```
accelerate & _  & velocity = stop:  slow
```

Stuck At Operator (STO) : This operator replaces a condition with true or false (1 or 0).

```
1  & !brake & velocity = stop:  slow
```

Associative Shift Operator (ASO) Changes the association between variables. For example, assume the original line would be: `(accelerate | !brake) & velocity = stop:  slow` . A possible mutant would then be the following: `accelerate | (!brake & velocity = stop): slow` .

Arithmetic Operator Replacement (ARO) : This mutation operator replaces an algebraic operator with another algebraic operator. For the sake of this example, assume that instead of setting it to `slow`, `velocity` is an integer variable and is increased by 2:

```
accelerate & !brake & velocity=stop:  velocity+2
```

A mutant could be:

```
accelerate & !brake & velocity=stop:  velocity-2
```

6.2 Specification Mutation

Ammann et al. [77] initially proposed specification mutation for test case generation. An SCR specification is converted to an SMV model and a set of temporal logic constraints, both of which represent the mode transitions. Mutation is applied to both the textual description of the model and the requirement properties. Initially, the model satisfies all temporal logic constraints. Mutating either the model or the constraints might lead to property violations.

The first option is to verify mutant models with regard to the temporal logic constraints. For each mutant model there is one counterexample for every constraint that is not satisfied. A resulting counterexample illustrates how an erroneous implementation would behave, therefore a correct IUT is expected to behave differently when a resulting test case is executed on it. Consequently, such traces can be used as negative test cases, i.e., a fault is detected if an IUT behaves identically.

As second option, mutation of the temporal logic constraints can result in properties that are not satisfied by the original model, and can also be used to create counterexamples. As traces are created from the original model, resulting test cases are positive test cases.

A mutant is *equivalent*, if it behaves identically to the original model. Equivalence can be further constrained by requiring the observable behavior to be identical to that of the original, that is, the output values have to be identical at all times. Equivalent mutants do not result in counterexamples.

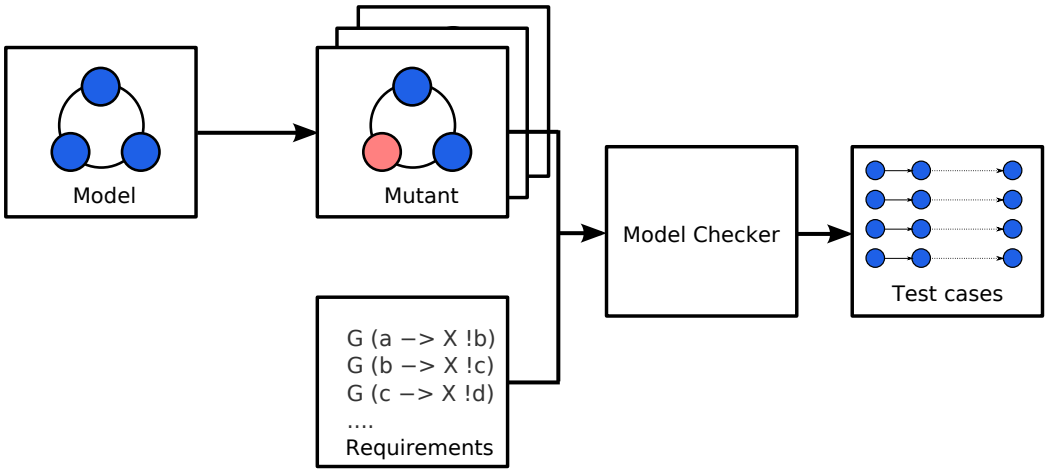


Figure 11: Test cases from mutants violating the specification.

Fraser and Wotawa [80, 81] take a similar approach to generate test cases, but instead of creating a model and properties that represent the same SCR specification, a behavioral model and a set of requirement properties derived from formalizing user requirements is used. The original model is assumed to satisfy all requirement properties. Figure 11 depicts the process of deriving test cases with requirement properties. The same process applies if the properties are derived from an SCR specification. Negative test cases illustrate requirement property violations, therefore test cases are traceable to requirement properties. Traceability with property mutants is not always possible, as a mutation can completely change the meaning of a property. However, with only a certain restricted subset of mutation operators (e.g., RRO, LRO, SNO, ENO), property mutants are related to the original properties. In general, the percentage of property mutants that do not result in counterexamples is higher than for model mutants. With this approach, a model mutant that creates no counter examples is not necessarily equivalent; the specification might just be too weak to detect the change.

The approach presented in [71] can also be seen as related to this approach. Here, the model is also mutated and verified with regard to requirement properties. As described in Section 5.3, the objective is to derive dangerous traces with regard to safety properties. For this, the original and mutant model are merged, so that the combined model can take both, the original and the mutated transition. The process of merging a model and its mutant is illustrated with the language SMV in [71]. A special variable `original`, which is only false if the mutated transition is taken, is added to the model. This is used to create special trap properties based on the requirement properties, as described in Section 5.3.

6.3 Reflection

Ammann and Black [76] create logical formulas that "reflect" the transition relation of a model; this process is called *reflection*. These reflected properties resemble the logical properties derived from SCR specifications, as described in [77] and the previous section. With regard to the transition system definition given in

Section 4.2, there is one such reflected property for each simple transition:

$$\Box\,(\alpha \wedge \gamma \rightarrow \bigcirc \beta)$$

The reflection process is straight forward in principle, but there are several subtle issues when applying it to a concrete modeling language. For example, in the language of the model checker SMV there is an implicit semantics based on the syntactic ordering of case statements, which has to be resolved as there is no ordering for properties. To overcome this problem, it is necessary to make the implicit information contained in the ordering of the transitions explicit. In [76], this process is called *expoundment*. Basically, instead of using each antecedent $condition_i$ (which represents $\alpha \wedge \gamma$) as such, they are converted to $(\bigwedge_{1 \leq j < k} \neg condition_j) \wedge condition_k$ for $condition_k$.

The CC example NuSMV model (Figure 4) results in the following reflected properties (simplified; automatic expoundment might result in more complex but logically identical properties, especially for the default branch):

$$\Box\,((\texttt{accelerate} \wedge \neg\texttt{brake} \wedge \texttt{velocity} = \texttt{stop}) \rightarrow \bigcirc\, \texttt{velocity} = \texttt{slow})$$
$$\Box\,((\texttt{accelerate} \wedge \neg\texttt{brake} \wedge \texttt{velocity} = \texttt{slow}) \rightarrow \bigcirc\, \texttt{velocity} = \texttt{fast})$$
$$\Box\,((\neg\texttt{accelerate} \wedge \neg\texttt{brake} \wedge \texttt{velocity} = \texttt{fast}) \rightarrow \bigcirc\, \texttt{velocity} = \texttt{slow})$$
$$\Box\,((\neg\texttt{accelerate} \wedge \neg\texttt{brake} \wedge \texttt{velocity} = \texttt{slow}) \rightarrow \bigcirc\, \texttt{velocity} = \texttt{stop})$$
$$\Box\,(\texttt{brake} \rightarrow \bigcirc\, \texttt{velocity} = \texttt{stop})$$

There is one more simple transition that needs to be covered – the default branch. Here, two things have to be considered: First, the antecedent is not explicitly available, but is the conjunction of the negations of all earlier antecedents. Second, the NuSMV model states that `velocity` does not change. To represent this as a temporal logic property, an auxiliary variable `P_velocity` is necessary, which is defined as follows:

```
VAR
  ..
  velocity: boolean;
  P_velocity: boolean;
...
ASSIGN
  next(P_velocity) := velocity;
```

In our example, this results in the following property (simplified):

$$\Box\,(((\texttt{accelerate} \wedge \neg\texttt{brake} \wedge \texttt{velocity} = \texttt{fast}) \vee (\neg\texttt{accelerate} \wedge \neg\texttt{brake} \wedge \texttt{velocity} = \texttt{stop})) \rightarrow \bigcirc\, \neg(\texttt{P\_velocity} = \texttt{velocity}))$$

Once a set of reflected properties is derived, mutation can be applied to these properties. In [76], the resulting mutants are used to determine the mutation adequacy of a given test suite. In fact, the mutants of the reflected properties can be used just like coverage related trap properties [82] in order to generate test cases. Figure 12 depicts the process of test case generation with reflection. If the mutant property describes a transition that does not exist in the actual model, then the model checker returns a counterexample that takes the correct transition. The mutant properties can be greatly varied by applying different mutation operators. An invaluable source of information for this approach is [55].

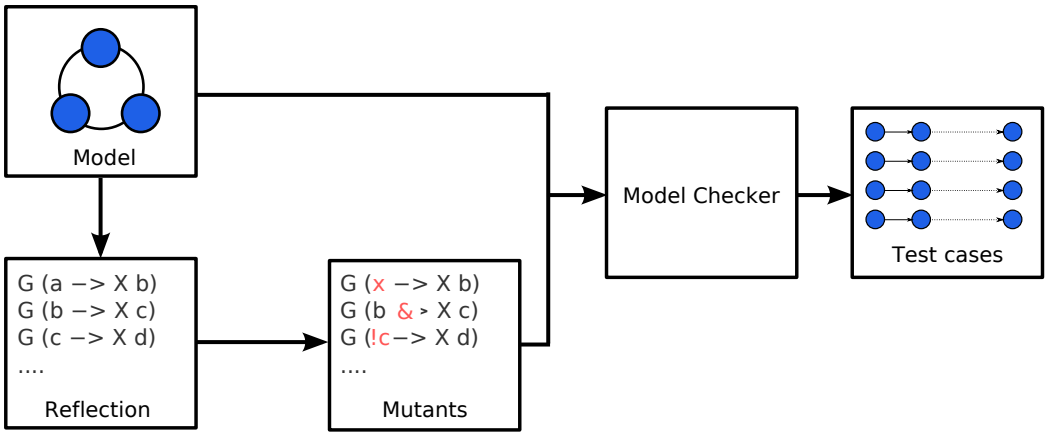


Figure 12: Mutation based test case generation with reflection.

Gargantini [83] proposed a related approach based on Abstract State Machine specifications (introduced in Section 4.4). Guard conditions of update rules are mutated according to a given fault model. From the mutated conditions, *detection conditions* are derived. The idea is that a fault in a Boolean expression can be discovered if the detection condition evaluates to true. For a given Boolean expression ϕ and a mutant ϕ', the detection condition is $\phi \oplus \phi'$. The operator $\oplus$ is the xor-operator, which means that the detection condition is only true if ϕ and ϕ' have different values. Test cases are generated by converting an ASM specification to a SPIN or SMV model. The considered guard conditions are mutated, and a set of detection conditions is created by combining each mutant ϕ' with its original condition ϕ as $\phi \oplus \phi'$. Trap properties are created by negating the detection conditions; i.e., claiming that they are never true. As ASM specifications can be hierarchic, additional outer guard conditions have to be included in the trap property:

$$\Box\,(A \rightarrow \neg(\phi \oplus \phi'))$$

Here, A denotes the conjunction of the outer guard conditions. The property is equivalent to $\Box\,(A \rightarrow (\phi \leftrightarrow \phi'))$. Test cases can be derived as usual by checking the trap properties against the model.

Considering the example ASM specification of the CC model, given in Figure 9, assume a mutant of the guard of rule 1 from `(velocity == stop) && (accelerate == True) && (brake == False)` to `not(velocity == stop) && (accelerate == True) && (brake == False)`. This results in the following trap property:

$$\Box\,(\neg((\texttt{velocity} = \texttt{stop} \wedge \texttt{accelerate} \wedge \neg\texttt{brake}) \oplus ((\texttt{velocity} \neq \texttt{stop} \wedge \texttt{accelerate} \wedge \neg\texttt{brake})))$$

6.4 State Machine Duplication

Okun et al. [84] identified the problem that when using mutation in the reflection approach there is no guarantee that a test case propagates a fault to an observable output. As one possible solution, *In-line expansion* is proposed. In-line expansion considers only reflections of the transition relations of output variables. In these reflections, internal variables are replaced with in-line copies of their transition relations. This replacement is repeated until the formula references no more internal variables. In-line expansion results in very efficient, but also very large test suites, as the number of mutants can increase quite significantly.

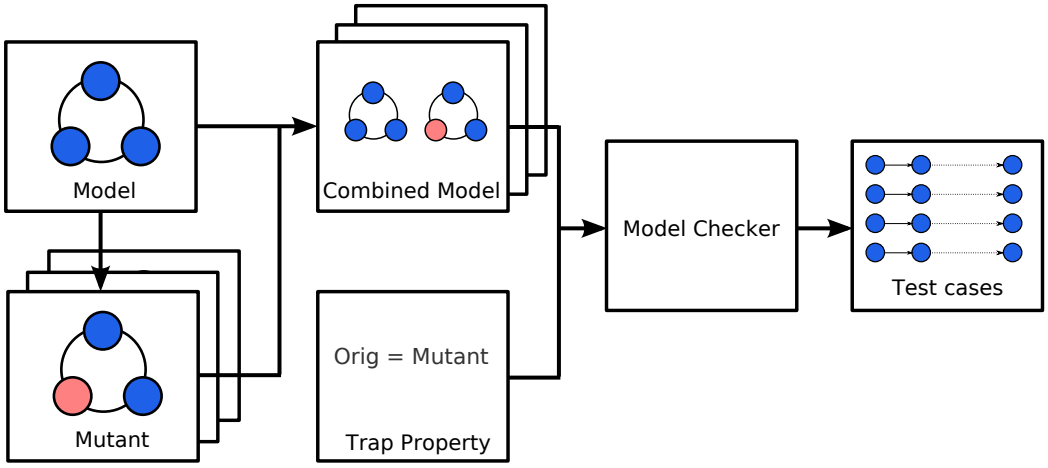


Figure 13: State machine duplication based test case generation.

As an alternative, an approach called *state machine duplication* is proposed in [84]. This approach is based on model mutation, but uses an equivalence checking method to derive counterexamples. As illustrated in Figure 13, for each mutant model, a combined model where mutant and original model are executed in parallel is created. Both original and mutant model share the identical input values, therefore inequivalence can be shown with a trace where the output values differ. The model checker can easily be used to create such a trace, by verifying a property of the following type for each output variable out, or alternatively creating the conjunction of all output variables:

$$\Box\,(original.out = mutant.out)$$

In the CC example model, the following property would be used:

$$\Box\,(\texttt{original.velocity} = \texttt{mutant.velocity})$$

Boroday et al. [35] use this approach in the formal setting of modules as described in Section 2. In this setting, the composition of specification module S and mutant M with outputs O results in a counterexample if:

$$S||M \not\models \Box \bigwedge_{p \in O} (p = p')$$

If the mutant is equivalent to the original model with regard to the outputs, then the combined model satisfies these properties. If the mutant is not equivalent, then each such property results in a counterexample usable as a test case where the fault is propagated to an output. In practice, creation of a test suite with state machine duplication takes longer than with reflection, because there is the overhead of creating the combined models, and calling the model checker separately on each combined model; mutants of reflected properties can be verified in a single run of the model checker. Experiments [84] have, however, shown that test suites created with state machine duplication are better.

7 ISSUES IN TESTING WITH MODEL CHECKERS

Testing with model checkers is an active area of research, and as such there are many issues that still need to be solved. The main showstopper for industry acceptance of model checker based testing is probably the limited performance. A main cause of this problem is the state explosion problem, but there are other issues contributing to a potentially bad performance. Even if the performance is acceptable, the results of the test case generation might not be as good as possible. Some application scenarios, like regression testing, need special treatment. Nondeterministic models or properties that require nonlinear counterexamples are further examples of issues with model checker based testing. This section reviews identified problems and proposed solutions.

7.1 Abstraction

The main cause for performance problems with model checkers is the state explosion, which signifies the large or intractable state spaces that can easily result from complex models. Especially software model checking is susceptible to the state explosion problem. Abstraction is a popular method to overcome the state explosion problem. Abstraction is an active area of research, and many abstraction techniques have been presented in recent years. This has made it possible to verify properties on very large models. In general, abstraction methods are tailored towards verification, and therefore are not always useful in the context of testing.

A full survey of available techniques is out of the scope of this document; as an example technique, we mention counterexample guided abstraction refinement (CEGAR) [85], which refines an abstract model until no more spurious counterexamples are generated when verifying a property. This method ensures soundness, which means that a property that holds on the abstract model also holds on the concrete model. In contrast, when generating test cases with a model checker, the objective is different: Properties that are violated by a concrete model should also be violated by the abstract model.

Ammann and Black [86] define a notion of soundness in the context of test case generation, which expresses that any counterexample of an abstracted model has to be a valid trace of the original model. A method called *finite focus* is proposed and shown to be sound with regard to this soundness definition. Finite focus only considers a limited set of states, for example only a fixed subset of variables of large or unbounded domains. An additional state machine is defined, which changes from state *sound* to *unsound* whenever a transition is taken that is out of the finite focus. Once the unsound state is reached, this state machine stays in this state.

Constraint rewriting rules are defined, which basically rewrite temporal operators such that they evaluate to true when an unsound state is reached. In [86] this rewriting is defined for CTL, and a correctness proof is given. The same rules apply to LTL properties. Consequently, the constraint rewriting $CR(\phi)$ for an LTL/CTL property ϕ is recursively defined as follows, where v denotes a Boolean value, and s is a special variable that is true if the state is sound or otherwise false. OP denotes any of the LTL operators $\Box$, $\bigcirc$, $\Diamond$, or when considering CTL properties **AG** , **AF** , **AX** , **EG** , **EF** , **EX** . The operator OP_U stands for either **A** or **E** in the context of the CTL until operator, or is a blank placeholder in the case of LTL. Atomic propositions are denoted by a.

Definition 21 (Constraint Rewriting)

$$CR(\phi) = \begin{cases} cr(\phi, True) & \textit{if } \phi \textit{ begins with a temporal operator} \\ s \rightarrow cr(\phi, True) & \textit{else.} \end{cases}$$

$$
\begin{array}{lcl}
cr(a, v) & = & a \\
cr(\neg\phi, v) & = & \neg cr(\phi, \neg v) \\
cr(\phi_1 \wedge \phi_2, v) & = & cr(\phi_1, v) \wedge cr(\phi_2, v) \\
cr(\phi_1 \vee \phi_2, v) & = & cr(\phi_1, v) \vee cr(\phi_2, v) \\
cr(\phi_1 \rightarrow \phi_2, v) & = & cr(\phi_1, \neg v) \rightarrow cr(\phi_2, v) \\
cr(\phi_1 \equiv \phi_2, v) & = & cr(\phi_1, v) \equiv cr(\phi_2, v) \\
cr(OP\ \phi, True) & = & OP\ (s \rightarrow cr(\phi, True)) \\
cr(OP\ \phi, False) & = & OP\ (s \wedge cr(\phi, False)) \\
cr(OP_U\ \phi_1\, \mathsf{U}\, \phi_2, True) & = & OP_U\ \phi_1\ \mathsf{U}\ \phi_2 \rightarrow cr(\phi_2, True)) \\
cr(OP_U\ \phi_1\, \mathsf{U}\, \phi_2, False) & = & OP_U\ \phi_1\ \mathsf{U}\ \phi_2 \wedge cr(\phi_2, False))
\end{array}
$$

When creating test cases from a model which is abstracted with the finite focus method, this constraint rewriting has to be applied to all properties involved in the test process, that is, trap properties, reflected properties, etc. Any counterexample created from such a rewritten property is sound with regard to the abstraction. This means that the test case applies to the abstracted and the original model. A property where the constraint rewriting has been applied might be satisfied by the abstract model, while the original model would result in a counterexample. Therefore, the number of test cases on an abstract model is usually smaller. This shows that abstraction can not only be used to increase the performance of the test case generation, but also as a means to control the size of resulting test suites.

7.2 Improving the Test Suite Generation Process

One main cause for bad performance during test case generation is the model checker itself. Improvement of model checking techniques is an important area of research. For example, a case study by Heimdahl et al. [53] showed that bounded model checking can be superior for test case generation, at least for certain models and coverage criteria. As another example, directed model checking [17] is a recently proposed technique, which is of interest to testing with model checkers, because its aim is the efficient generation of counterexamples and not exhaustive verification. An overview of current research to improve model checking is out of the scope of this document. As another example of how the performance can be improved, abstraction techniques have been considered in the previous subsection.

Both coverage and mutation based approaches to test case generation call the model checker far more often than really necessary, as identified by Hong and Ural [87], Fraser and Wotawa [88], and Zeng et al. [89]. For example, consider a coverage criterion that is represented by a set of trap properties T. Traditionally, the model checker is called for each trap property $t \in T$. As a consequence, many duplicate test cases are created, and many test cases are subsumed by other, longer test cases. Black and Ranville [90] describe winnowing of test cases as a means to remove such redundant test cases once a complete test suite has been generated. As described by Fraser and Wotawa [91], even test cases that are not duplicates or subsumed by other test cases can contain a significant amount of redundancy if they contain common prefixes.

In [88], it is proposed to monitor trap properties during test case generation. Each time a counterexample is generated the remaining trap properties are analyzed with regard to this new counterexample. A trap property that is already covered does not need to be considered for test case generation; it is not necessary to call the model checker on it.

As a concrete technique to perform this monitoring, LTL rewriting based upon an approach described in [92] is proposed in [88]. The proposed rewriting techniques are used in runtime verification to determine whether a given execution trace shows a property violation. Havelund and Rosu [92] claim that their rewriting engine is capable of 3 million rewritings per second, and there are approaches that try to further optimize this approach, e.g., [93–95]. The following definition gives the rewriting rules necessary to determine whether a temporal logic property ϕ for Kripke structure $K = (S, s_0, T, L)$ is violated when state $s \in S$ is observed. The application of s to ϕ is denoted as $\phi\{s\}$. A property is evaluated with regard to a trace by sequentially rewriting the property for every state in the trace. If the rewriting results in a contradiction at any state, then the property is violated. In contrast to the rewriting rules given in [92], the following definition of the rewriting does not treat the final state specially. When rewriting for runtime verification, a pass/fail verdict is expected at the end of an execution trace. In contrast, finite trace semantics are not needed for test case monitoring, because only property violations are of interest. If the rewriting after the last state results in a property, it is sufficient to know that the property is not yet covered.

Definition 22 (State Rewriting)

$$
\begin{array}{lcl}
(\Box\,\phi)\{s\} & = & \phi\{s\} \wedge \Box\,\phi \\
(\bigcirc\,\phi)\{s\} & = & \phi \\
(\Diamond\,\phi)\{s\} & = & \phi\{s\} \vee \Diamond\,(\phi) \\
(\phi_1\ \mathsf{U}\ \phi_2)\{s\} & = & \phi_2\{s\} \vee (\phi_1\{s\} \wedge (\phi_1\ \mathsf{U}\ \phi_2)) \\
(\phi_1 \wedge \phi_2)\{s\} & = & \phi_1\{s\} \wedge \phi_2\{s\} \\
(\phi_1 \vee \phi_2)\{s\} & = & \phi_1\{s\} \vee \phi_2\{s\} \\
(\phi_1 \rightarrow \phi_2)\{s\} & = & \phi_1\{s\} \rightarrow \phi_2\{s\} \\
(\phi_1 \equiv \phi_2)\{s\} & = & \phi_1\{s\} \equiv \phi_2\{s\} \\
(\neg\phi)\{s\} & = & \neg(\phi\{s\}) \\
a\{s\} & = & a \textit{ if } a \notin L(s) \textit{ else } true
\end{array}
$$

Fraser and Wotawa [96] represent mutant models as temporal logic properties, which allows application of this approach to mutation based approaches. Each mutant model is represented by a unique characteristic property. Characteristic properties are similar to the reflected properties described in Section 6, but are extended to cover all possible effects a mutation can have in a transition system. Instead of monitoring trap properties, the characteristic properties can be monitored. Whenever a characteristic property is covered by a counterexample, it is not necessary to include the mutant represented by this characteristic property in the test case generation.

When converting each counterexample to a distinct test case, the resulting test suite contains redundancy. Monitoring avoids duplicate or subsumed test cases, but different test cases might still share identical prefixes. As described in the next section, these common prefixes do not contribute to the overall fault detection ability, but consume time during test case generation and execution. This can be avoided by creating test cases incrementally instead of mapping each counterexample to a test case. This approach was initially proposed by Hamon et al. [97]. After creating a counterexample, the initial state of the model for the next verification process remains the final state of the counterexample. In [97] this is achieved by directly calling application interface functions of the model checker SAL. Incremental generation of test cases in combination with property monitoring is used in [88]. The choice of which trap property to verify next influences the length of the test cases that are generated. In [97] the trap properties are chosen randomly (in the order

provided). In [88] this is done as well, but in many cases the rewriting of trap properties leads to hints of which trap properties can lead to very short test cases. For example, if trap property $\phi = \Box(x \rightarrow \bigcirc y)$ is rewritten to $\phi' = y \wedge \Box(x \rightarrow \bigcirc y)$ after the final state of a trace, then only a single additional transition might be necessary to cover ϕ (i.e., violate it with $\neg y$).

Hong and Ural [87] use subsumption relations between items described by a coverage criterion to reduce the costs of the test case generation. An entity subsumes another entity if exercising the former guarantees exercising the latter. The time used by the test case generation is reduced by first calculating a minimal spanning set, and then only using coverage entities in this minimal spanning set to derive test cases.

Model checking is used to determine subsumption between two entities. It is assumed that the entities are represented as LTL formulas, such that a path exercises the entity, if it fulfills the LTL formula. For entities e_1 and e_2, represented by LTL formulas $ltl(e_1)$ and $ltl(e_2)$, e_1 subsumes e_2 if a model K satisfies the following property:

$$ltl(e_1) \rightarrow ltl(e_2)$$

For each coverage criterion, a different formula $ltl(e)$ has to be defined for the entities e. Hong and Ural [87] define these formulas for control and data flow coverage criteria. In [87], the subsumption relation is used to derive minimal spanning sets for coverage criteria. A spanning set for a coverage criterion is a subset of its entities, such that exercising all items in the spanning set covers all entities described by the coverage criterion. A spanning set is minimal, if there exists no spanning set with less elements.

The minimal spanning set is derived by first creating a subsumption graph, in which vertices represent coverage entities and arcs represent subsumption. Subsumption information is derived by model checking the above property for pairs of coverage entities. Strongly connected components are collapsed into one vertex, which results in a reduced subsumption graph. Let $v_i, ...v_n$ be the vertices of the reduced subsumption graph which have no incoming arc; that is, they are not subsumed. $V_1, ...V_n$ are the sets of strongly connected components of the subsumption graph corresponding to $v_1, ..., v_n$. A minimal spanning set is $\{v'_1, ..., v'_n\}$, such that $v'_i \in V_i$ for all $1 \leq i \leq n$. Hong and Ural [87] present two different algorithm to derive subsumption graphs, one requires n^2 calls to the model checker for n coverage entities and identifies all possible minimal spanning sets. The alternative algorithm reduces the complexity by only creating one possible minimal spanning set.

Monitoring avoids that the model checker is called for trap properties that are covered by the traces selected so far, whilst the subsumption approach avoids model checking of trap properties that are *always* subsumed by other trap properties. The results can be quite different, and even though monitoring can result in smaller test suites, the actual success depends on the order in which trap properties are chosen. Consequently, a combination of these approaches is conceivable.

Zeng et al. [89] collect test cases created with a model checker in a structure called *test tree*. Each counterexample is merged into the existing test tree. Identical prefixes are simply overlaid in the tree, which automatically removes duplicate or subsumed test cases. Once a complete test tree has been produced covering all test requirements, a test suite is derived as the set of paths from the tree root to a leaf. This achieves the coverage criterion used for test case generation with a minimized test suite. It is also suggested that each time a sequence is generated, the remaining test requirements are analyzed whether any of them are fulfilled. A concrete method for this is the rewriting technique presented above, and proposed in [88].

7.3 Improving the Results of the Test Suite Generation Process

Performance is a main concern for test case generation; the generation process itself needs to be sufficiently fast to be applicable to models of realistic size. However, performance is also an important factor during test case execution. If there are too many or too long test cases, execution of a test suite might not be feasible. This is even more the case when considering regression testing, where a test suite is applied repeatedly after changes in an implementation of specification. Some of the approaches described in the previous section improve the test case generation such that smaller test suites result. This section considers optimizations to existing test suites.

With high execution costs in mind, the test case generation process should ideally result in minimal test suites in the first place. A test suite can either be minimal with regard to the number of test cases, or the number of transition in the test suite. In the context of testing with model checkers, both tasks are NP-hard, as shown in [59].

Test suites created with model checkers are not minimal; in addition, they often consist of test cases that do not contribute to the fault sensitivity. For example, different trap properties might result in identical test cases. If a (passing) test case is a prefix of another test case, then it is not necessary to execute the shorter test case if the longer one is also executed; the short test case is subsumed by the longer one. (For failing tests long test cases are subsumed by shorter prefixes). Black and Ranville [90] describe several methods to remove unnecessary test cases: Clearly, duplicates and subsumed test cases can be safely removed. The *cross section* of a requirement is the ratio of test cases that satisfy a test requirement to test cases in total:

$$CS(r) = \frac{\text{\# satisfying tests}}{\text{\# tests}}$$

A test suite can be minimized by selecting those test cases, that satisfy test requirements with small cross sections. Such test cases can be identified with their *Resolution*:

$$RES(t) = \sum \frac{1}{CS(r)^2}$$

The higher the resolution of a test case is, the more small cross section requirements it fulfills. A test suite is minimized by iteratively selecting the test case with the greatest resolution that fulfills a yet unfulfilled test requirement, until all test requirements are fulfilled.

Another technique proposed in [90] is minimization, which selects a subset of a test suite that achieves a given coverage criterion. This is also known as *test suite reduction*, which is defined by Harrold et al. [98] as follows:

Given: A test suite TS, a set of requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of TS, $T_1, T_2, \ldots, T_n$, 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 problem of finding the optimal (minimal) subset is NP-hard, therefore several heuristics have been proposed [98–100]. Test suite reduction results in a new test suite, which must contain at least one test case from each subset T_i. The reduced test suite therefore consists of less test cases; this reduces the overall fault detection ability, as shown in several experiments [101–103] (although there are other claims [104]). Note that this reduction of fault sensitivity would also occur when using an optimal instead of a heuristic reduction approach.

Heimdahl and Devaraj [101] conducted their experiments in the context of model checker based test case generation. These experiments also lead to the conclusion that test suite reduction can significantly reduce the size of a test suite, but the fault detection ability suffers from this reduction.

In [91] it is shown that test suites created with model checkers often contain a significant amount of redundancy, which means that test suites are bigger than would be necessary with regard to their fault detection ability. Common prefixes are identified as a main source of redundancy. To measure the redundancy, test cases are represented as a tree, where the root node represents the initial state.

Definition 23 *Test Suite Execution Tree: Test cases $t_i = \{s_0, s_1, ...s_l\}$ of a test suite TS can be represented as a tree, where the root node equals the initial state common to all test cases: $root(TS) = s_0$. For each successive, distinct state s_j a child node is added to the previous node s_i:*

$$s_j : (s_i, s_j) \in t_i \rightarrow s_j \in children(s_i)$$

The depth of the tree equals the length of the longest test case in TS. $children(x)$ denotes the set of child nodes of node x. If there are different initial states, then a virtual root node that connects different initial states can be added, as also used by Zeng et al. [89]. When viewing test cases as a tree, redundancy exists along paths to a node that has more than one child node. Consequently, the redundancy of a test suite can be quantified as follows:

Definition 24 *Test Suite Redundancy: The redundancy R of a test suite TS is defined with the help of the execution tree:*

$$R(TS) = \frac{1}{n-1} \cdot \sum_{x \in children(root(TS))} \mathcal{R}(x) \tag{18}$$

The redundancy of the tree is the ratio of the sum of the redundancy values $\mathcal{R}$ for the children of the root-node and the number of arcs in the tree ($n-1$, with n nodes). The redundancy value $\mathcal{R}$ is defined recursively as follows:

$$\mathcal{R}(x) = \begin{cases} (|children(x) - 1|) + \sum_{c \in children(x)} \mathcal{R}(c) & if \quad children(x) \neq \{\} \\ 0 & if \quad children(x) = \{\} \end{cases} \tag{19}$$

In [91], it is proposed that test suites can be improved by splitting test cases with common prefixes and recombine them such that the common prefixes are avoided. Hence, an optimized test suite still fulfills the original test requirements for most conceivable types of test requirements, but the overall test suite size is reduced.

Finally, a technique that is used to improve the speed with which faults are detected is test case prioritization. Test case prioritization describes the task of finding an ordering of the test cases of a given test suite such that a given goal is reached faster. The test case prioritization problem is defined by Rothermel et al. [105] as follows:

Given: T, a test suite; PT, the set of permutations of T; f a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

In this definition, PT is the set of all possible orderings of T, and f is a function that yields an award value for any given ordering it is applied to. The function f represents the goal of the prioritization; for example, achievement of a coverage value as fast as possible, or improvement of the rate at which faults are detected.

In the context of testing with model checkers, prioritizing was considered by Fraser and Wotawa [106]. In general, the first step of prioritization is to analyze each test case with regard to its coverage of a certain criterion or mutation score; test case analysis with model checkers is described in Section 8. Then, the test cases simply have to be arranged in descending order according to their coverage values of mutation scores. In [106], the use of the property relevance (see Section 5.4) that links requirement properties and test cases is also proposed. In general, the prioritization reduces the average number of test cases that need to be executed in order to detect a fault.

7.4 Quality Concerns for Coverage Based Testing

Heimdahl et al. performed several experiments to evaluate model checker based testing. In [53], a case study that analyzed the scalability of test case generation with model checkers, the authors observed that several condition based coverage criteria resulted in too short test cases that are not good at detecting faults. In [107], a pilot study was conducted to investigate the suitability of condition based coverage criteria. In this experiment, test suites were generated using different condition based coverage criteria for a close to production model of a flight guidance system from Rockwell Collins Inc. The fault detection ability of the different test suites was measured on mutant versions of the model. The experiment showed that a set of randomly generated test cases generated using the same effort were superior to all coverage based test suites.

This result was due in part to a peculiar behavior of the model in use that was not considered by the specifiers, but often exploited by the bounded model checker used by Heimdahl et al., which always returns the shortest possible counterexamples. The solution applied in [107] was to define invariants to prohibit the unwanted behavior. The other conclusion drawn by Heimdahl et al. is that lazy evaluation techniques interfere with condition based coverage criteria. In general, this shows that a suitable model has a crucial influence on the result of model checker based test case generation.

In a consequent experiment, Devaraj et al. [108] showed that while coverage criteria are suitable for analysis purposes, there are problems when using them for test case generation. The identified problem is that coverage of a trap property does not guarantee that a considered part of the specification is actually executed by the resulting test case. As a solution, auxiliary variables that indicate whether some part of the specification was executed are introduced in the model and the trap properties.

An evaluation of three specification coverage criteria was performed by Abdurazik et al. [109]. Test cases were automatically created for a given example model using full predicate coverage, transition pair coverage and specification mutation coverage, which is the mutation approach based on reflected properties presented in Section 6. The resulting test cases of one criterion were evaluated with regard to the other criteria. No subsumption relations could be detected between the considered criteria. The results showed that while full predicate and specification mutation related test suites are more related with each other than transition pair test suites.

7.5 Regression Testing

Regression testing is applied when previously tested code is changed, in order to ensure that no new errors are introduced. A straight forward approach to regression testing is *retest all*. Here, all available test

cases are executed, which might be very time consuming and expensive. Therefore, *selective retesting* tries to select only a subset of the available test cases, which are sufficient to detect faults introduced with the changes. Traditionally, only changes in the source code are considered. Changes in the specification, however, also require regression testing.

Xu et al. [110] present an approach to regression testing with model checkers, where a special comparator creates properties from two versions of a model, the original version and a changed version. Each such property covers one test path that has been changed; in [110] special variables in the model are introduced to identify such paths, and the properties are implemented as assertions on these variables. It is suggested that comparators can act on different levels of abstraction. The resulting properties are verified on the changed model. Only those properties that result in counterexamples need to be considered for regression testing, properties that hold on the changed model represent test paths that do not need to be executed.

Fraser et al. [111] evaluate different techniques to create regression test cases and update existing test suites when a model is changed. In a first step, an existing test suite is analyzed to determine which test cases are still valid for the changed model and which are not. This can be done with a model checker, by either symbolically executing the test case on the new model and comparing outputs of the test case and the model, or by extracting *change properties* from the two versions of the model and then checking the test case models against these properties. A change property represents the change in the transition system, such that a test case that takes a different transition violates the property.

Once obsolete test cases have been identified, there are different approaches to create new test cases. The first approach is to determine the behavior of the changed model with regard to the input of obsolete test cases; that is, the test cases are adapted to the new model. Alternatively, sets of trap properties are generated from the old and changed model, and then the difference in these sets is calculated and used to generate test cases. Finally, property and model rewriting is proposed in [111], which lets all test cases focus on the model changes. Every resulting test case contains at least one changed transition. Test cases created with any of these approaches can be used as regression tests, and when combined with those test cases from a previous test suite that are still valid, form a new test suite.

These methods are evaluated in [111], and it is shown that there is a trade-off between time consumed for generating and executing new test cases and overall quality of a test suite after several changes. Consequently, the preferred method depends on the available resources and quality requirements.

7.6 Fault Visibility

The state of a model is defined by the values of its variables. These variables can be input or output to the system, but they can also be internal variables. Internal variables might not be directly observable. Therefore, it is important that a test case ends with some observable event or change, such that a verdict is possible. For example, trap properties for structural coverage criteria or trap properties created by the reflection approach explicitly consider the transition relations of internal variables. Such trap properties usually end with a transition where an internal variable takes on an interesting value. If this value cannot be observed, the test case does not fulfill its intended purpose.

Okun et al. [84] propose two approaches that explicitly creates such counterexamples that result in an observable change in an output variable. In-line expansion repeatedly replaces internal variables in the properties used for test case generation with their transition relations until there are no more internal variables left. This process can be applied to any kind of trap property. As an alternative, Okun et al. [84] propose state machine duplication, described in Section 6.4.

Hong et al. [58] assume the existence of a special predicate $exit$, which is true in any exit state (e.g., in the final vertex of a data flow graph). It is also suggested that the initial state can be used as an exit state, such that test cases can be seamlessly executed. To make use of this predicate, trap properties have to include a reference to this predicate, which can for example be done with an implication:

$$\ldots \rightarrow \Box \neg exit$$

When using a requirement property based approach it depends on the requirement properties, whether test cases are fully observable or not: If the properties include internal variables, then there is a chance that observability is not always achieved.

7.7 Nondeterminism

Although a model checker can verify nondeterministic models, trace counterexamples represent only one possible choice for each nondeterministic branch. Consequently, counterexamples can only serve as test cases when using deterministic models. If a trace generated from a nondeterministic model is executed as a test case on an implementation, the test case might falsely detect a fault if the implementation makes different choices at the nondeterministic branching points. The correct verdict in this case would be *inconclusive*, as neither pass or fail can be concluded.

A simple solution that is applicable as long as there is not too much nondeterminism is presented in [112]. Here, the model is extended with an indicator variable that shows, whether a nondeterministic transition was chosen or not. When interpreting counterexamples as test cases, the execution framework has to check whether this flag is true when the implementation does not behave as expected. If the flag is set, then an inconclusive verdict is given, or else a fault is detected. It is also straight forward to extend test cases to a tree like structure, where there are different branches for different nondeterministic choices. In [112], this is done in a lazy fashion; that is, whenever an inconclusive verdict occurs during test case execution, the last known deterministic state is used as the initial state of the system, and a new counterexample is derived. This new counterexample serves as a new branch in the old test case. The applicability of such an approach depends on the amount of nondeterminism. Furthermore, if the implementation is nondeterministic itself, then applicability decreases. This means that nondeterminism as a means of underspecification or implementation choice can be handled to a certain degree, but not asynchronous, distributed systems.

Boroday et al. [35] distinguish between *weak* and *strong* test cases. A test case t for model S and mutant M is weak if M can produce an output sequence in response to t that S cannot produce. A test case t is strong if every output sequence of M in response to t differs from the corresponding sequence of S. Under fairness assumptions, a weak test case can reveal any fault if repeatedly executed; the repeated execution requires a reliable reset transitions. The method presented in [112] could be used to distinguish weak test cases from strong test cases: a (linear) test case is weak if it contains an inconclusive verdict.

Boroday et al. [35] describe methods to derive test cases for nondeterministic specifications, based on the state machine duplication approach (see Section 6.4). For the simpler case when the specification is deterministic and only the mutant is nondeterministic, weak test cases can be derived by the generic state machine duplication approach. For strong test cases, Boroday et al. [35] describe a method to derive an observer from a mutant specification.

An observer $Obs(M)$ for module M uses all outputs of M as inputs. A hidden variable $found$ is added, and the hidden variables in M are removed. Determinization is possibly performed by powerset construction. Except in trivial cases the observer is not input-enabled; additional sink states are added to

make the module input complete. In these sink states, the variable $found$ is set to true. A strong test case for a nondeterministic mutant is therefore derived if the following property does not hold:

$$S||Obs(M) \models \Box \neg\, found$$

If not only the mutant, but also the specification is nondeterministic, then weak test cases can be generated by creating an observer from the specification. A weak test case for a nondeterministic specification is therefore derived if the following property does not hold:

$$Obs(S)||M \models \Box \neg\, found$$

Because of its complexity, Boroday et al. [35] do not consider generation of strong test cases, but describe a method to detect strong test cases.

7.8 Shortcomings of Linear Trace Counterexamples

The approach of converting test requirements to temporal logic properties and then using resulting counterexamples as test cases only works as long as test requirements can be fulfilled by such linear traces. MC/DC, for example, requires pairs of test cases to cover conditions. As shown in [30], such requirements can be expressed with CTL. However, current model checkers do not support full CTL but only a linear subset such as such as $ACTL^{det}$ or LIN [28]. To overcome this problem, it is proposed in [30] that model checkers do not generate linear traces but a tree like structure called *evidence graphs*. Evidence graphs can be nonlinear, and illustrate for any CTL formula, why it is satisfied or violated. If an evidence graph is nonlinear, then it is necessary to create several test cases to cover the graph. This is simply achieved by creating one test case for each path from the root node to a leaf node of the evidence graph. The approach described in [30] is not implemented. Clarke et al. [29] proposed an algorithm to create tree-like counterexamples, which would serve a similar purpose; at the time of this writing, however, there is no available implementation.

8 TEST CASE ANALYSIS WITH MODEL CHECKERS

Model checkers are not only useful when it comes to creating test cases. Given an extant set of test cases, a model checker can be used to evaluate the quality, for example with regard to satisfaction of a given coverage criterion. A nice aspect of this approach is that coverage can be measured without actually executing test cases. Different activities during the development process can result in test cases; for example, use cases created during the requirements phase, manually created test cases, or test cases created with any automated method. It is an important task from a practical perspective to evaluate how good these test cases are.

8.1 Symbolic Test Case Execution

Analysis of test cases with a model checker is based on the idea of representing test cases as verifiable models, based on an approach by Ammann and Black [76]. Test cases are represented as constrained finite state machines (CFSM), which have an explicit state counter on which the values of all other variables depend. Test cases are converted to CFSM models with an additional variable, e.g., `State`. It is initialized with 0 and increased until the final state of the test case is reached. The values of all variables are set according only its value.

```
-- specification  AG (brake -> AX velocity = stop)
-- is false as demonstrated by the following
-- execution sequence
-> State: 1.1 <-
  accelerate = 0
  brake = 0
  velocity = stop
-> State: 1.2 <-
  accelerate = 1
  brake = 1
  velocity = slow
-> State: 1.3 <-
  accelerate = 0
  brake = 0
```

Figure 14: Counterexample created by NuSMV showing that brakes do not work in mutant model (edited for brevity).

As an example, consider test case derival by checking a mutant model of the CC example given in Section 3 (`accelerate | !brake & velocity = stop: slow`) against the first requirement property specified in Equation 13 in Section 3. NuSMV returns the counterexample shown in Figure 14, which can be used as a negative test case.

In the trace in Figure 14, at every state only those variables that changed their values are listed. In state 1.2 both `accelerate` and `brake` are activated, while due to the mutation at the same time `velocity` changes to `slow`. In state 1.3 `velocity` is still `slow`, which is a violation of the requirement that it should be `stop`. When converted to an SMV model, this trace results in the model listed in Figure 15.

This SMV model is suitable for analysis with a model checker, for example to measure coverage or a mutation score. The latter can only be directly measured in the case of weak mutation, which means that the change caused by the mutation does not have to propagate to an output to be considered as killed. In order to simulate the execution of a test case on a model, further processing is necessary. The test case is combined with the model by moving the model's main module to a sub-module of the test case, and changing all input variables to parameters of that module. This new sub-module is instantiated in the test case model, and the input variables are used as parameters, thus ensuring that the mutant model uses the inputs provided by the test case. The result is shown in Figure 16.

Finally, for each output variable a property is added that requires the output variables of the mutant model and of the test case to be equal for the duration of the test case (or alternatively, a conjunction of all these properties). After the last state of the sequence the test case does not specify how the values change. In the test case model, this is modeled by not changing the variables. However, the mutant model might still change as time progresses. Therefore, the assertion is extended to only be valid while the last step of the test case has not been exceeded.

$$\square\,(\texttt{State} < \texttt{MAX\_STATE} \rightarrow \texttt{velocity} = \texttt{model.velocity})$$

Calling the model checker on the combined model and these properties, any counterexample illustrates

```
MODULE main
VAR
  accelerate: boolean;
  brake: boolean;
  velocity: {stop, slow,
     fast};
  State: 0..2;
ASSIGN
  init(accelerate):=0;
  next(accelerate):= case
    State = 0: 1;
    State = 1: 0;
    1: accelerate;
  esac;
  init(brake) := 0;
next(brake) := case
  State = 0: 1;
  State = 1: 0;
  1: brake;
esac;
init(velocity):= stop;
next(velocity):=case
  State = 0: slow;
  State = 1: slow;
  1: velocity;
esac;
init(State) := 0;
next(State) := case
  State<2: State+1;
  1: State; esac;
esac;
```

Figure 15: Test case as verifiable SMV model.

```
MODULE Model(accelerate, brake)
VAR
  velocity: {stop, slow, fast};
ASSIGN
... As in (mutant) model

MODULE main
VAR
  model: Model(accelerate, brake);
... As in testcase model
```

Figure 16: Test case model combined with original model to simulate test case execution.

that the test case fails on the model. If the model checker does not return a counterexample, then the test case passes.

8.2 Coverage Analysis

Coverage analysis measures how thoroughly a given test suite exercises a system under test. A coverage criterion describes the items that should be executed (*covered*) by at least one test case; for example, lines of code, or branches in the control flow. Coverage criteria can also be based on specifications or models. Different coverage criteria were presented in Section 4 and Section 5. Each item described by the coverage criterion is represented as a single trap property, as described in the previous section. These trap properties can not only be used for test case generation, but also for determining coverage values. The test coverage is the percentage of items that are actually covered, i.e., reached during test case execution.

Definition 25 (Test Coverage) *The coverage C of a test suite TS with regard to a coverage criterion represented by a set of trap properties $\mathcal{P}$ is defined as the ratio of covered properties to the number of properties in total:*

$$C = \frac{1}{|\mathcal{P}|} \cdot |\{x | x \in \mathcal{P} \wedge covered(x, TS)\}|$$

The predicate $covered(a, TS)$ is true if there exists a test case $t \in TS$ such that t covers a, i.e., $t \not\models a$.

When checking a test case model (e.g., Figure 15) against a trap property, the model checker returns a counterexample if the test case covers the item represented by the trap property. Care has to be taken because a test case is only a finite prefix of an execution path. The path might be truncated such that a trap property is violated *because* of the truncation. For example, consider evaluation of a condition with the $\bigcirc$ operator on the final state of a test case. As there is no defined next state after the final state (or alternatively, the next state after the final state might implemented as the final state itself again), this state might cause a property violation.

A common, practical solution to this problem is to rewrite properties such that they can not be violated because of a finite truncation. A special variable that evaluates to true only if the current state of a sequence is any state prior to the final state is assumed. In [55], this variable is denoted as `Sustain`. This variable can be used to rewrite temporal logic formulas such that they only evaluate to false if violated before the final state, else to true. The same rewriting rules as given in Section 7.1 can be used, only the variable s is replaced with `Sustain`.

The test coverage of a given test suite is determined as follows:

1. Each test case is converted to a verifiable model, as described above.
2. Each test case model is checked against the rewritten versions of all remaining trap properties.
3. Each trap property that results in a counterexample is covered, and does not need to be checked again.

The overall test coverage is calculated from the number of covered trap properties according to Definition 25.

8.3 Mutation Analysis

As introduced in Chapter 6, another common analysis technique besides coverage analysis is mutation analysis. Here, a given test suite is examined with regard to a given set of mutants, in order to determine how many of the mutants can be distinguished from the original by the test cases. Usually, mutation analysis is applied to the source code, but specification mutation is receiving increasing attention — for example, consider the mutation based test case generation presented earlier.

Definition 26 (Mutation Score) *[76] The mutation score S for a given method $\mathcal{M}$ to create mutants, a test set t for any specification r equals the number of mutants killed by the test set, k, divided by the total number of mutants, N, produced by $\mathcal{M}$ on r:*

$$S(\mathcal{M}, r, t) = \frac{k}{N}$$

A special case of mutation analysis is presented by Ammann and Black [76]. Here, not the model but properties that represent the transition relation are mutated. As described above, these properties can be used like trap properties for test case generation. Similarly, these properties can also be used for analysis of test cases like trap properties. The mutation score is calculated from the number of mutant properties that result in a counterexample when checked against a test case model. This kind of mutation analysis uses weak mutation, which means that a mutant is killed if an erroneous state results immediately after the mutated transition.

In contrast, in strong mutation a mutant is killed if the final output is different from the original version. Strong mutation analysis can be performed by considering model mutants. Model mutants need different treatment in order to determine a mutation score. Again, each test case is converted to a verifiable model. Then, each mutant is successively combined with a test case model as described above, until the verification of such a mutant/test case model combination results in a counterexample. A counterexample indicates that the test case failed, which in turn means that the mutant is *killed*.

9 FURTHER USES OF MODEL CHECKERS IN SOFTWARE TESTING

9.1 Testing with Software Model Checkers

All techniques presented so far assume the existence of a formal model of the system under test that can be used to generate test cases. In practice, the creation of a sufficient model is one of the most difficult steps in model based testing. Sometimes the development process is supported by tools or specification languages which can serve as a basis for creating a verifiable model. Conversion between different formalisms is usually automatable; for example, Black [113] consider the generation of models from high level specifications. Often, however, a model has to be generated manually, which is difficult and error prone. Therefore there is interest in applying model checking to source code directly, removing the need for a model. This is commonly referred to as *software model checking*.

There are two different paths that have been taken to apply model checking to verification: Several tools create models in the input languages of popular model checkers from the source code. Other tools implement their own model checking procedures. For example, Bandera [114] creates SMV or Promela models from Java code. The first version of Java PathFinder [115] also converted Java programs to Promela models. Further tools that are built on top of existing model checkers are JCAT [116], Park et al. [117]

convert Java code to SAL models; Bogor [118] tries to provide a language independent software model checking framework.

The second version of Java PathFinder [119] includes a specialized virtual machine that interprets byte-code. Verisoft [120] executes C program code in order to avoid the need to represent program states and statements. CMC [121] additionally stores information about visited states. Bounded model checking is used to verify C code in CBMC [122]. SLAM [123] converts C code to Boolean abstractions that are model checked. Blast [124] uses counterexample guided abstraction refinement to verify C code.

Testing with software model checkers has been considered by Beyer et al. [125], who use the model checker Blast to create test cases from C code. Test cases can be generated with regard to predicates (i.e., safety properties), and locations in the source code. Consequently, it is possible to derive test cases for code-based coverage criteria. Visser et al. [126] use the Java PathFinder model checker to derive test cases in a similar manner. A source translation for symbolic execution with model checkers is presented by Sarfraz Khurshid and Visser [127]. This has been implemented as an extension to Java PathFinder, and can be used to generate test cases [128].

These findings show that test case generation with software model checkers is possible in theory, but scalability is not the only issue in practice. While test case generation with operational specifications creates test sequences that include the expected output, test cases created directly from the source code do not solve the oracle problem. Therefore, this is an area where further research will be needed.

9.2 Testing Timed Automata

Timed automata are automata that include special variables called clocks, which include information about time, and can be used in guard conditions, etc. Uppaal [129] is a popular model checker based on timed automata. Hessel et al. [130] proposed test case generation using Uppaal. In this approach, a special timed variant of CTL is used to formalize test purposes or coverage criteria. The generation of test cases with either test purposes or properties created for coverage criteria as described in Section 4 is proposed. This method is of particular interest for timed systems, because Uppaal supports generation of not only shortest but also quickest traces.

9.3 Combinatorial Testing

A new application of model checkers for test case generation was proposed by Kuhn and Okun [131]. Combinatorial testing tries to provide a high level of coverage of a system's input domain with a small number of test cases. The number of possible input combinations is usually extremely high; for example, a system with 20 inputs with 10 values each allows a total of 10^{20} different combinations. If, however, only a limited number of combinations is selected, then this number is reduced significantly. Considering all possible pairs of inputs for the above example results in 190 different input pairs with 100 different possible input combinations for each pair resulting in 19,000 different test cases which is substantially smaller than the overall number of different combinations.

The underlying idea of combinatorial testing can be best explained using a small example. Consider a system with 3 boolean input variables v_1, v_2, and v_3. All 2-way combinations would be v_1 v_2, v_1 v_3, and v_2 v_3. Only for these variable combinations all possible input value combinations have to be tested leading to 12 test case instead of 16. In general there are $\begin{pmatrix} n \\ k \end{pmatrix}$ different combinations when we have n input variables and we want to compute all k-way combinations. For each of this combinations all possible input

value tuple are generated. Variables which are not in a k-way combination are assigned to a value which can be a random value or a value which allows to execute the program under test.

In practice, 3 to 6-way combinations are also used in addition to pairs and provide good results. The empirical results in [131] showed a fault detection rate of 100 percent for a 5-way combination. The underlying assumption of combinatorial testing is that only smaller subsets of input variables are responsible for certain outputs. Hence, only those inputs must be considered when testing a specific functionality.

In [131], a model checker is used to derive test cases for *t-way* coverage. Given assertions of the form $\mathbf{AG}\ (P \rightarrow \mathbf{AX}\ R)$ and t-way variable combinations, $v_1 \wedge v_2 \wedge ... \wedge v_t$ where each v_i is a condition comprising a variable and a assigned value, three different types of trap properties are proposed:

$$\mathbf{AG}\ (v_1 \wedge v_2 \wedge ... \wedge v_t \wedge P \rightarrow \mathbf{AX}\ \neg R)$$
$$\mathbf{AG}\ (v_1 \wedge v_2 \wedge ... \wedge v_t \rightarrow \mathbf{AX}\ \neg 1)$$
$$\mathbf{AG}\ (v_1 \wedge v_2 \wedge ... \wedge v_t \rightarrow \mathbf{AX}\ \neg R)$$

The first property might be trivially true if t is large because P together with $v_1, \ldots, v_t$ computes to false which makes the implication true. Because of this reason [131] proposes using the second property which simply forces a single step to be taken ($\neg 1$ is always false). Alternatively, the final property removes the condition P to avoid trivially true cases.

9.4 Testing Composite Webservices with Model Checkers

Web services are a recently popular mechanism to allow interaction of heterogeneous systems via the internet. A particular strength of such techniques is that different services can be composed to form new, more complex services. There are several different languages that can be used to describe web services and aid the automatic composition.

Composed web services result in complex behaviors, where the components can be distributed across networks and implemented with different tools and systems. Therefore, verification of composed web service models as well as testing of composed web service implementations is very important. The use of model checkers to verify web service composition has been proposed by several researchers. A combined approach of verification and testing based on model checkers has been proposed by Huan et al. [132]. In this approach, OWL-S (Web Ontology Language for Web Services) specifications are translated to a C-like language, which is verified with the model checker Blast. The model checker is also used to create witnesses that can be used as test cases, following the approach presented by Beyer et al. [125].

Garcia-Fanjul et al. [133] translate web service compositions specified with BEPL into Promela, the language of the model checker SPIN. Then, trap properties are used to create transition coverage test suites.

9.5 Adaptive Model Checking

Adaptive model checking [134] is an advanced combination of model checking and testing. Verification is performed on an incomplete model. If a counterexample is found, then the counterexample is executed as a test case on an actual implementation. If the system passes the test case, then a property violation has been found. If the test case does not pass, then the model is refined according to the actual execution result. This is also related to black-box checking [135], where no model at all to start with is assumed.

9.6 On-the-fly Testing with Model Checkers

All approaches to test case generation presented so far in this survey create test cases *offline*; that is, the test cases are first generated from a model, and only once this generation process is done are they executed. An alternative approach is to interleave test case generation and execution; this is known as *online* or *on-the-fly* testing. On-the-fly testing has several advantages to offline testing; it can be continued for a very long time, reduces the state explosion problem because only a limited part of the state space needs to be considered at a time, and nondeterminism is handled naturally.

Examples of on-the-fly testing tools based on model checkers are T-Uppaal [136], based on Uppaal, and the work presented by de Vries and Tretmans [137], who use the model checker SPIN. These tools are not based on model checking algorithms, but rather use the modeling and simulation features of the underlying model checkers.

10 TOOLS

Although testing with model checkers has been considered by several research groups, much of the work was done on research prototypes that were never released to the public. This section considers the test case generation tools that are publicly available.

There is an online demonstration tool [138] for mutation based test case generation with model checkers based on the work by Ammann et al. [77]. While it is only possible to generate test cases for the cruise control example application used in [77], the tool helps to illustrate the steps involved in the process.

Since version 3.0, SAL [23] includes the tool SAL-ATG [139], which allows test case generation with SAL. As SAL provides a Scheme-based environment, this tool offers many possibilities for customization and extension. SAL-ATG does not use trap properties, but requires that the model is extended with *trap variables*, which are true only when a test goal is reached. This basically allows similar coverage goals as with regular trap properties, although it is slightly more complicated to rewrite the model than to simply provide properties. In general, most coverage criteria that can be expressed as trap properties can also be encoded in the model. Figure 17 shows the car controller example model from Figure 4 as a SAL model. The variables `t0`–`t5` are not actually part of the specification, but are trap variables for simple transition coverage. SAL-ATG can use these trap variables to create a simple transition coverage test suite. For this, the list of goals has to be specified as listed in Figure 18. Assuming this list of goals is saved in a file called `car_control_goals.scm` and the model is saved in a file called `car_control.sal`, SAL-ATG is started with the following command:

```
sal-atg car_control main car_control_goals.scm
```

SAL-ATG will try to find test cases such that every trap variable is true at some point. There are several options to the test case generation; for details see [139].

ATGT (ASM Tests Generation Tool) [140] is a Java-based tool that implements the concepts presented in [64, 65, 83] to automatically create test cases for ASM specifications. It offers a graphical user interface and uses the model checker SPIN [22]. The tool automatically creates trap properties, and illustrates them graphically. As an example to get started, the ASM model listed in Figure 9 can be used with ATGT, and a version of a popular safety injection system model is available on the tool's website [140].

```
car_control: CONTEXT =
BEGIN
  speed: TYPE = {stop, slow, fast};

  main: MODULE =
  BEGIN
    INPUT
      accelerate, brake : BOOLEAN
    OUTPUT
      velocity: speed
    LOCAL
      t0, t1, t2, t3, t4, t5: BOOLEAN

    INITIALIZATION
      velocity = stop;
      t0 = FALSE; t1 = FALSE; t2 = FALSE;
      t3 = FALSE; t4 = FALSE; t5 = FALSE;

    TRANSITION
    [
      accelerate = TRUE AND brake = FALSE AND velocity = stop -->
        velocity' = slow; t0' = TRUE;
    []
      accelerate = TRUE AND brake = FALSE AND velocity = slow -->
        velocity' = fast; t1' = TRUE;
    []
      accelerate = FALSE AND brake = FALSE AND velocity = fast -->
        velocity' = slow; t2' = TRUE;
    []
      accelerate = FALSE AND brake = FALSE AND velocity = slow -->
        velocity' = stop; t3' = TRUE;
    []
      brake = TRUE -->
        velocity' = stop; t4' = TRUE;
    []
      ELSE -->
        t5' = TRUE;
    ]
  END;
END
```

Figure 17: Simple car controller as SAL specification with trap variables for simple transition coverage.

```
(define goal-list '(
 "t0" "t1" "t2" "t3" "t4" "t5"
))
```

Figure 18: Test goal list for SAL-ATG.

11 OUTSTANDING RESEARCH ISSUES

Many researchers have considered testing with model checkers over the last couple of years, and significant progress has been made to turn model checker based testing into a technique suitable for real world application. Many issues remain, however. Section 7 gave an overview of issues that have been considered so far. Not surprising, the main problem is performance. Research on model checkers is progressing, and the size of models that can be handled constantly increases. There is a need to adapt model checking techniques to faster counterexample creation. Directed model checking [17] is an example of such a technique.

At the same time it is not sufficient to blame the performance of model checkers. Even if model checkers could handle models of deliberate size, many of the currently examined testing techniques would result in unfeasibly large test suites. Therefore, research on model abstraction is essential, both for performance and for scalability reasons. Abstraction is an active research topic, but it is seldom considered with a software testing background. Abstraction techniques suitable for verification purposes might not be suitable for testing. This leaves many unanswered questions, for example, how do test models differ from verification models, and what abstraction techniques are suitable for testing?

There is a lack of documented empirical experience with testing with model checkers. Most work evolves around a set of small, well known example applications. The only available case study that evaluated the scalability [53] showed promising results, but later studies [107] showed that the considered example application has some peculiarities that make it questionable, whether the results are really representative. Further experience reports would not only answer questions about scalability, but could also be used to compare the many available techniques with regard to their relative power. Such a comparison would be invaluable for someone wanting to use model checkers for testing.

Even if all performance problems were resolved, there is still one intrinsic problem to all model based testing approaches: Where does the model come from? In most work on model based testing, the existence of a suitable formal model is assumed. The model creation, however, is one of the most difficult parts of the whole development process. Creating models manually is a complicated task, and two specifiers writing a model for an application will probably come up with different models. Different models, however, will most likely result in different test suites. Some approaches try to avoid the use of a model altogether, for example, black-box checking techniques (see Section 9.5). Alternative approaches try to extract models from source code, and sometimes model based development tools are used, which means that a verifiable model naturally results from the development process. Such approaches introduce new problems, for example, what exactly is tested by test cases resulting from the model: the implementation, or just the tools that created the model from the source code or vice versa?

ACKNOWLEDGEMENTS

Thanks to Paul Black for providing useful suggestions, and to Sanjai Rayadurgam for helpful explanations.

REFERENCES

[1] John Callahan, Francis Schneider, and Steve Easterbrook. Automated Software Testing Using Model-Checking. In *Proceedings 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.

[2] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Ed Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, Enschede, the Netherlands, April 1997. Springer-Verlag.

[3] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical Report No. 04/2006, Department of Computer Science, The University of Waikato (New Zealand), April 2006.

[4] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 3540262784.

[5] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A New Symbolic Model Verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag. ISBN 3-540-66202-2.

[6] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA*, pages 46–57. IEEE, 1977.

[7] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag. ISBN 3-540-11212-X.

[8] E. Allen Emerson and Joseph Y. Halpern. 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*, pages 169–180, New York, NY, USA, 1982. ACM Press. ISBN 0-89791-070-2. doi: 10.1145/800070.802190.

[9] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985. ISSN 0004-5411. doi: 10.1145/2455.2460.

[10] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[11] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA., 1 edition, 2001. 3rd printing.

[12] Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32st Conference on Design Automation (DAC)*, pages 427–432. ACM Press, 1995.

[13] Orna Lichtenstein and Amir Pnueli. 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*, pages 97–107, New York, NY, USA, 1985. ACM Press. ISBN 0-89791-147-4. doi: 10.1145/318593.318622.

[14] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Computer Society, June 1986.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla. 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*, pages 117–126, New York, NY, USA, 1983. ACM Press. ISBN 0-89791-090-7. doi: 10.1145/567067.567080.

[16] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag. ISBN 3-540-11494-7.

[17] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 57–79, New York, NY, USA, 2001. Springer-Verlag New York, Inc. ISBN 3-540-42124-6.

[18] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN 0792393805.

[19] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986. ISSN 0018-9340.

[20] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992. ISSN 0360-0300. doi: 10.1145/136035.136043.

[21] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag. ISBN 3-540-65703-7.

[22] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997. ISSN 0098-5589. doi: 10.1109/32.588521.

[23] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

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

[25] David L. Dill. The murphi verification system. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393, London, UK, 1996. Springer-Verlag. ISBN 3-540-61474-5.

[26] Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 User Manual, Oct 1997.

[27] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Proceedings of the Eighth Conference on Computer Aided Verification (CAV 1996)*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 1996.

[28] Edmund Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 208–224. Springer-Verlag, 2004.

[29] Edmund M. Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 19–29, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9.

[30] Duminda Wijesekera, Lingya Sun, Paul Ammann, and Gordon Fraser. Relating counterexamples to test cases in CTL model checking specifications. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 75–84, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-850-3. doi: 10.1145/1291535.1291543.

[31] Robert Meolic, Alessandro Fantechi, and Stefania Gnesi. Witness and counterexample automata for ACTL. In *Formal Techniques for Networked and Distributed Systems*, volume 3235 of *Lecture Notes in Computer Science*, pages 259–275, 2004.

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

[33] Hyoung Seok Hong and Insup Lee. Automatic Test Generation from Specifications for Control-Flow and Data-Flow Coverage Criteria. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2003.

[34] Li Tan, Oleg Sokolsky, and Insup 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.

[35] Sergiy Boroday, Alexandre Petrenko, and Roland Groz. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science*, 190:3–19, 2007.

[36] Orna Kupferman and Moshe Y. Vardi. Model checking revisited. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 36–47, London, UK, 1997. Springer-Verlag. ISBN 3-540-63166-6.

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

[38] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages 12 Years Later. In *Proceedings of the IEEE*, volume 91, pages 64–83, 2003.

[39] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79:1305–1320, 1991.

[40] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[41] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[42] Jan Tretmans. Testing Concurrent Systems: A Formal Approach. In J.C.M Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999.

[43] C. L. Heitmeyer. *Encyclopedia of Software Engineering*, volume 2, chapter Software Cost Reduction. John Wiley & Sons, 2002.

[44] Mats P. E. Heimdahl, Sanjai Rayadurgam, and Willem Visser. Specification Centered Testing. In *Proceedings of the Second International Workshop on Automates Program Analysis, Testing and Verification (ICSE 2000)*, 2000.

[45] S. Rayadurgam and M. P. E. Heimdahl. Test-sequence generation from formal requirement models. In *HASE '01: The 6th IEEE International Symposium on High-Assurance Systems Engineering*, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0769512755.

[46] Sanjai Rayadurgam and Mats P. E. Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. Technical Report 01-005, University of Minnesota, Minneapolis, January 2001.

[47] Jeffrey M. Thompson, Mats P. E. Heimdahl, and Steven P. Miller. Specification-based prototyping for embedded systems. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 163–179, London, UK, 1999. Springer-Verlag. ISBN 3-540-66538-2. doi: 10.1145/318773.318940.

[48] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng.*, 20(9):684–707, 1994. ISSN 0098-5589. doi: 10.1109/32.317428.

[49] Paul Ammann, Jeff Offutt, and Hong Huang. Coverage criteria for logical expressions. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 99, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2007-3.

[50] A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *ICECCS*. IEEE Computer Society, 1999. ISBN 0-7695-0434-5.

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

[52] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. Citation will be complete by the time the review cycle is over.

[53] Mats P.E. Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-Generating Test Sequences using Model Checkers: A Case Study. In *Third International International Workshop on Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 42–59. Springer Verlag, October 2003.

[54] Sanjai Rayadurgam and Mats P.E. 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.

[55] Paul Ammann, Paul E. Black, and Wei Ding. Model Checkers in Software Testing. Technical Report NIST-IR 6777, National Institute of Standards and Technology, 2002.

[56] Jr. Sheldon B. Akers. On a theory of boolean functions. *Journal of the Society for Industrial and Applied Mathematics*, 7(4):487–498, 1959. doi: 10.1137/0107041.

[57] Sergiy A. Vilkomir and Jonathan P. Bowen. Reinforced Condition/Decision Coverage (RC/DC): A New Criterion for Software Testing. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 291–308, London, UK, 2002. Springer-Verlag. ISBN 3-540-43166-7.

[58] Hyoung S. Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *Tools and Algorithms for the Construction and Analysis of Systems : 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002. Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–161. Springer Verlag Gmbh, 2002.

[59] Hyoung S. Hong, Sung D. Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data flow testing as model checking. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 232–242, Washington, DC, USA, 2003. IEEE Computer Society.

[60] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985. ISSN 0098-5589. doi: 10.1109/TSE.1985.232226.

[61] Hyoung S. Hong and Hasan Ural. Dependence testing: Extending data flow testing with control dependence. In *Testing of Communicating Systems*, volume 3502 of *Lecture Notes in Computer Science*, pages 23–39. Springer Verlag Gmbh, 2005.

[62] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Sung Deok Cha. Automatic Test Generation from Statecharts Using Model Checking. Technical report, MS-CIS-01-07, 2001.

[63] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000. ISSN 1529-3785. doi: 10.1145/343369.343384.

[64] A. Gargantini and E. Riccobene. Asm-based testing: Coverage criteria and automatic test sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, 2001.

[65] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003. Proceedings*, volume 2589 of *Lecture Notes in Computer Science*, pages 263+. Springer Verlag Gmbh, 2003.

[66] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven 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. ISBN 1-59593-263-1.

[67] John R. Callahan, Stephen M. Easterbrook, and Todd L. Montgomery. Generating Test Oracles Via Model Checking. Technical report, NASA/WVU Software Research Lab, 1998.

[68] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in actl formulaas. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 279–290, London, UK, 1997. Springer-Verlag. ISBN 3-540-63166-6.

[69] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. In *CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 82–96, London, UK, 1999. Springer-Verlag. ISBN 3-540-66559-5.

[70] Mitra Purandare and Fabio Somenzi. Vacuum cleaning CTL formulae. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 485–499, London, UK, 2002. Springer-Verlag. ISBN 3-540-43997-8.

[71] Paul Ammann, Wei Ding, and Daling Xu. Using a Model Checker to Test Safety Properties. In *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems (ICECCS 2001)*, pages 212–221, Skovde, Sweden, 2001. IEEE.

[72] Gordon Fraser and Franz Wotawa. Property relevant software testing with model-checkers. *SIGSOFT Software Engineering Notes*, 31(6):1–10, 2006. ISSN 0163-5948. doi: 10.1145/1218776.1218787.

[73] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11:34–41, 1978.

[74] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report, School of Information and Computer Science, Georgia Inst. of Technology, Atlanta, Ga., Sept. 1979.

[75] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Comput. Lang.*, 10 (1):63–73, 1985. ISSN 0096-0551. doi: 10.1016/0096-0551(85)90011-6.

[76] Paul Ammann and Paul E. Black. A Specification-Based Coverage Metric to Evaluate Test Sets. In *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 239–248, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0418-3.

[77] Paul E. Ammann, Paul E. Black, and William 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.

[78] T. Srivatanakul, J. A. Clark, S. Stepney, and F. Polack. Challenging formal specifications by mutation: a CSP security example. In *Tenth Asia-Pacific Software Engineering Conference*, pages 340–350, 2003.

[79] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation Operators for Specifications. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, Washington, DC, USA, 2000. IEEE Computer Society.

[80] Gordon Fraser and Franz Wotawa. Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*, pages 16–22, Los Alamitos, CA, USA, 2006. IEEE Computer Society. ISBN 0-7695-2703-5. doi: 10.1109/ICSEA.2006.75.

[81] Gordon Fraser and Franz Wotawa. Using and improving requirement properties for mutation based test-case generation. *22. WI-MAW Rundbrief*, Jahrgang 12(2):5–23, 2006.

[82] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation of Model Checker Specifications for Test Generation and Evaluation. *Mutation testing for the new century*, pages 14–20, 2001.

[83] Angelo Gargantini. Using Model Checking to Generate Fault Detecting Tests. In *Proceedings of the International Conference on Tests And Proofs (TAP)*, Zurich, Switzerland, 2007.

[84] Vadim Okun, Paul E. Black, and Yaacov Yesha. Testing with Model Checker: Insuring Fault Visibility. Technical Report NIST-IR 6929, National Institute of Standards and Technology, 2003.

[85] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, London, UK, 2000. Springer-Verlag. ISBN 3-540-67770-4.

[86] Paul Ammann and Paul E. Black. Abstracting Formal Specifications to Generate Software Tests via Model Checking. In *Proceedings of the 18th Digital Avionics Systems Conference*, volume 2. IEEE, 1999.

[87] Hyoung S. Hong and Hasan Ural. Using model checking for reducing the cost of test generation. In *Formal Approaches to Software Testing*, volume 3395 of *Lecture Notes in Computer Science*, pages 110–124. Springer Verlag Gmbh, 2005.

[88] Gordon Fraser and Franz Wotawa. 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*, pages 64–74, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-850-3. doi: 10.1145/1291535.1291542.

[89] Hongwei Zeng, Huaikou Miao, and Jing Liu. Specification-based test generation and optimization using model checking. *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE'07)*, 0:349–355, 2007. doi: 10.1109/TASE.2007.46.

[90] P. E. Black and S. Ranville. Winnowing tests: Getting quality coverage from a model checker without quantity. In *Digital Avionics Systems, 2001. DASC. The 20th Conference*, volume 2, pages 9B6/1–9B6/4 vol.2, 2001.

[91] Gordon Fraser and Franz Wotawa. Redundancy based test-suite reduction. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2007.

[92] Klaus Havelund and Grigore Rosu. Monitoring programs using rewriting. In *ASE '01: Proceedings of the 16th IEEE Int. Conference on Automated Software Engineering*, page 135, Washington, DC, USA, 2001. IEEE Computer Society.

[93] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program Monitoring with LTL in Eagle. In *PADTAD'04, Parallel and Distributed Systems: Testing and Debugging*, 2004.

[94] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6(2):158–173, 2004. ISSN 1433-2779. doi: 10.1007/s10009-003-0117-6.

[95] Grigore Rosu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, 2005. ISSN 0928-8910. doi: 10.1007/s10515-005-6205-y.

[96] Gordon Fraser and Franz Wotawa. Mutant Minimization for Model-Checker Based Test-Case Generation. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 161–168. IEEE Computer Society, 2007. doi: 10.1109/TAICPART.2007.4344120.

[97] Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating Efficient Test Sets with a Model Checker. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 261–270, 2004.

[98] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993. ISSN 1049-331X. doi: 10.1145/152388.152391.

[99] Jeffery von Ronne Christie Hong Gregg Rothermel, Mary Jean Harrold. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[100] Hao Zhong, Lu Zhang, and Hong Mei. An experimental comparison of four test suite reduction techniques. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 636–640. ACM Press, 2006. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134380.

[101] Mats Per Erik Heimdahl and George Devaraj. Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In *ASE*, pages 176–185. IEEE Computer Society, 2004. ISBN 0-7695-2131-2.

[102] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, 29(3):195–209, 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1183927.

[103] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. 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 Software Maintenance*, page 34. IEEE Computer Society, 1998. ISBN 0-8186-8779-7.

[104] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. In *ICSE '95: Proceedings of the 17th Int. Conference on Software Engineering*, pages 41–50. ACM Press, 1995. ISBN 0-89791-708-1. doi: 10.1145/225014.225018.

[105] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 179, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0016-1.

[106] Gordon Fraser and Franz Wotawa. Test-case prioritization with model-checkers. In *Proceedings of the IASTED International Conference on Software Engineering (SE'07)*, 2007.

[107] Mats Per Erik Heimdahl, George Devaraj, and Robert Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In *HASE*, pages 178–186. IEEE Computer Society, 2004. ISBN 0-7695-2094-4.

[108] George Devaraj, Mats P. E. Heimdahl, and Donglin Liang. Coverage-directed test generation with model checkers: Challenges and opportunities. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, pages 455–462, Washington, DC, USA, 2005. IEEE Computer Society. doi: 10.1109/COMPSAC.2005.66.

[109] Aynur Abdurazik, Paul Ammann, Wei Ding, and Jeff Offutt. Evaluation of three specification-based coverage testing criteria. In *Proceedings of the 6th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2000)*, pages 179–187, Tokyo, Japan, September 2000. IEEE Computer Society.

[110] Lihua Xu, Marcio Dias, and Debra Richardson. Generating regression tests via model checking. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 336–341, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2209-2-1.

[111] Gordon Fraser, Bernhard Aichernig, and Franz Wotawa. Handling model changes: Regression testing and test-suite update with model-checkers. *Electronic Notes in Theoretical Computer Science*, 190: 33–46, 2007.

[112] Gordon Fraser and Franz Wotawa. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2007)*, page 45, Los Alamitos, CA, USA, 2007. IEEE Computer Society. ISBN 0-7695-2937-2. doi: 10.1109/ICSEA.2007.71.

[113] Paul E. Black. Modeling and Marshaling: Making Tests From Model Checker Counterexamples. In *Proc. of the 19th Digital Avionics Systems Conference*, pages 1.B.3–1–1.B.3–6 vol.1. IEEE, 2000.

[114] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-206-9. doi: 10.1145/337180.337234.

[115] Klaus Havelund. Java pathfinder, a translator from java to promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, page 152, London, UK, 1999. Springer-Verlag. ISBN 3-540-66499-8.

[116] Claudio DeMartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent java programs. *Softw. Pract. Exper.*, 29(7):577–603, 1999. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(199906)29:7¡577::AID-SPE246¿3.0.CO;2-V.

[117] David Y. W. Park, Ulrich Stern, Jens U. Skakkebaek, and David L. Dill. Java model checking. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 253, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0710-7.

[118] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-743-5. doi: 10.1145/940071.940107.

[119] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0710-7.

[120] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-853-3. doi: 10.1145/263699.263717.

[121] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002. ISSN 0163-5980. doi: 10.1145/844128.844136.

[122] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. ISBN 3-540-21299-X.

[123] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. *SIGPLAN Not.*, 36(5):203–213, 2001. ISSN 0362-1340. doi: 10.1145/381694.378846.

[124] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *Model Checking Software: 10th International SPIN Workshop, Portland, OR, USA, May 9-10, 2003. Proceedings*, pages 235–239. Springer-Verlag, 2003.

[125] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating Tests from Counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04, Edinburgh)*, pages 326–335. IEEE Computer Society Press, 2004.

[126] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-820-2. doi: 10.1145/1007512.1007526.

[127] Corina S. Pasareanu Sarfraz Khurshid and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS '03: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, Warsaw, Poland, 2003. Springer-Verlag.

[128] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for java containers using state matching. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-263-1. doi: 10.1145/1146238.1146243.

[129] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1 - 2):134–152, December 1997.

[130] Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-Optimal Real-Time Test Case Generation Using Uppaal. In Alexandre Petrenko and Andreas Ulrich, editors, *Proceedings of the Third International Workshop on Formal Approaches to Software Testing (FATES 2003)*, volume 2931, pages 114–130, 2004.

[131] D. Richard Kuhn and Vadim 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. doi: 10.1109/SEW.2006.26.

[132] Hai Huan, Wei-Tek Tsai, Raymond Paul, and Yinong Chen. Automated model checking and testing for composite web services. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, pages 300–307. IEEE Computer Society, 2005.

[133] Jose Garcia-Fanjul, Javier Tuya, and Claudio de la Riva. Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In *Proceedings of the International Workshop on Web Services Modeling and Testing (WS-MaTe 2006)*, pages 83–94, 2006.

[134] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370, London, UK, 2002. Springer-Verlag. ISBN 3-540-43419-4.

[135] Doron Peled. Model checking and testing combined. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 47–63. Springer, 2003.

[136] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online Testing of Real-time Systems Using UPPAAL. In Jens Grabowski and Brian Nielsen, editors, *Proceedings of the 4th International Workshop on Formal Approaches to Testing of Software (FATES 2004)*, volume 3395 of *Lecture Notes in Computer Science*. Springer-Verlag GmbH, 2004.

[137] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using SPIN. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):382–393, March 2000. doi: 10.1007/s100090050044.

[138] Paul E. Black. Demonstration of Generating Tests from Formal Specifications [web page]. URL http://hissa.nist.gov/ black/AFTG/. [Accessed October 24th, 2007].

[139] Grégoire Hamon, Leonardo de Moura, and John Rushby. Automated Test Generation with SAL. Technical report, Computer Science Laboratory, SRI International, 2005.

[140] Angelo Gargantini. ATGT: ASM Tests Generation Tool [web page]. URL http://cs.unibg.it/gargantini/projects/atgt/. [Accessed October 24th, 2007].