Coverage Based Test-Case Generation using Model Checkers

Sanjai Rayadurgam and Mats P. E. Heimdahl

Department of Computer Science and Engineering, University of Minnesota
200 Union Street S.E., 4-192, Minneapolis, MN 55455, USA
E-mail: {rsanjai,heimdahl}@cs.umn.edu

Abstract

This paper presents a method for automatically generating test cases to structural coverage criteria. We show how a model checker can be used to automatically generate complete test sequences that will provide a predefined coverage of any software development artifact that can be represented as a finite state model. Our goal is to help reduce the high cost of developing test cases for safety-critical software applications that require a certain level of coverage for certification, for example, safety-critical avionics systems that need to demonstrate MC/DC (modified condition and decision) coverage of the code.

We define a formal framework suitable for modeling software artifacts, like, requirements models, software specifications, or implementations. We then show how various structural coverage criteria can be formalized and used to make a model checker provide test sequences to achieve this coverage. To illustrate our approach, we demonstrate, for the first time, how a model checker can be used to generate test sequences for MC/DC coverage of a small case example.

1. Introduction

Software development for critical embedded control systems, such as the software controlling aeronautics applications and medical devices, is a costly, time consuming, and error prone process. In such projects, the validation and verification phase (V&V) consume approximately 50%–70% of the software development resources. Thus, if the process of deriving test cases for V&V could be automated and provide requirements-based and code-based test suites that satisfy the most stringent standards (such as DO-178B–the standard governing the development of flight-critical software for civil aviation [29]), dramatic time and cost savings would be realized.

This paper presents a method for automatically generating test cases to *structural coverage criteria*. We show how a *model checker* can be used to generate complete test sequences that provide a predefined coverage of any software development artifact that can be represented as a finite state model. We provide a formal framework that is (1) suitable for defining our test-case generation approach and (2) easily used to capture finite state representations of software artifacts such as program code, software specifications, and requirements models. We show how common structural coverage criteria can be formalized in our framework and expressed as temporal logic formula used to challenge a model checker to find test cases. Finally, we demonstrate how a model checker can be used to generate test sequences for MC/DC coverage.

If a software artifact, may that be code, specification, or requirements model, can be represented as a finite state transition system, *model checking* techniques [11] can be used to generate test cases. Model checkers are tools that explore the reachable state space of a model and report if properties of interest are violated in some state. If a violation is detected, the tool will report a sequence of inputs that brought the system to the violating state. Here, we show how one may use various test coverage criteria as challenges to the model checker. For example, we can assert that the "true" branch out of a decision point cannot be taken. If this branch in fact can be taken, the model checker will generate a sequence of inputs (with associated outputs) that forces the code to take this branch—we have found a test case that exercises this branch (if we talk about branch coverage) or the condition (if we talk about basic condition coverage) [3].

The remainder of the paper is organized as follows. Section 2 provides the relevant background and the goals of our work. Section 3 defines a formal framework suitable for model checking in which we can discuss the coverage of various software engineering artifacts. Section 4 shows how common structural coverage criteria can be expressed in terms of the system model. Section 5 discusses how these criteria can be encoded as temporal logic properties for model checking and Section 6 illustrates these ideas on a small example. We discuss and compare some of the related work by others in Section 7 and then conclude the paper.

2. Background

The notion of test adequacy criteria has been extensively researched and studied [32, 15]. The criteria establish the objectives of testing in a quantifiable manner. They help answer questions like, what should be tested, how much testing is required, and how the testing goals could be achieved.

Testing criteria can be classified into different types based on two orthogonal schemes [32] – the *source of the*

information used to specify the criteria and *the underlying testing approach* used to satisfy the criteria. Thus, *specification-based* testing and *code-based* testing are examples of classification under the former scheme while under the latter are *structural* testing, *fault-based* testing and *error-based* testing. In structural testing, the criteria is specified in terms of coverage of certain constructs in the software artifact. Traditionally, this has been used for testing code. Examples include, various code coverage criteria, like statement coverage, branch coverage, condition coverage and data-flow testing. In fault-based testing, the criterion is specified in terms of some measurement of fault-detecting ability of the tests. Mutation testing is an example of this approach. Error-based testing requires tests to check the software artifact at certain error-prone points which are determined based on our knowledge about the most likely types and places for mistakes in the software artifact. Boundary testing of code is an example of this approach. The same rationale is often used in creating checklist items for inspections of software artifacts like specifications and design.

In this paper we deal with structural testing techniques where the criteria are expressed in terms of the structure of the system under test. However, rather than making the method specific to a software artifact, we define a formal model of the system and define the criteria and techniques in terms of the system model. The approach discussed here could then be applied to any software artifact that can be mapped to the formal system model. For example, one could map specifications written in languages like SCR [20, 18] and RSML^{-e} [17, 30] to this formal model. Equally, one could map implementations in languages like Java to this model using techniques that extract abstract models from program source code [12, 16]. In particular, our goal is to build a structural testing framework that could be applied equally well to both formal specifications and program code, especially of critical systems, which is our research focus. We have chosen to work mainly with *condition-based coverage criteria* since they are easily adaptable to testing of formal specifications as well as implementations. These criteria are concerned with how well a test case exercises the conditions guarding the decision points in a program (or formal specification if we are interested in specification-based testing). For example, basic condition coverage requires test cases that make the condition in each decision point take on both truth values. Other criteria, such as MC/DC (modified condition and decision coverage [9]) are more involved. MC/DC requires the test cases to demonstrate that each basic predicate making up a condition independently affects the outcome of the condition. Coverage criteria will be discussed in more detail in Section 4.

Figure 1 shows the overall view of the test generation framework. We map software artifacts to a finite state system suitable for model checking and use a model checker to find test cases by formulating the test criterion as a property to be verified. For example, to test a transition, between states A and B, guarded with condition C, we formulate a condition describing a test case testing this transition—the sequence of inputs must take the model to state A; in state A, C must be true, and the next state must be B. This is a property expressible in the logics CTL (computational tree

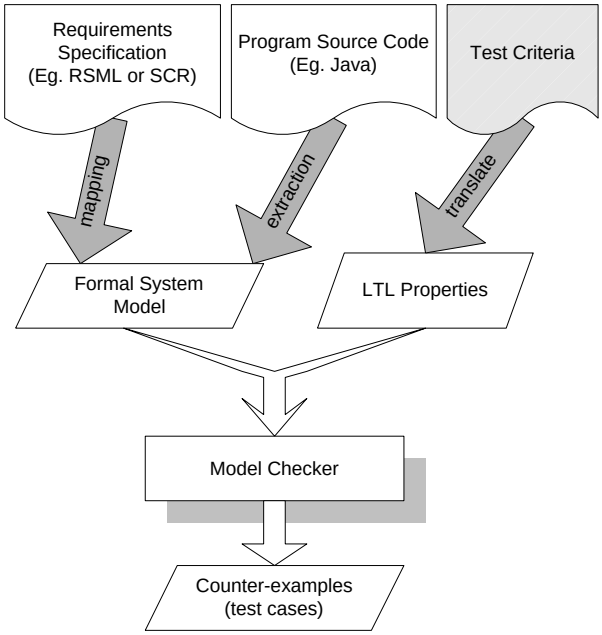


Figure 1. Test Generation Framework

logic) or LTL (linear time temporal logic) used in common model checkers. We can now challenge the model checker to find a path to such a state by negating the property (by asserting that there is no such input sequence) and start verification. The model checker will search for a counterexample demonstrating that this property is, in fact, satisfiable; such a counterexample constitutes a test case that will exercise the transition of interest. Repeating this process for each transition in the formal model, we derive test sequences that provide transition coverage of the model.

3. System Model

To provide a rigorous foundation for our work we present a formal framework that can be used to capture any software artifact that can be model checked, such as program code or formal models.

3.1. State Space

The system model has a finite control component and finite (but, typically large) or infinite data component. For our purposes we do not distinguish between control and data components of the system. We assume that the system state is uniquely determined by the value of *n* variables, $\{x_1, x_2, \ldots, x_n\}$, where each x_i takes its value from its domain D_i. Thus, the reachable state space of the system is a subset of $D = D_1 \times D_2 \times \ldots \times D_n$, where D is the domain of the state vector $\overrightarrow{x} = (x_1, \ldots, x_n)$. The system may move from one state to another subject to the constraints imposed by its transition relation, which defines the legal moves. The set of initial values for the variables, which is a subset of D, is assumed to be defined by a predicate ρ. The structure of this predicate is similar to that of the transition

relation, discussed next. Note that, in general, some of the D_i may be infinite. However, to generate test cases using a model checker, finite abstractions of such infinite domains are necessary. The abstraction methods one could use are independent of the approach described in this paper. However, if abstractions are used, the generated test sequences will have abstract data which must then be made concrete by applying inverse abstractions.

3.2. Transition Relation

The transition relation is a subset of $D \times D$, specified as a boolean predicate on the values of the variables defining the state-space. We assume that there is a transition relation for each variable, x_i, specifying how the variable may change its value as the system moves from one state to another. Informally speaking, the transition relation for x_i can be thought of as specifying three components: a set of pre-state values of x_i, a set of post-state values for x_i and the condition which *guards* when x_i may change from a pre-state value to a post-state value. We adopt the usual convention that primed versions of variables refer to their post-state values while the unprimed versions refer to their pre-state values.

Definition 1. *A **predicate** is a boolean valued function parameterized by variable references.*

The transition relation itself could be viewed as a predicate that tests for membership in an appropriate subset of $D \times D$. But, for our purposes, we need to examine the structure of the transition relation for a given system. All the components that make up the transition relations are again in turn predicates.

Definition 2. *A **clause** is a predicate that cannot be broken down into sub-predicates connected by boolean operators.*

Clauses, which are atomic units for our purposes, are the building blocks for the predicates. Predicates are made up of clauses combined using boolean operators.

Definition 3. *A **pre-state predicate** for a variable x_i is a predicate whose only parameter is the variable reference x_i. We use $\alpha_{i,j}$ to denote the j^{th} pre-state predicate of x_i.*

Definition 4. *A **post-state predicate** for a variable x_i is a predicate whose parameters are from the set $\{x_1, ..., x_n, x'_1, ..., x'_i\}$, in which every clause includes the variable reference, x'_i. We use $\beta_{i,j}$ to denote the j^{th} post-state predicate of x_i.*

Definition 5. *A **simple transition** for a variable x_i is a conjunction of a pre-state predicate for x_i, a post-state predicate for x_i, and a predicate called the **guard** whose parameters are from the set $\{x_1, ..., x_n, x'_1, ..., x'_{i-1}\}$. We use $\gamma_{i,j}$ to denote the j^{th} guard and $\delta_{i,j}$ to denote the j^{th} simple transition of x_i. Thus, $\delta_{i,j} = \alpha_{i,j} \wedge \beta_{i,j} \wedge \gamma_{i,j}$.*

Definition 6. *The **complete transition** for a variable x_i, denoted δ_i, is the disjunction of all simple transitions for x_i. Thus, $\delta_i = \bigvee_{j=1}^{n_i} \delta_{i,j}$, where n_i is the number of simple transitions for the variable x_i.*

Note that the definitions require an ordering among variables and thus eliminate circular dependencies among the post-state values of the variables. The transition for a variable can be viewed as a collection of triples, *(pre-state, post-state, guard)*, where each triple describes a part of the transition relation for the variable. If any of the components is absent, it is taken to be the constant predicate *true*.

Finally,

Definition 7. *The **transition relation** Δ, is the conjunction of the complete transitions of all the variables $x_1, ..., x_n$. Thus, $\Delta = \bigwedge_{i=1}^{n} \delta_i$.*

The initial state predicate, ρ is similar to a transition relation in which all the guards and the pre-state predicates are absent (i.e, equivalent to the constant predicate *true)*, and there are no unprimed variable references in the post-state predicates. Viewed this way, ρ can be thought of as specifying a transition that resets the system from any state to its initial state.

The demarcation of boundaries between pre-state, guard and post-state predicates may seem arbitrary. However, this structure makes it possible to use this formalism to capture systems described in a variety of languages. In some state-based specification languages and in the input languages of model-checkers like SMV, a post-state predicate of $\beta \equiv (x' = x)$ is implicitly assumed to hold when none of the explicitly specified transitions hold for the variable x. However, to conform to the structure of the transition relation defined above, such implicit transitions must be made explicit. Also, at times, a transition relation may have to be rewritten to an equivalent relation that conforms to this structure. Such issues must be handled by the mapping from a given language to this system model.

3.3. Basic Transition System

We now define a basic transition system M as a tuple, $M = (D, \Delta, \rho)$, where D represents the state-space of the system, Δ, represents the transition relation, and ρ characterizes the initial system state. This system model will serve as a basis for formulating various coverage criteria and for deriving properties that could be refuted by a model-checker yielding, in the process, valid execution sequences for the software artifact represented by the model.

4. Structural Coverage Criteria

In Section 2, we briefly discussed condition-based coverage criteria. Here we define a representative collection of condition-based criteria in terms of our system model.

First, we must formalize the notion of a *test case* and a *test suite*. The unit of interest to us in our system model is the simple transition, which, we may recall, is a triple of predicates, (α, β, γ) specifying the pre-state, post-state and guard respectively. Since predicates are parameterized by variable references and states are assignment of values to variables, it is meaningful to evaluate predicates using states as arguments. If p is a predicate and s_i and s_j are states, we use $p(s_i, s_j)$ to denote the value of the predicate

p obtained by substituting the unprimed variable references with the values of the corresponding variables in state s_i and the primed variable references with the values of the corresponding variables in state s_j. If a predicate is parameterized only by unprimed (or only by primed) variable references then we use $p(s_i)$ instead. A test case s is simply a sequence of states $\langle s_1, \ldots, s_m \rangle$, where each state is related to its successor by the transition relation Δ, of the system and $\rho(s_1)$ is true, i.e., $s_1$is an initial state. A test suite is a set of test cases.

4.1. Simple Transition Coverage

Definition 8. *A test suite is said to achieve simple transition coverage for a basic transition system $M = (D, \Delta, \rho)$, if for any simple transition (α, β, γ) of any variable x, there exists a test case s such that for some i, $\alpha(s_i) \wedge \beta(s_i, s_{i+1}) \wedge \gamma(s_i, s_{i+1})$ holds true.*

In other words, for every simple transition for each variable, there is a test case in the test suite in which the simple transition is taken. If we think of each simple transition as defining a possible case for a variable to change its value as the system moves from one state to another, this coverage criterion is analogous to branch coverage on code. Note that in this view of the system there is no distinction between pre-state, post-state and guard predicates and it is reflected in the criterion. Since the system model ensures that all possible transitions are specified explicitly, this coverage criterion is equivalent to saying that all branches are covered.

4.2. Simple Guard Coverage

Alternatively, if we think of transitions in terms of triples *(pre-state, post-state, guard)* and consider the guard as the equivalent of a decision point in the program, we could define guard coverage as guard taking both *true* and *false* values in the test suite. Formally,

Definition 9. *A test suite achieves simple guard coverage if for any simple transition (α, β, γ) there exist test cases s and t such that for some i and j:*

1. *$\alpha(s_i) \wedge \beta(s_i, s_{i+1}) \wedge \gamma(s_i, s_{i+1})$; i.e., the simple transition is taken in the transition from s_i to s_{i+1} in the test case s*
2. *$\alpha(t_j) \wedge \neg\beta(t_j, t_{j+1}) \wedge \neg\gamma(t_j, t_{j+1})$; i.e., the simple transition is not taken in the transition from t_j to t_{j+1} in the test case t*

Here the objective is to test both "branches" of the guard condition. Note that condition (1) by itself is equivalent to transition coverage which is therefore a weaker criterion than simple guard coverage. However, in practice, a test case to satisfy condition (2) can often be combined with one that satisfies condition (1) for a different guard, thus achieving simple guard coverage with a test suite which has roughly the same size as required for simple transition coverage.

4.3. Complete Guard Coverage

If simple guard coverage criterion is at one end of the spectrum (that does not consider the individual clauses making up the guard), complete guard coverage is at the other end, where all possible combinations of truth values of the guard clauses are tested. This, by definition, is exponential in the number of clauses in the guard and so can be used in practice only in guards with a small number of clauses. Formally,

Definition 10. *If the clauses in a guard γ of a simple transition (α, β, γ) are $\{c_1, ..., c_l\}$, a test suite that achieves complete guard coverage must include 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 is analogous to the multiple condition coverage in program source code. A similar criterion can be defined for the whole transition instead of considering the guard alone. If the clauses in the guard are not independent, then there will be vectors u for which no feasible solutions exist. In such cases it is typically the case that there is an error in the definition of the transition relation. When a model-checker is used to generate test cases this will manifest as an invariant property of the system.

4.4. Clause-wise Guard Coverage

Finally, we look at the code based coverage criterion called modified condition/decision coverage (MC/DC) and define an analogous criterion. MC/DC was developed to meet the need for extensive testing of complex boolean expressions in safety-critical applications [9]. Ideally, one should test every possible combination of values for the conditions thus achieving *multiple condition coverage*. However, the number of test cases required to achieve this grows exponentially with the number of conditions and hence becomes huge or impractical for systems with tens of conditions per decision point. MC/DC was developed as a practical and reasonable compromise between decision coverage and multiple condition coverage. It has been in use for several years in the commercial avionics industry. A test suite is said to satisfy MC/DC if executing the test cases in the test suite will guarantee that:

- every point of entry and exit in the program has been invoked at least once,
- every condition in a decision in the program has taken on all possible outcomes at least once, and
- each condition has been shown to independently affect the decision's outcome

where a condition is an atomic boolean valued expression that cannot be broken into boolean sub-expressions. A condition is shown to independently affect a decision's outcome by varying only that condition while holding all other conditions at that decision point fixed. Thus, a pair of test cases must exist for each condition in the test-suite to satisfy MC/DC. However, test case pairs for different conditions need not necessarily be disjoint. In fact, the size of

MC/DC adequate test-suite can be as small as $N + 1$ for a decision point with N conditions.

If we think of the system as a realization of the specified transition relation, it evaluates each predicate on each transition to determine which transitions are enabled and thus each predicate becomes a decision point. The predicates in turn are constructed from clauses – the basic conditions. For our current purposes, we focus on the guard condition of the transitions relations.

Definition 11. *A test suite S **covers** a clause c of the guard γ of a simple transition (α, β, γ), if it contains two test cases s and t, such that for some i and j:*

1. $\alpha(s_i) \wedge \beta(s_i, s_{i+1}) \wedge \gamma(s_i, s_{i+1})$ *; i.e., the simple transition is taken in the transition from s_i to s_{i+1} in the test case s,*
2. $\alpha(t_j) \wedge \neg\beta(t_j, t_{j+1}) \wedge \neg\gamma(t_j, t_{j+1})$*; i.e., the simple transition is not taken in the transition from t_j to t_{j+1} in the test case t,*
3. $c(s_i, s_{i+1}) = \neg c(t_j, t_{j+1})$*; i.e., the value of the clause c of the guard γ differs for the two transitions, and*
4. $\forall d \neq c$ *in γ, $d(s_i, s_{i+1}) = d(t_j, t_{j+1})$; i.e., all other clauses in the guard γ have the same value in both the transitions.*

*The test suite S meets the **clause-wise guard coverage** criterion for a basic transition system $M = (D, \Delta, \rho)$, if it covers each clause of each guard $\gamma_{i,j}$ in every simple transition $\delta_{i,j}$ in Δ.*

The intuition behind these conditions is that the two test cases s and t together show that the clause c independently affects the decision of whether or not the simple transition is taken. Note that the second condition imposes the constraint that the post-state predicate should not hold for test case t. This is in tune with the notion that a transition is considered to be taken if its pre-state and post-state specifications are met by a pair of consecutive states in a test case. It must be noted that in both the test cases, the transition relation for the system Δ should hold true, for each test case by our definition is a valid sequence of states.

If, however, we considered the simple transition as a whole without distinguishing the pre-state, the post-state and the guard components, then the criterion will require that the post-case predicate β must hold true also for t. This will require a test case where either the post-state is not reachable from the pre-state (Δ evaluates to false) or the post-state is reachable from the pre-state through some other simple transition (Δ evaluates to true). In the former case, every correct implementation should fail the test case, since the implementation should not reach an unreachable state. In the later case, the test cases show that the guard condition does not independently affect the decision of whether the particular transition is indeed taken, which typically points to a specification error. As another variation, we might consider test cases that satisfy a condition obtained by negating β in condition 1. This will lead to a test case with either an unreachable state or one that reveals non-determinism in the system. While such tests are interesting and useful, those are not within the scope of the method discussed here.

Why should we restrict the coverage requirements to just the guard condition? Should we not include all the clauses in every simple transition? It is definitely meaningful and necessary to test all the clauses. In fact, using the criteria as such on the simple transitions is straightforward. However, in our formulation of the basic system model, the intent is that the three components, pre-state, guard and post-state serve different purposes. We use pre-state predicate in a way similar to the nodes in a control-flow graph. It marks regions of interest in the state-space that the system may visit. The post-state predicate can be thought of as defining a functional relationship between inputs and outputs as the system moves across the state-space. The guard condition is similar to a decision point in a program and it governs how the system proceeds from one state to another. So, it is meaningful to apply condition coverage criteria to guard predicates.

5. Test Case Generation

Our approach to generating test cases involves using the model-checker as the core engine. A set of properties called *trap properties* [14], is generated and the model-checker is asked to verify the properties one by one. These properties are constructed in such a way that they fail for the given system specification, which in our case is the basic transition system $M = (D, \Delta, \rho)$, leading the model checker to produce a counter-example. The counter-example shows a valid sequence of states that any conforming implementation should follow. This sequence of states becomes a test case. The task is then to generate the trap properties in such a way that the set of counter-examples generated will be adequate to satisfy the coverage criteria that we are dealing with. The properties that we generate can be expressed in Linear Temporal Logic (LTL) which is handled by model-checkers like SPIN [21] and SMV [26]. We will demonstrate this process with the most complex of the coverage criteria that we discussed so far, namely, clause-wise guard coverage (CGC). The process of generating trap properties for other criteria are similar to this and also easier to generate because there are no dependencies across different test cases.

5.1. Trap Property Generation

The trap properties for CGC are derived from the four conditions defined in Subsection 4.4. The main issue, however, is that the constraints span two test cases s and t. A trap property can only produce one counter-example. So the method that generates these properties should handle the conditions that span multiple test cases. One way to handle this, is by duplicating the basic transition system M and construct a new system which models two instances of M simultaneously. In general, constraints that span p test cases in the original model, could be reduced to those that span a single test case in a model that includes p simultaneous copies of M. This apparently inefficient approach may work for some systems. The approach discussed here, however, uses only a single instance of the model and instead

uses the structure of the predicates to formulate a separate trap-property for each test case.

Revisiting the conditions defined earlier for the coverage criterion, we note that conditions (1) and (2) refer to separate test cases s and t, and so they can be directly used in formulating a pair of trap properties, one for each test case. Conditions (3) and (4) span the two test cases s and t. If we examine these conditions, they express constraints on a per-clause basis for each clause of the guard-condition. The constraints specify either that a clause should evaluate to the same value in both the test cases [condition (4)] or that a clause should evaluate to different values in the two test cases [condition (3)]. Our approach is to instantiate the clauses to specific values and use those values explicitly in the trap properties.

Leaving aside, for the moment, the task of actually finding those values, assume that there exists a pair of boolean vectors u and v, each of length equal to l, the number of distinct clauses in the guard γ under consideration. Assume that the clauses in γ are $\{c_1, ..., c_l\}$. Suppose the clause under consideration is c_m. Then we may rewrite the conditions as:

1. $\alpha(s_i) \wedge \beta(s_i, s_{i+1})$,
2. $\alpha(t_j) \wedge \neg\beta(t_j, t_{j+1})$,
3. $\bigwedge_{k=1}^{l} c_k(s_i, s_{i+1}) = u_k$, and
4. $\bigwedge_{k=1}^{l} c_k(t_j, t_{j+1}) = v_k$

with the following constraints on u and v :

- $u_m = \neg v_m$; i.e. the two boolean vectors u and v differ in their m^{th} component
- $\bigwedge_{k=1,k\neq m}^{l} u_k = v_k$; i.e.. the two boolean vectors u and v agree in all other components
- $\gamma|_{c_i=u_i} \wedge \neg\gamma|_{c_i=v_i}$; i.e. the guard condition γ evaluates to true and false respectively when the clauses c_k are substituted with the corresponding values from the vectors u and v .

With this formulation, it is clear that properties (1) and (3) deal with only test case s and similarly properties (2) and (4) deal only with test case t. We now formulate a pair of trap properties as follows:

1. $G\left(\alpha \wedge \left(\bigwedge_{k=1}^{l} c_k = u_k\right) \Rightarrow \neg\beta\right)$; i.e., it is globally true for the basic transition system M that if the pre-state condition α holds and the clauses in the guard evaluate to truth values indicated by the vector u, then the post-state condition β will not hold. A counter-example to this will be a transition from a reachable state s_i to a state $s_{i+1}$where the pre-state condition α holds at s_i and the post-state condition β holds at state $s_{i+1}$and the clauses in the guard evaluate to the truth values specified by the vector u, making the guard γ hold true.
2. $G\left(\alpha \wedge \left(\bigwedge_{k=1}^{l} c_k = v_k\right) \Rightarrow \beta\right)$; i.e., it is globally true for the basic transition system M that if the pre-state condition α holds and the clauses in the guard evaluate to truth values indicated by the vector v then the post-state condition β will hold. A counter-example to this will be a transition from a reachable state s_i to a state $s_{i+1}$where the pre-state condition α holds at s_i but the post-state condition β does not hold at state $s_{i+1}$and the clauses in the guard evaluate to the truth values specified by the vector v, making the guard γ false.

Note that we have moved the guard γ to the constraints on the vectors. This reformulation is equivalent to the original set of properties under the following assumption:

> If a clause c independently affects the value of a guard γ in a basic transition system M, then for any assignment of truth values to the remaining clauses in γ that does not mask the effect of c on γ, there exists two test cases s and t of the basic transition system M that stand witness to the independence of c, simultaneously conforming to the truth assignment for the remaining clauses.

It is possible that this does not hold in some cases. If so, the model-checker may end up proving the trap property. If that happens, it may mean that the property is an invariant of the system or that there is a specification error. In either case, additional work is required either in constructing a different trap property, or in correcting the specifications before proceeding.

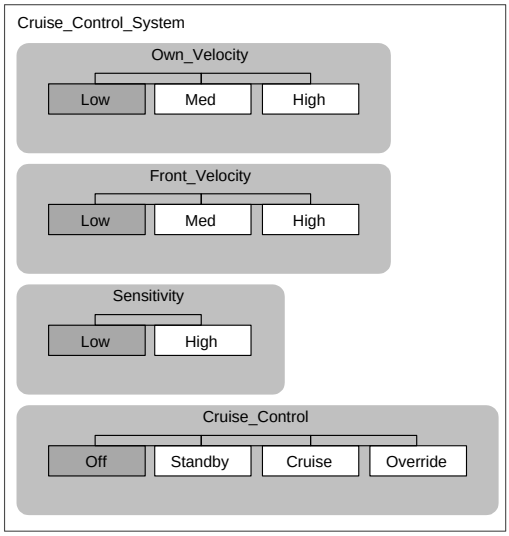
The constraints on u and v must be handled by the mechanism that produces the two vectors. The vector pair could be generated, for example, using the method discussed in[27] or [23]. We could also use the model-checker itself to generate the (u, v) pair by letting u and v vectors as input variables for a system that models two instances of the guard γ in terms of the components of u and v vectors respectively, and asserting that there is no such pair (u, v) for which the guard evaluates to different truth values:
$G\left(\left(u_m = \neg v_m \wedge \bigwedge_{k=1,k\neq m}^{l} u_k = v_k\right) \Rightarrow (\neg\gamma_u \vee \gamma_v)\right)$
The counter example produced will be an assignment of truth values to the boolean vectors u and v such that they differ only in the m^{th} component and the guard γ evaluates to *true* under u and *false* under v.

6. Illustrative Example

We now illustrate the process using a simple example. We use an RSML^{-e} model of a cruise control system and generate test cases for CGC. In the process, we also describe the mapping from the RSML^{-e} language specifications to the basic transition system model. The structure of the specifications, in particular, the tables describing the next state transitions are used to generate the vectors u and v that instantiate the truth assignments for the clauses.

The example we consider is a simple cruise-control system that appeared in [8]. It combines a small part of the collision-avoidance logic from the TCAS II specification [24] with cruise-control logic for automobiles. We specified this system using RSML^{-e} and translated it[1] to

[1]The translation was made by hand, but the semantics of RSML^{-e} easily lends itself to automatic translation.



Variable: Sensitivity
Location: Cruise_Control_System
Transition: Low ⟶ High
Condition:

Sensitivity_Setting = Sensitivity_Type::Sens_High	T	*
..Own_Velocity IN_STATE High	*	T
..Front_Velocity IN_STATE Low	*	T

Figure 2. Cruise Control System

SMV (Cadence Berkeley Labs version) [25], to generate test cases. The complete SMV code for the example is available in [28]. Figure 2 shows a portion of the RSML^{-e} specification containing the state machine and a definition of a state transition from state *Low* to state *High* for the state variable *Sensitivity*. Each column represents a conjunction of the truth values in the column and the columns form a disjunction, A * represents a *don't care* condition. If we were instead interested in generating test cases from source code, the transition may be implemented as a decision-point in the system event-loop that assigns values to variables, as follows:

```
if (Prev_Sensitivity == ST_Low)
  if (SensitivitySetting == TY_Sens_High
      || (Own_Velocity == High
          && Front_Velocity == Low))
    Sensitivity = ST_High;
```

Either way, the system can be modeled as a basic transition system and used in our approach. The system has four state variables `Own_Velocity`, `Front_Velocity`, `Sensitivity` and `Cruise_Control`. The velocities can be in one of the three states, `Low`, `Medium`, `High` and the cruise control system can be in one of the four states `Off`, `Standby`, `Cruise` and `Override`. The sensitivity can be `High` or `Low` depending on user setting or based on the states of the own and front vehicle velocities.

Let us consider the transition from `Low` to `High` for `Sensitivity`. This transition can be taken if either the sensitivity is set to high or the velocity of own vehicle is high while that of the front vehicle is low. Thus the guard condition for this simple transition is:

```
DV_SensitivitySetting = TY_Sens_High /*1*/
| (EQ_Own_Velocity = ST_High         /*2*/
   & EQ_Front_Velocity = ST_Low)     /*3*/
```

There are three clauses and hence a total of six trap properties are required to generate test cases to cover the guard. The following three (u, v) boolean vector pairs can be used: `([T F F],[F F F]), ([F T T],[F F T]), ([F T T],[F T F])`. In each pair, the first boolean vector makes the guard true and the second makes it false. Since the second and third pairs have a vector in common, there are only five distinct trap properties: (the "P" prefixed variables store the value of the variable in the previous state):

```
1. G( (P_Sensitivity = Low)
     &  (DV_Sensitivity = TY_Sens_High)
     & !(EQ_Own_Velocity = ST_High)
     & !(EQ_Front_Velocity = ST_Low)
    => !(Sensitivity = High))

2. G(  (P_Sensitivity = Low)
     & !(DV_Sensitivity = TY_Sens_High)
     & !(EQ_Own_Velocity = ST_High)
     & !(EQ_Front_Velocity = ST_Low)
    =>  (Sensitivity = High))

3. G(  (P_Sensitivity = Low)
     & !(DV_Sensitivity = TY_Sens_High)
     &  (EQ_Own_Velocity = ST_High)
     &  (EQ_Front_Velocity = ST_Low)
    => !(Sensitivity = High))

4. G(  (P_Sensitivity = Low)
     & !(DV_Sensitivity = TY_Sens_High)
     & !(EQ_Own_Velocity = ST_High)
     &  (EQ_Front_Velocity = ST_Low)
    =>  (Sensitivity = High))

5. G(  (P_Sensitivity = Low)
     & !(DV_Sensitivity = TY_Sens_High)
     &  (EQ_Own_Velocity = ST_High)
     & !(EQ_Front_Velocity = ST_Low)
    =>  (Sensitivity = High))
```

Properties 1 and 3 assert that the transition is not taken, while properties 2, 4 and 5 assert that the transition is taken. Each of these, when asserted as a global property of the system, forces the model checker to produce a counter-example to the assertion. For example, when property 1 is asserted, SMV produces a two-step counter-example:

```
   DV_Front_Speed  0            50
     DV_Own_Speed  0            0
   DV_Sensitivity  TY_Sens_Low  TY_Sens_High
EQ_Front_Velocity  ST_Low       ST_High
  EQ_Own_Velocity  ST_Low       ST_Low
   EQ_Sensitivity  ST_Low       ST_High
 P_EQ_Sensitivity  ST_NONE      ST_Low
```

This counter-example shows a case where the sensitivity changes because of the change in the sensitivity setting by the user, which was the intended test case for which the trap property 1 was generated. Similarly, the intent of property 2 is to show that under similar conditions when sensitivity setting is not changed by the user, the `Sensitivity` state remains in `ST_Low` and the counter-example produced by SMV demonstrates this:

```
   DV_Front_Speed  0            50
     DV_Own_Speed  0            0
   DV_Sensitivity  TY_Sens_Low  TY_Sens_Low
EQ_Front_Velocity  ST_Low       ST_High
  EQ_Own_Velocity  ST_Low       ST_Low
   EQ_Sensitivity  ST_Low       ST_Low
 P_EQ_Sensitivity  ST_NONE      ST_Low
```

In the above counter examples, the input data variables are the ones prefixed with "DV". The values of these `DV_` variables across the steps form a sequence of test inputs. The values that we would like to observe are those of the state variables which are prefixed with "EQ". These form the expected system outputs. Any implementation conforming to the specification of the cruise control system should produce these outputs for the given test inputs.

7. Related Work

Researchers have looked at various ways of defining test criteria based on specification. Recently, Offutt, *et al.* [27], defined various specification-based test coverage criteria for generating tests from state-based specifications and measured their effectiveness in terms of their fault-finding capabilities when used on a Cruise Control system specification written in SCR. In our current work, we have defined the criteria in terms of an underlying formal definition of the system model and proposed a systematic method to generate test cases.

To our knowledge, two other research groups have attempted to use model checking to generate test cases. Gargantini and Heitmeyer [14] describe a method for generating test sequences from requirements specified in the SCR notation. Our approach to test case generation using model-checker is similar to theirs. To derive a test sequence, a trap property is defined which violates some known property of the specification. The model-checker is then used to produce a counter-example to the trap property. The counter-example assigns a sequence of values to the abstract inputs and outputs of the system, thus making it a test sequence. The high level of abstraction used when modeling with SCR, requires that the abstract sequences must be instantiated with actual data, a non-trivial task.

Ammann and Black [2, 1] combine mutation analysis with model-checking based test case generation. They define a specification based coverage metric for test suites using the ratio of the number of mutants killed by the test suite to the total number of mutants. Their approach uses a model-checker to generate mutation adequate test suites. The mutants are produced by systematically applying mutation operators to both the properties specification and the operational specification, producing respectively, both positive test cases which a correct implementation should pass, and negative test cases which a correct implementation should fail.

Blackburn and Busser [5] map specifications written in SCR to the formalism of their T-VEC tools in terms of pre-conditions and functional mappings from inputs to outputs. The T-VEC tool then generates test vectors as pairs of pre/post system states. However, a valid sequence of inputs that leads the system to the state-pair is not generated. Burton, *et al.* [6], discuss the use of theorem-proving approach for generating test cases. Callahan, *et al.* [7], describe how traces from a simulation of the implementation can be verified by a simulation of the specification where the expected outputs are computed using the SPIN model-checker. Engels, *et al.* [13], also use SPIN to generate test sequences where the *testing purpose* is defined as a verification goal by the test engineer.

The main challenge for all these approaches is to find appropriate abstraction techniques to reduce the model size. State space explosion and infinite state spaces (caused by floating point and integer variables) are major obstacles in model checking software artifacts that must be addressed. Work in the area of abstraction techniques [10, 22, 19, 31] can be leveraged to create models amenable to model-checking. However, the main issue is the loss of detail in the abstraction process that makes instantiation of a test sequence from a counter example a non-trivial task.

8. Conclusion and Future Work

We have demonstrated an approach to automate test-case generation for software engineering artifacts such as code and formal specifications. We use structural coverage criteria to define what test cases are needed and the test cases are then generated using the power of a model checker to generate counter-examples. We explained the process with an example. Initial results indicate that the approach will scale well to larger systems and have the potential to dramatically reduce the costs associated with generating test cases to high levels of structural coverage.

The use of model-checkers as a test generation tool is being actively explored by some other researchers [4, 1, 14]. There are, however, several challenges that need to be addressed in the future. First, the problem of state space explosion can affect the search for counter-examples. Abstraction techniques that reduce the state-space could make instantiating the test case with concrete data somewhat difficult since the counter example would be presented in the abstract domain. One of the main advantages of a model-checking approach to test generation is automatic instantiation of the test sequences with actual data for the system inputs and outputs. Ways of instantiating counter-examples with specific data values in the presence of abstraction must be investigated. Second, environment specification is always a difficult issue in modeling systems. Often, the environment is modeled in a conservative fashion allowing more behavior for the environment than is actually possible. When such a model of the environment is used, the generated test cases may not be useful since they may represent

scenarios that cannot happen. A human tester would possibly take environment constraints into account and generate more *realistic* test cases. Incorporating environment constraints in the trap properties without adversely affecting the search for counter-example is a problem worth exploring.

References

[1] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering*. IEEE Computer Society, Nov. 1999.

[2] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, Nov. 1998.

[3] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[4] P. E. Black. Modeling and marshaling: Making tests from model checker counterexamples. In *Proc. of the 19th Digital Avionics Systems Confrence*, October 2000.

[5] M. R. Blackburn, R. D. Busser, and J. S. Fontaine. Automatic generation of test vectors for SCR-style specifications. In *Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS'97*, June 1997.

[6] S. Burton, J. Clark, and J. McDermid. Testing, proof and automation. An integrated approach. In *Proceedings of First International Workshop on Automated Program Analysis, Testing and Verification*, June 2000.

[7] J. Callahan, F. Schneider, and S. Easterbrook. Specification-based testing using model checking. In *Proceedings of the SPIN Workshop*, August 1996.

[8] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *Proc. of CAV'97, LNCS 1254*, pages 316–327. Springer, June 1997.

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

[10] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transaction on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[11] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[12] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. of the 22nd Int'l Conf. on Software Engineering*, pages 439–448, June 2000.

[13] A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Proceedings of TACAS'97, LNCS 1217*, pages 384–398. Springer, 1997.

[14] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

[15] J. B. Goodenough and S. L. Gerhart. Toward a theory of testing: Data selection criteria. In R. T. Yeh, editor, *Current trends in programming methodology*. Prentice Hall, 1979.

[16] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.

[17] M. P. Heimdahl, J. M. Thompson, and B. J. Czerny. Specification and analysis of intercomponent communication. *IEEE Computer*, pages 47–54, April 1998.

[18] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[19] C. Heitmeyer, J. K. Jr., B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[20] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

[21] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software Engineering*, pages 279–295, May 1997.

[22] G. J. Holzmann. Designing executable abstractions. In *Proc. of the 2nd Workshop on Formal Methods in Software Practice*. ACM, 1998.

[23] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proc. of Int'l Symp. on Software Testing and Analysis*, pages 95–107, August 1994.

[24] N. G. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. TCAS II Requirements Specification.

[25] K. L. McMillan. Symbolic Model Verifer (SMV) - Cadence Berkeley Laboratories Version. Available at http://www-cad.eecs.berkeley.edu/~kenmcmil/smv.

[26] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[27] A. J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, October 1999.

[28] S. Rayadurgam and M. P. Heimdahl. Coverage based test data generation using model checkers. Technical Report 01-005, Dept. of Computer Science and Engineering, University of Minnesota, Minneapolis, January 2001.

[29] RTCA. *Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[30] J. M. Thompson, M. W. Whalen, and M. P. Heimdahl. Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757, July 2000.

[31] M. Young. How to leave out details: Error-preserving abstractions of state-space models. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis (TAV 2)*, 1988.

[32] H. Zhu, P. Hall, and J. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.