Model Checkers in Software Testing

Paul Ammann
George Mason University

Paul E. Black
NIST

Wei Ding
George Mason University

February 1, 2002

Abstract

The primary focus of formal methods is static analysis of specifications and code, but there is also a long tradition of exploiting formal methods for testing. This paper continues this tradition by exploring the role of model checkers in software testing. Model checkers were originally developed to check that state machines conformed to specifications expressed in a temporal logic. We show how to apply the powerful computation engines in model checkers to the problems of test generation and test evaluation for a variety of test coverage criteria defined on model-based specifications.

1 Introduction

The use of formal methods has been widely advocated to reduce the likelihood of errors in the early stages of system development. Some of the chief drawbacks to applying formal methods is the difficulty of conducting formal analysis [10] and the perceived or actual payoff in project budget. Testing is an expensive part of the software budget, and formal methods offer an opportunity to significantly reduce the testing costs.

A broad span of research from early work on algebraic specifications [24] to more recent work such as [44] addresses the problem of relating tests to formal specifications. Counterexamples from model checkers have been recognized as potentially useful test cases. Callahan, Schneider, and Easterbrook used a model checker to generate tests that cover each block in a certain partitioning of the input domain [14]. Engels *et al* used a model checker to generate network tests [23]. Ammann *et al* defined a mutation analysis approach to generating and recognizing tests with a model checker [5, 4]. Gargantini and Heitmeyer used model checkers [25] to generate tests for systems with SCR (Software Cost Reduction) [28] requirement specifications.

Current Mode	Event	New Mode
$TooLow$	$@T(WaterPres \geq Low)$	$Permitted$
$Permitted$	$@T(WaterPres \geq Permit)$	$High$
$Permitted$	$@T(WaterPres < Low)$	$TooLow$
$High$	$@T(WaterPres < Permit)$	$Permitted$

Initial State : Mode $= TooLow, WaterPres < Low$

Mode Transition Table for *Pressure*.

Mode	Events	
$High$	$False$	$@T(Inmode)$
$TooLow$, $Permitted$	$@T(Block = On)$ $WHEN\ Reset = Off$	$@T(Inmode)\ OR$ $@T(Reset = On)$
$Overridden$	$True$	$False$

Event Table for *Overridden*.

Mode	Conditions	
$High,\ Permitted$	$True$	$False$
$TooLow$	$Overridden$	NOT $Overridden$
$Safety\ Injection$	Off	On

Condition Table for *Safety Injection*.

Table 1: Safety Injection Tables

1.1 Running Example

To illustrate our work, we use the Safety Injection problem as a running example. Table 1 gives the SCR tables from [8] for this problem. It comes from part of a nuclear reactor safety system. Broadly, if the water pressure is too low, reserve water is injected, unless overridden. The system is overridden depending on the pressure, whether the override is blocked, and whether it is reset. The notation $@T(expr)$ signifies *expr* becoming true. [8] describes SCR and the Safety Injection model in detail. The Appendix also has a description of Safety Injection in the formalism of the Symbolic Model Verifier (SMV) model checker [35, 13]. In Section 4.2.1, we give the results of applying the mutation model described in this paper to the example.

1.2 What is a Model Checker?

A model checking specification consists of two parts. One part is the model: a state machine defined in terms of variables, initial values for the variables, and a description of the conditions under which variables may change value. The state space is defined by the possible combinations of valuations for the variables. The other part is temporal logic constraints over states and execution paths. Conceptually, a model checker visits all reachable states and verifies that the temporal logic properties are satisfied over each possible path, that is, the model checker determines if the state machine is a model for the temporal logic formula. Model checkers exploit clever ways of avoiding brute force exploration of the state space, for example, see [13, 20]. If a property is not satisfied, the model checker attempts to generate a counterexample in the form of a trace or sequence of states. For some temporal logic properties, no counterexample is possible. For example, if the property states that at least one possible execution path leads to a certain state and in fact no execution path leads to that state, there is no counterexample to exhibit.

The model checking approach to formal methods has received considerable attention in the literature, and readily available tools such as SMV [35, 13] for the Computation Tree Logic (CTL) and SPIN [30] for the Linear Time Logic (LTL) are capable of handling the state spaces associated with realistic problems [19]. Although model checking began as a method for verifying hardware designs, there is growing evidence that model checking can be applied with considerable automation to specifications for relatively large software systems, such as the Traffic Alert and Collision Avoidance System II (TCAS II) [15]. The increasing usefulness of model checkers for software systems makes them attractive targets for use in aspects of software development other than pure analysis, which is their primary role today.

Model checking has been successfully applied to a wide variety of practical problems, including hardware design, protocol analysis, operating systems, reactive system analysis, fault tolerance, and security. The chief advantage of model checking over the alternate approach of theorem proving is complete automation. Human interaction is generally required to prove all but the most trivial theorems. Readily available model checkers such as SMV and SPIN can explore the state spaces for finite, but realistic, problems without human guidance [19].

We use the SMV model checker. It is freely available from Carnegie Mellon University and elsewhere. The model checking algorithm in SMV has the advantage of being breadth first; hence the counterexamples that we interpret as test cases tend to be short.

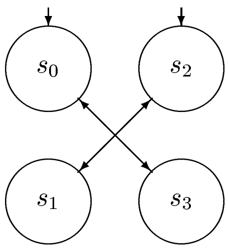
1.3 What is a Test Case?

For our purposes, state machine Σ is a tuple (S, δ, S_0) where S is a set of states, δ is a transition relation, and $S_0 \subseteq S$ is a nonempty set of initial states. Typically, Σ is specified with some succinct description of a state machine, SM, to avoid enumerating the potentially large state space. As a very small example, consider the simple state machine in Figure 1(a). Below, in Figure 1(b), is an explicit enumeration of Σ.

Figure 1(c) is a description SM in an SMV-like syntax. `VAR` declares two boolean variables, `x` and `y`. State s_i is represented by encoding i in binary with the string `xy`, where $false$ is 0 and $true$ is 1. The `ASSIGN` section gives both initial and subsequent values for each variable. The `next` value of `x` is $false$, in case `x` is $true$, or $true$, in the case of `!x`. The keyword `esac` ends the case statement. Assignments need not be deterministic; for example, the initial assignment for `x` may be either $true$ or $false$.

Note that the description SM is not unique for a given Σ. For this state machine, we could use a single variable, `mstate`, with states 0, 1, 2, and 3 and give subsequent values with `next(mstate) := 3 - mstate;`. Another alternative is to use four boolean variables, `s0`, `s1`, `s2`, and `s3`. Many other descriptions are possible.

A machine Σ defines γ, a possibly unbounded set of traces, where each trace is a sequence of states. Each sequence in γ begins with a state in the set of initial states, S_0, and continues, as allowed by the transition relation, δ, with zero or more states from S. A *test set* T is a nonempty, finite subset of γ. Each *test case* $t \in T$ is further required to be of finite length. Note that the notion of a test case here is more abstract than that for traditional program testing, where a test case is usually specified as an input/output pair. We ignore the difference between inputs and outputs at this point; both are merely encoded in the state description. Of course, the traces we derive here must eventually be refined into traditional test cases suitable for execution by an implementation. As an example, for the machine in Figure 1, T might include the test t equal to $s_2,\ s_1,\ s_2$. In the description SM, this corresponds to the sequence of $(x,\ y)$ pairs $(true,\ false)$, $(false,\ true)$, $(true,\ false)$. Suppose that the system modeled by Σ has x as an input and y as an output. Executing test t requires starting the system with input $x = true$, verifying that output y is $false$, toggling input x to $false$, verifying that output y is $true$, then setting input x back to $true$, and verifying that output y returns to $false$.



(a) Example State Machine

$$S = \{s_0,\ s_1,\ s_2,\ s_3\}$$
$$\delta = \{(s_0,\ s_3),\ (s_3,\ s_0),\ (s_1,\ s_2),\ (s_2,\ s_1),\}$$
$$S_0 = \{s_0,\ s_2\}$$

(b) Explicit Enumeration of Σ

```
VAR   x: boolean; y: boolean;
ASSIGN
   init (x) := {false, true}; init (y) := false;
   next (x) := case x: false; !x: true; esac;
   next (y) := case y: false; !y: true; esac;
```

(c) Description SM

Figure 1: Example of Σ vs. SM

2 Evaluating and Generating Tests With Model Checkers

In this section we address the potentials of evaluating and generating test sets with model checkers.

2.1 Evaluating Test Sets

Except in the smallest case, it is impractical to evaluate a piece of code on all of its behavior. Even if it is purely combinatorial or functional, that is, the output is a function of the input with no memory or state, the number of input sets easily becomes too large to exhaustively try. When a system has memory, input histories or traces must be used. While no substitute for good design and development practices, testing is necessary to confirm the quality of an implementation. Thus the tester must choose some relatively tiny set of tests.

Can a tiny set of tests, no matter how chosen, really be expected to detect most faults? The "coupling effect hypothesis" [21], supported in [38], states that tests that detect simple faults are likely to detect complex faults, too. Thus the tester has some confidence that choosing tests to find simple faults may suffice. Test sets can be chosen by many different criteria, such as, random sample, frequency of use, critical cases, domain boundary, coverage, etc.

Coverage or adequacy is an intuitively appealing measure: if tests don't exercise "all" the aspects of a piece of software, they are more likely to miss faults. Also unlike frequency of use and critical cases, we can measure coverage without needing to know how the software is used. One simple notion of "all" is every statement in the implementation; the aspects are simply executions of each statement. In the following code, if `a` is greater than 7, all the code is executed. On the other hand, if `a` is less than or equal to 7, the statement in the body of the conditional is not executed. We can turn this definition into a coverage measure by dividing the number of aspects satisfied, three statements in the first case and two statements in the second, by the total number of aspects, three statements for this example.

```
b := 7;
c := b - a;
if (a > b) then
  c := a - b;
endif
```

Other definitions, and hence coverage measures, of the aspects of a piece of software are certainly possible. For instance, branch coverage requires all control transfers to be exercised. The above code needs two tests, one with $a > 7$ and one with $a \leq 7$, to test when the branch is taken and when the branch is not taken. In fact we can compare any particular test set against a criterion to determine if and how well it satisfies the criterion.

We can divide test criteria into two broad categories: those that apply to source code, see for instance the extensive survey in [46], and those that apply to specifications, for instance [22, 36, 17, 1, 16, 14, 23, 25, 26, 4]. Metrics based on source code are more clearly tied to the implementation. However specification-based metrics allow some language independence and are applicable even without source code, for instance, in the design phase or during acceptance testing.

After [27] we say that a *test criterion*[1] is a predicate that defines properties of a specification to be exercised to constitute a "thorough" test, i.e., one whose successful execution provides strong evidence that the software implements the specification. More simply, since software should correspond with its specification, a test set with better coverage of the specification is likely to more accurately assess the quality of a piece of software than a test set with poorer coverage.

Measuring coverage using a model checker begins with a specification of the system and a set of tests to be evaluated against it; see Figure 2. Although the specification need not be a complete description of all behavior of the system, the more detailed it is, the more that can be checked. The set of tests is converted to a set of finite state machines. Each machine is a subset of possible system behavior; the subset is the behavior given by the test. Consider the following simplified test case, which essentially turns `Reset` on then off again. The first line indicates that `Reset` and `Block` should be `Off`and `Pressure` should be `TooLow`. In the next execution step `Reset` should be `On`. Since variables not reported are unchanged from the previous step, `Block` should still be `Off` and `Pressure` should be `TooLow`. In the final step, `Reset` should be `Off` again, and `Block` and `Pressure` unchanged.

```
Reset = Off; Block = Off; Pressure = TooLow; STEP;
Reset = On; STEP;
Reset = Off; STEP;
```

A state machine representing this test case is constrained to start with `Reset` and `Block` `Off` and `Pressure` `TooLow`, then set `Reset` `On`, and finally, set `Reset` `Off` with no further change in the variables. Section 2.1.1 explains in more detail the process of turning a test into an appropriate state machine.

The next step is that some test criterion is chosen, such as mutation adequacy [4], conjunctive complementary closure (CCC) partitions [14], modified condition/decision coverage (MC/DC) [16], automata theoretic [17], branch coverage [25], transition pair coverage [41], disconnection or redirection faults [26], stuck-at faults [1], etc. For a specification, the criterion defines a set of properties. For a test set to satisfy the test criterion, each property must hold for at least one test case in the test set. These properties are called *test requirements*. Section 3 describes some different criteria and how the corresponding test requirements can be derived for a given specification.

[1] Our notion of test criteria falls under the general concept of "test purposes" in [23].

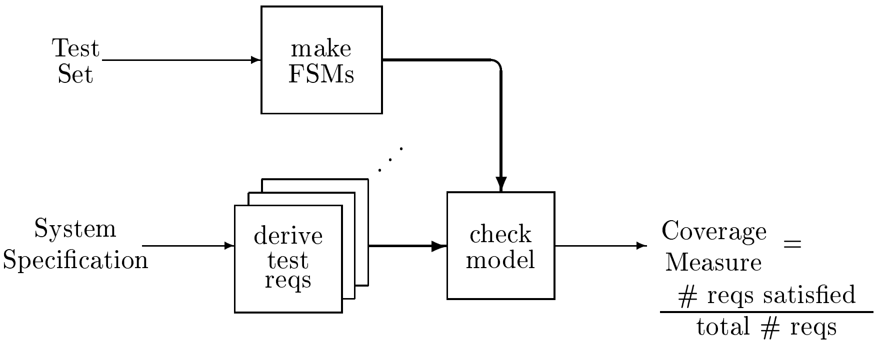


Figure 2: A Specification-Based Test Measurement

A particular criterion may express requirements either negatively or positively. That is, a requirement may be considered satisfied if it is found to be inconsistent with an execution (negative), or it may be satisfied if it is consistent (positive). Suppose we apply a simple state coverage criterion to the safety injection example: a test set is adequate if `Pressure` takes the values `TooLow`, `Permitted`, and `High`.

In the CTL notation used in SMV, and which we use other places in this paper, `A` is the universal quantifier all across possible alternative traces. The notation `G` suggests "global", that is, for all future states, `F` means for some states in the "future," and the notation `X` means "next" states. Hence, `AG` specifies properties that should hold in every future state on all possible traces. That is, `AG` specifies invariants. `E`, recalling "there exists" ($\exists$) in standard mathematical notation, is the existential quantifier across possible alternative traces. Hence, `EF` specifies properties that should hold in some future state of some possible trace. Logical conjunction is represented by `&`, `->` is implication, `<->` is equivalence, and `!` is logical negation.

Stated as positive requirements, state coverage can be checked with

```
EF Pressure = TooLow
EF Pressure = Permitted
EF Pressure = High
```

If these are found to be consistent, the requirements are satisfied. Stated as negative requirements, state coverage is checked with the following requirements.

```
AG !(Pressure = TooLow)
AG !(Pressure = Permitted)
AG !(Pressure = High)
```

If these are found to be inconsistent, the requirements are satisfied. Negative requirements, or "never-claims" as they are called in [23], can be used to generate tests, as we explain in Section 2.2.

2.1.1 Symbolic Execution of Test Cases

How can test cases be evaluated against requirements? Conceptually a test case is a single trace through the state machine that represents the behavior of the system. We can express the test case as a *constrained finite state machine*, or CFSM, by adding a special variable, *State*, that controls the machine. Each original variable gets a new value depending solely on *State*. If not specifically changed, it is unchanged.

Consider again the simple test case that turns `Reset` on then off.

```
Reset = Off; Block = Off; Pressure = TooLow; STEP;
Reset = On; STEP;
Reset = Off; STEP;
```

The corresponding constrained finite state machine may be described in SMV as follows. The controlling variable, *State*, is declared to have three possible values, 0 through 2. The `init` statements set each variable to the appropriate initial value. Since `Block` and `Pressure` do not change during the test, their next-state specifications are trivial. The value of `Reset` is driven solely by the *State*. *State* is incremented twice. Note that this specification is automatically generated, not hand-crafted.

```
VAR
State : 0..2;

ASSIGN
init(Reset):= Off;
init(Block):= Off;
init(Pressure):= TooLow;
init(State):=0;

next(Block) := case
        1 : Block;
esac;
next(Pressure) := case
        1 : Pressure;
esac;
next(Reset) := case
        State = 0 : On;
        State = 1 : Off;
        1 : Reset;
esac;
next(State) := case
        State < 2: State + 1;
        1 : State;
esac;
```

The final step to evaluate a test set is to check, or symbolically execute, the CFSM's one at a time against the requirements. An adequacy or coverage measure is the ratio of the number of requirements satisfied to the total number of requirements. Higher ratios indicate that the test set covers the specification more completely or better. Lower ratios indicate less complete coverage. We give details of how to compute a more accurate coverage in Section 2.1.2. The flow in Figure 2 is then a metric, or standard way of measurement, of how completely a test set exercises a specification.

Handling the End of Test

There are some additional details in converting test cases to CFSM's. In a reactive system, such as Safety Injection, freezing the state at the end of the test may be fine. However consider systems that have no quiescent state, such as a free-running counter. A consistent specification indicates that the state always changes. The specification is inconsistent with any CFSM generated as described, since the state freezes. However conceptually the specification is not wrong. Also, model checkers generally treat deadlock—that is, the absence of a next state—as an error, and so it is useful to encode test cases in such a way that spurious results are not generated by any behavior, or lack thereof, beyond the end of a test.

To avoid false reports of inconsistency, one may elaborate the CFSM with a special variable or macro, *Sustain*[2], that has the value $false$ when the test ends. All the specifications are mechanically rewritten to include *Sustain* and evaluate to *true* when *Sustain* is $false$. This rewriting is detailed and proved correct in [3]. For the above test, all we need add is a definition of the following macro.

```
Sustain := State < 2;
```

Implementing *Sustain* as a variable may make evaluating temporal logic constraints faster, because the macro is not evaluated for each constraint. However since model checkers are usually limited by the state space, a macro probably allows larger models to be checked.

For SMV [35, 13], the requirements, rendered in CTL [18], are elaborated according to the following rules. If a CTL formula does not begin with a temporal operator, it is rewritten as an implication so it has the value *true* when *Sustain* is $false$. Otherwise the temporal operator rewriting rule is applied.

$$CR(f) = \begin{cases} cr(f, True) & \text{if } f \text{ begins with a temporal operator} \\ Sustain \rightarrow cr(f, True) & \text{otherwise} \end{cases}$$

Formulae are rewritten recursively so that embedded temporal operators, that refer to future states, are rewritten to be *true* when *Sustain* is $false$ in those future states. Constraint rewriting with a value, $cr(f, v)$, tracks whether the formula has been negated. If the formula is a logical negation, implication, or equivalence, the value is negated in rewriting some of the subexpressions.

[2] Callahan, Easterbrook, and Schneider name a similar variable **done** [14].

Otherwise the subexpressions are rewritten with the value unchanged.

$$\begin{array}{lcl}
cr(!\,f,v) & = & !\,cr(f,\tilde{\ }v) \\
cr(f\ \&\ g,v) & = & cr(f,v)\ \&\ cr(g,v) \\
cr(f\mid g,v) & = & cr(f,v)\mid cr(g,v) \\
cr(f\ {->}\ g,v) & = & cr(f,\tilde{\ }v)\ {->}\ cr(g,v) \\
cr(f\ {<\!->}\ g,v) & = & cr(f,\tilde{\ }v){->}cr(g,v)\ \&\ cr(g,\tilde{\ }v){->}cr(f,v)
\end{array}$$

If the formula is a temporal operator, it is rewritten so the expression becomes $true$ (or $false$) when $Sustain$ is $false$. The operators AG, AF, AX, EG, EF, and EX follow these patterns. The meta-variable `OP` represents one of these six operators.

$$\begin{array}{lcl}
cr(\texttt{OP}\ f,True) & = & \texttt{OP}\ Sustain\ {->}\ cr(f,True) \\
cr(\texttt{OP}\ f,False) & = & \texttt{OP}\ Sustain\ \&\ cr(f,False)
\end{array}$$

The operators A...U and E...U follow these patterns.

$$\begin{array}{lcl}
cr(\texttt{OP}g\texttt{U}f,True) & = & \texttt{OP}g\texttt{U}Sustain\ {->}\ cr(f,True) \\
cr(\texttt{OP}g\texttt{U}f,False) & = & \texttt{OP}g\texttt{U}Sustain\ \&\ cr(f,False)
\end{array}$$

If the formula is none of the above, say, a variable, it is unchanged: $cr(f,v) = f$.

For example, the following specification states one instance of the rule that if `Reset` changes, `Block` does not change.

```
SPEC AG(Reset = Off & Block = On ->
          AX(Reset = On  -> Block = On))
```

Rewriting to stop checking at the end of the test yields the following clause.

```
SPEC AG(Sustain -> (Reset = Off & Block = On ->
          AX(Sustain -> Reset = On  -> Block = On)))
```

Correctness of Rewrite Rules

We will now argue that when $Sustain$ is $false$, every constraint evaluates to $true$. This is a proof sketch; see [3] for a more formal proof. Note that by definition, once $Sustain$ becomes $false$, it remains $false$. We first argue the rules for rewriting universally quantified expressions to be $true$ are correct. The AG rule describes an invariant on every state along all paths; clearly this rule exempts a particular state if $Sustain$ is $false$ in that state. The AF rule describes a property of some future state along all paths; if $Sustain$ is $false$, it is $false$ in all future states, and therefore the property is $true$ for all future states. The AX rule is a special case of the AF rule where the future state is simply the next state.

Finally, the meaning of $\mathrm{A}g\mathrm{U}f$ requires f to become true eventually and for g to hold until it does. If $Sustain$ is $false$, the second rewritten predicate $s\ {->}\ f$ holds thus satisfying that the second predicate becomes $true$ eventually. Also when *Sustain* is $false$, the second predicate is *true* immediately, which satisfies the "until" condition, too.

Next we argue that the rules for rewriting boolean expressions are correct. If the formula is a conjunction or disjunction, both subexpressions become *true* when *Sustain* is $false$, thus the expression is *true*. If the formula is a logical negation, we rewrite the subexpression to be $false$ when *Sustain* is $false$, thus the expression is *true*. Similarly for implication, we rewrite left hand subexpression to be $false$ when *Sustain* is $false$ and the right hand subexpression to be *true* when *Sustain* is $false$. Equivalence is rewritten as two implications.

We now argue the case when we to rewrite the expression to be $false$. The rewrites for AG, AF, and A ... U force the value to be $false$ when *Sustain* is $false$. For AX we must also use the definition of CTL structures that every state has at least one outgoing transition. Otherwise if some state had no next state, AX False would be vacuously *true*.

Finally, the rules for rewriting existential quantifiers to be *true* or $false$ are similar. They can be derived from the rules for universal quantifiers. For instance,

$$\begin{array}{rcll}
cr(\texttt{EF}\ \mathrm{f}, True) & = & cr(!\ \texttt{AG}\ !\mathrm{f}, True) & \text{DeMorgan's law} \\
& = & !\ cr(\texttt{AG}\ !\mathrm{f}, False) & \text{negation rewrite} \\
& = & !\ \texttt{AG}\ Sustain\ \&\ cr(!\mathrm{f}, False) & \text{AG, } \texttt{False} \text{ rewrite} \\
& = & \texttt{EF}\ Sustain\ {-}{>}\ !cr(!\mathrm{f}, False) & \text{DeMorgan's law} \\
& = & \texttt{EF}\ Sustain\ {-}{>}\ cr(!!\mathrm{f}, True) & \text{negation rewrite} \\
& = & \texttt{EF}\ Sustain\ {-}{>}\ cr(\mathrm{f}, True) &
\end{array}$$

Note that rewriting EX expressions to be *true* also uses the definition that every state has a next state. If some state had no next state, EX True would be $false$ for that state, since there exists no next state at all.

Using these rewrite rules to stop checking requirements at the end of a test, we can turn test cases into state machines and soundly compare the requirements against them.

2.1.2 Implementing a Metric on Specifications

The specification coverage metric for a test set, T, over a specification, s, is conceptually simple. It is similar to the test data adequacy of Wu, *et al.* [45], but must be applied to specifications, not programs. In this section we give a formal definition and explain ways of computing coverage more accurately.

Let $\mathcal{C}$ be a criterion and method for creating a set of test requirements, N be the number of requirements created from the specification, and k be the number of requirements satisfied by the test set. The score, S, is

$$S(\mathcal{C}, s, T) = \frac{k}{N} \tag{1}$$

We usually express the score as a percentage. The lowest, or worst, score is 0% when no test requirements are satisfied. The highest, or best possible, score is 100% when all requirements are satisfied.

Winnowing

Some methods to create requirements may produce useless or duplicate requirements. For instance, a path coverage criterion may create a requirement to use a particular transition several times, once for each path that includes it, or it may require the use of an infeasible transition. A mutation adequacy criteria may create a requirement that is vacuously true. To increase the precision and accuracy of some methods, the metric includes a step to discard redundant requirements. We call this step *winnowing*. There are three distinct types of requirements that one may wish to discard.

1. Impossible requirements, i.e., positive requirements that are infeasible or negative requirements that are consistent,

2. Trivial requirements, i.e., positive requirements that are *true* or negative requirements that are $false$, and

3. Duplicate requirements, i.e., different instances of semantically identical requirements.

Each type has a different effect on the score and needs a different approach to eliminate.

The first type of requirement to winnow are those that are impossible to satisfy. For example, a branch coverage criterion may require that each guard of a clause is exercised. The default case when the `Pressure` is `TooLow` has a disjunction of conditions that would yield the following two positive requirements.

```
SPEC EF(Pressure=TooLow & WaterPres >= Low)
SPEC EF(Pressure=TooLow -> EX(!WaterPres >= Low))
```

The first requirement is infeasible, that is, it is impossible to have `Pressure` `TooLow` and `WaterPres` greater than or equal to `Low`. Since this requirement is always inconsistent, it is impossible to satisfy with any test. Impossible requirements prevent any test set from scoring 100%.

The next type of requirement to winnow is trivial requirements, that is, positive requirements that evaluate to *true* in all conditions or negative requirements that evaluate to $false$ in all conditions. For instance, a "replace_vars" mutation operator might replace a variable, `p`, with another variable, `q`, that is the semantic negation of `p`. The resulting negative requirement, `AG (p <-> q)`, is always $false$ or inconsistent, and therefore is satisfied by any test. Counting them would inflate the score.

Trivial requirements may be detected by comparing the negations of all requirements with a totally unconstrained state machine specification in a single run of the model checker. Suppose a negative requirement `P` is always false. The negation `!P` is always true. Thus when `AG (!P)` is consistent, `P` is false and trivially satisfied by any test.

Finally we may winnow semantic duplicates, that is, requirements that evaluate the same for all possible tests. For instance, suppose the mutation operators "negate_expr" and "replace_oper" are applied to `AG (P<Q)` and produce `AG (! P<Q)` and `AG (P>=Q)` respectively. These are exactly the same: any test either kills both or neither. Leaving duplicate requirements, instead of removing all but one copy, adds more weight to the duplicates and discretizes the score.

Duplicate requirements may be detected by essentially comparing every requirement against every other requirement. For instance, if we have requirements `AG r1`, `AG r2` and `AG r3`, check the following.

```
AG (r1 <-> r2)
AG (r1 <-> r3)
AG (r2 <-> r3)
```

Only duplicates will be consistent. In practice, we can reduce the number of comparisons by running a few tests to quickly determine requirements that are not duplicates, then comparing possibly-duplicate requirements with each other.

2.1.3 Minimizing Test Sets

We note that one can use the measurement in Equation 1 to minimize test sets. One can examine which requirements are satisfied by which tests and select a subset of tests that yield a satisfactory score. Although choosing a minimal subset is NP-complete, some test sets may be drastically reduced with little or no loss of coverage. In the flight guidance example discussed in Section 4.2.2, minimization reduces the number of tests from 311 to 166. Although full coverage is maintained, the number of test cases is cut by nearly half.

2.2 Generating Tests

In addition to measuring coverage, model checkers can also be used to generate tests that satisfy various requirements derived from formal specifications, as demonstrated in [14, 5, 23, 25]. Figure 3 shows the overall approach. Not coincidentally, it is similar to the specification-based test adequacy measurement in Figure 2. As with adequacy measurement, one begins with a system specification suitable for a model checker. After that, all processing can be automatic.

Automatically generating the inputs for tests, even in sophisticated and constrained combinations, is usually straight-forward. However coming up with an oracle, that is, the part of the test system that validates the results, is often labor intensive. The oracle might be a reference implementation, expected results that are captured or derived offline and read from a file, a special checking program, or something else. However if software cannot be economically judged as to whether it passes a test, the test is of little value. To be clear about this point, remember from Section 1.3 that a test case includes both a set of inputs or stimuli and the expected result or response. We also use the term *complete*

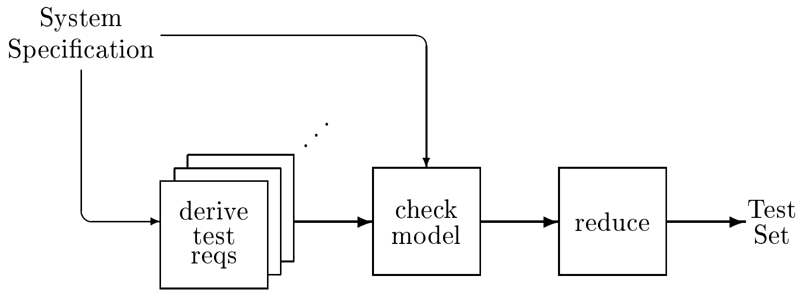


Figure 3: Automatic Test Generation

test case to emphasize that it includes inputs and results. The above approach uses formal specifications to automatically generate complete test cases.

The approach begins with the user selecting some test criterion, such as mutation adequacy [4], CCC partitions [14], MC/DC [16], automata theoretic [17], branch coverage [25], transition pair coverage [41], disconnection or redirection faults [26], stuck-at faults [1], etc. To generate tests, the criterion must produce negative requirements, that is, requirements that are satisfied when found to be inconsistent, as explained in Section 2.1. The model checker checks the requirements against the system specification. If the requirement can be satisfied, it is found to be inconsistent, and the model checker generates a counterexample. The counterexample is an exact description of an example execution, or trace, of the system specification that demonstrates the inconsistency. That is, the counterexample is a test that satisfies the requirement.

Since counterexamples have both stimuli and expected values, they have enough information to be automatically converted into complete test cases. No separate oracle or checker is needed for the output.

2.2.1 Turning Counterexamples into a Test Set

Since a single counterexample may satisfy more than one requirement, we have the opportunity to reduce the set. In fact, the original set of counterexamples may be reduced by an order of magnitude for some test criteria, such as mutation adequacy. The set of counterexamples may be reduced by processes as simple as eliminating syntactic duplicates and removing counterexamples that are "prefixes" of other, longer counterexamples, or as computationally intensive as searching for a minimal set.

A program scans the model checker output for counterexamples, and extracts any that are found, writing one trace for each counterexample. Figure 4 is part of a typical SMV output, edited for brevity. Note that values that do not change from the previous state are not given. For example, SafetyInjection is `On` in state 2.1, is not shown in state 2.2, so it is still `On`, then is shown to change to `Off` in state 2.3.

```
-- specification AG (Pressure = TooLow -> AX (!Press... is false
-- as demonstrated by the following ...
state 2.1:
SafetyInjection = On
Reset = On
Overridden = 0
Block = Off
WaterPres = 2
Pressure = TooLow

state 2.2:
Reset = Off

state 2.3:
SafetyInjection = Off
Overridden = 1
Block = On
```

Figure 4: Typical counterexample

The model checker may produce the same counterexample several times, so duplicate traces are automatically combined. To further reduce the test set, any trace that is a prefix of another trace is automatically discarded since it is unnecessary. Suppose there is a trace $\mathcal{A}$ that is a prefix of another, longer trace $\mathcal{B}$. Any positive requirement on states visited[3] that is satisfied by $\mathcal{A}$ is also satisfied by $\mathcal{B}$. Any positive requirement on transitions between states that is satisfied by $\mathcal{A}$ is also satisfied by $\mathcal{B}$. What if it requires the trace to stop at the end of $\mathcal{A}$? Since in SMV every state must have a next state, no trace can actually end—just the counterexample leading to an indicated state ends. In fact, there is no way to specify such a requirement in CTL.

As pointed out in Section 2.1.2, the test set may be further reduced by selecting a subset with the same, or sufficient, coverage. For large specifications the decision of how much effort should be devoted to minimizing the number of test cases depends on how much it costs to reduce the test set compared with the cost of testing, that is the product of the cost of running the test set and the number of times the test set will be run.

A test case corresponding to the trace assigns initial values to variables as described in the first state and then changes values for the inputs, or independent variables, such as `Block`, `Reset`, and `Overridden`, as specified in subsequent states. Correct operation is verified by checking if the values of dependent variables, $SafetyInjection$ in this case, agree or disagree with the those computed by the implementation.

[3] If we require that a state is *not* visited, a shorter sequence, $\mathcal{A}$ in this case, may satisfy the requirement while a longer sequence does not. Thus the restriction to positive requirements.

It is worth noting that system level tests can only rely on explicit inputs and outputs visible at the system level. Mode or internal variables, such as `Pressure` for this example, may not be visible or may be implicit in the implementation even though they are explicit in the specification. As a result, the utility of tests depends, in part, on the visibility of variables in the particular implementation under test.

3 Test Criteria

The previous section showed how to use a model checker either to evaluate an existing test set against a given test criterion or to generate a test set that satisfied the test criterion. Specific criteria were not addressed, nor were details given on how to encode the test requirements for a particular criterion into temporal logic formulae suitable for a model checker. In this section, we turn our attention to specific criteria and the associated encodings in temporal formulae. First, we develop a specification-based mutation test criterion. Since this contribution is novel to this paper, this section is relatively formal and detailed. Then we turn to other specification-based criteria, specifically uncorrelated full predicate coverage, which is related to MC/DC coverage [16], transition pair coverage [41], and branch coverage [25].

Briefly, a mutation coverage criterion requires tests to distinguish between the original specification and a slightly different one (a "mutation"). Uncorrelated full predicate coverage requires tests to exercise different behaviors when each variable in every boolean expression is true or false. One step up from requiring that every transition be exercised, transition pair coverage requires the execution for every state of every combination of transitions into and out of that state. Finally, branch coverage requires that every case in SMV `next` statements is exercised.

3.1 Specification-Based Mutation Coverage

Traditional program mutation analysis [21] is a code-based method for developing a test set that is sensitive to small syntactic changes to the structure of a program. Following is a theoretical foundation for mutation analysis of state machine descriptions such as might be found in SMV or SPIN.

We are interested in generating and evaluating test sets for mutation adequacy with respect to a state machine description SM. That is, we would like a set of tests that distinguish the behavior of the state machine Σ as described by SM from the behavior of each machine Σ' described by SM', where SM' is a minor syntactic variation of SM. For reasons of scalability, we choose SM instead of Σ as the basis for mutation analysis. Specifically, Σ is usually subject to state explosion problems, but SM typically is not. Why? Each additional variable in the description SM adds a few lines of text, while an additional variable in the state machine Σ roughly multiplies the number of states by the number of values of the variable. Hence, for n variables, the size of a description is typically $c_d n$ while the size of a state machine is c_s^n.

The choice of SM over Σ is also consistent with program-based mutation analysis, where the foundation of inquiry is the program source code as opposed to the underlying input-output relation. Another way of thinking about this decision is that mutation analysis is traditionally a syntactic, rather than a semantic, approach to testing, and this work continues that tradition.

Possible mutations to a description SM are defined by a set of mutation operators, Op. As an example, a mutation operator $op \in Op$ might replace an occurrence of a variable with another of compatible type. A set of mutant descriptions M is produced by systematically applying each mutant operator $op \in Op$ to SM. Each mutant $SM' \in M$ is the result of applying one mutation operator one time to SM. In other words, we only consider "first-order" mutations in this treatment [38]. Extensions beyond first-order mutations are straight forward, but seem unlikely to be useful.

The state machine Σ' defined by a mutated description SM' generates its own set of traces γ'. Adopting the terminology of program-based mutation analysis, we say that the mutant SM' is *equivalent* (to SM) if $\gamma' = \gamma$. An equivalent mutant cannot lead to a test case. For mutants that are not equivalent, the way in which γ' differs from γ is important. We say that a mutant SM' can be *killed* iff there is a trace t in γ that is not in γ'. That is, t kills SM' if Σ allows t but Σ' does not.

The reverse is also possible. We say that a mutant SM' can be *excluded* iff there is a trace t in γ' that is not in γ. That is, t excludes SM' if Σ' allows t but Σ does not. We ignore mutants that can be excluded, but not killed.[4] The reason is that a correct implementation cannot execute such a trace t. The vast majority of the testing literature focuses on test cases that are feasible rather than test cases that must be rejected. Nonetheless, there are specialized situations where such traces are useful for testing, such as safety testing, where one might wish to systematically attempt dangerous behavior to increase the confidence that the implementation correctly rejects the dangerous actions.

The distinction between killing and excluding mutants can be summarized as follows. A mutant is killed by showing that it fails to provide some required behavior. A mutant is excluded by showing that it provides some disallowed behavior. The Venn diagrams in Figure 5 illustrate the possibilities. Figure 5a illustrates an equivalent mutant, that, by definition, can be neither killed nor excluded. Figure 5b illustrates a mutant that can be killed but not excluded. Figure 5c illustrates a mutant that can be excluded but not killed. Finally, Figures 5d and 5e illustrate mutants that can be both killed and excluded.

We are now in a position to define the mutation testing criterion.

> A test set T is mutation adequate with respect to mutation operators Op and description SM iff for each killable mutated description SM' in M, the set of mutated state machine descriptions generated by applying Op to SM, there is a trace $t \in T$ that kills SM'.

[4] The resulting tests are called "negative tests" in [23] and "failing tests" in [5] .

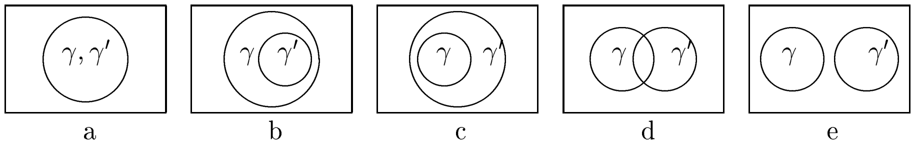


Figure 5: Venn diagrams for killable vs. excludable mutations

Next, we wish to implement mutation analysis with a model checker.[5] To do this, we implement each mutation operator with an equivalent construction in the temporal logic. Specifically, to implement each mutation SM' we construct a temporal logic formula f so that f is inconsistent with respect to Σ iff SM' can be killed. Specifically, f is constructed so that each trace $t \in \gamma - \gamma'$ is a counterexample that shows f to be inconsistent with Σ. If $\gamma - \gamma' = \emptyset$, as occurs in Figures 5a and 5c, the formula f is simply consistent with Σ.

In procedural terms, the following occurs. We elaborate on these steps in subsequent sections.

1. The description SM is expounded to make implicit parts of the description explicit.

2. A set of mutation operators Op is selected.

3. For each operator $op \in Op$, an equivalent operator op_t that produces the required temporal logic formulation is derived. In general, a proof is required to show that the derived operator op_t does in fact implement op.

4. Construct the set M_t of logic formulae produced by the various op_t applied to SM.

5. The logic formulae can be used for either test case evaluation or test case generation as described in Section 2.

We discuss the first three steps in detail below. Step 4 is mechanical; tool support for this aspect is mentioned later in the paper.

3.1.1 Expoundment

Before applying mutation operators to a description SM, there are two reasons to modify SM through a process we call *expoundment*. Expoundment essentially makes implicit parts of a specification explicit. Making implicit parts of a specification explicit is a standard aspect of many formal methods. It is

[5] In the original descriptions of this work [4, 5], the mutation model was defined in the context of both a state machine description and a set of temporal logic constraints. The approach taken here of using the model checker merely as a tool is considerably simpler. It also avoids certain unpleasantries such as inconsistent temporal logic clauses that nonetheless lack counterexamples.

clear from prior work that explicit specifications are definitely desirable from the perspective of test coverage [5].

The first reason for expoundment is to make more of the description available for mutation analysis. For example, in SMV, the default branch on a `case` statement offers little for a mutation analyzer to manipulate. If instead, the default is replaced with an explicit predicate detailing the conditions under which the default is taken, the mutation analyzer has a rich syntactic structure to manipulate. An example of a concrete benefit of expoundment is that mutation analysis without expoundment is unlikely to satisfy branch coverage [25] for a given application, but after expoundment, there are various mutation operators that imply branch coverage.

The second reason for expoundment is to ease the subsequent process of implementing a mutation with a temporal logic formula. For example, to implement mutation operators to the various branches in `case` structures, it is convenient if the guards on the branches form a partition and if the outcomes of each branch are pairwise disjoint. We discuss this aspect in section 3.1.3.

3.1.2 Mutation Operators

In this section, we illustrate six mutation operators on a description SM. We do so in the context of SMV descriptions. We explicitly do *not* claim to define a canonical set of mutation operators. Rather, we argue that the precise set Op suitable varies from one application to another. Below, we give a possible set.[6]

Consider the following SMV state machine description fragment from a `case` construct in a `next` structure. We use this fragment to illustrate six mutation operators defined below.

```
next(Overridden) := case
   ...
   Pressure=TooLow & Reset=Off & next(Reset)=On : 0
   ...
esac;
```

1. Const - replace one constant with another, e.g.,

   ```
   Pressure=High & Reset=Off & next(Reset)=On :  0
   ```

2. Negate - negate a boolean expression, e.g.,

   ```
   !(Pressure=TooLow) & Reset=Off & next(Reset)=On:  0
   ```

3. Operat - replace one boolean operator with another, e.g., replace “and” with “or”

   ```
   Pressure=TooLow | Reset=Off & next(Reset)=On :  0
   ```

[6] In Section 3.2 we argue that the stuck-at mutation operator is of special merit.

4. Vars - replace a variable with another variable, e.g.,

   `Pressure=TooLow & Block=Off & next(Reset)=On :  0`

5. Stuck-at - replace a simple condition with 1 (true) or 0 (false),

   `Pressure=TooLow & 0 & next(Reset)=On :  0`

6. Remove - remove a simple expression from conjunctions, disjunctions, and implications, e.g.,

   `Pressure=TooLow __ & next(Reset)=On :  0`

We emphasize that the preceding is one possible list of operators, but not a definitive one. Even for program-based mutation analysis, there is no theoretical characterization of an "ideal" set of mutation operators, although empirical results show a few operators to be quite powerful [11, 39, 40], and theoretical considerations [12, 32] prove that some operators subsume others. That is, any test set that kills all mutants of a subsuming operator also kills all mutants of the subsumed operator, while the opposite is not true. Thus if we use a subsuming operator, we need not use subsumed operators.

3.1.3 Reflection

The goal of reflection is to express the state machine in temporal logic. These temporal logic formulae can be manipulated to implement a test criterion to be analyzed by a model checker.

Consider the SMV `case` structure constraining possible `next` values of a variable, which is the context of the six mutation operators defined in section 3.1.2.

```
next (x) := case
   b1 : v1;
   b2 : v2;
   ...
   bN  : vN;
esac;
```

These semantics are typical of a programming language case statement. The guard `b1` is evaluated; if it is true, `v1` is the next value for `x`. The right hand side, `v1` may be a set, thereby allowing for nondeterminism. If `b1` is false, `b2` is evaluated. The case `bN` often is a default, which is `1` or true in SMV.

Suppose that expoundment has recast the `bi` to be a partition; that is the `bi` are pairwise disjoint and their union is universally true. Further, assume that the `vi` are pairwise disjoint. That is, if two guards have the same value for a target, the guards are joined into one guard.[7]

[7] We also assume that the `case` statement is "flat," that is, that nested `case` statements have been collapsed into one. This is not strictly necessary, but eases the exposition.

Theorem: If the `bi` are a partition and the `vi` are pairwise disjoint, any mutation to a guarded command, `bi : vi`, in a `next` construct for variable x can be implemented with the same mutation applied to the temporal logic formula:

```
SPEC AG(bi -> AX(x = vi) & !bi -> AX(!(x = vi))
```

(Note: If `vi` is a set, we write: `x in vi` instead of `x = vi`.)
Specifically, if the mutation is to `bi` to form `bi'`, then we get:

```
SPEC AG(bi' -> AX(x = vi) & !bi' -> AX(!(x = vi))
```

Similarly, if the mutation is to `vi` to form `vi'`, then we get:

```
SPEC AG(bi -> AX(x = vi') & !bi -> AX(!(x = vi'))
```

In either case, we define the traces of interest T to be those that include the state evaluated in the context of the `X` operator. A trace that ends in the prior state, while technically a counterexample in CTL, is not sufficient to kill the mutant.

Proof: To show this construct is correct, it is necessary to show that a trace t is in the set $\gamma - \gamma'$ iff t is in the set T.

First, assume t is in $\gamma - \gamma'$. Since the only difference between SM and SM' is in the mutation to the guarded command `bi : vi`, it must be the case that either predicate `bi` evaluates differently or `vi` differs. If the mutation is `bi'`, then there must be a state in t where `bi` and `bi'` evaluate differently, or else $\gamma - \gamma'$ is empty. In the subsequent state, if `bi` is true, x has the value `vi` in t, and if `bi` is false, x has some value not equal to `vi` in t. In either case, due to the assumption that the `bi` are a partition and the `vi` are disjoint, machine SM' assigns the wrong value (or possibly an undefined value) to x, and this is captured exactly by the CTL formula. If the mutation is `vi'`, then x simply has the wrong value for some state in t. The assumptions again prevent x from acquiring the correct value by coincidence.

Now, assume t is in T. Then, there must be the same transition in t where, depending on the mutation, either the source state has `bi` evaluate differently from `bi'` or the target state has $x =$ `vi` instead of $x =$ `vi'`. Due to the partition assumption on `bi` and the disjointness assumption on `vi`, t must necessarily be in $\gamma - \gamma'$.

Corollary Given the conditions in Theorem 1, the SPEC clause

```
SPEC AG(bi <-> bi')
```

is a satisfactory implementation for mutations to `bi` if each trace t in T is defined to include at least one additional state beyond the counterexample to the CTL formula. Further, assuming there is at least one mutation operator that applies to each `bi`, then tests that kill these mutations will also kill any mutation to `vi`. Thus, we prove that mutation operators that apply to the `vi` are redundant under the partition of the `bi` and disjointness of the `vi` assumptions.

Proof: It might seem curious that it is not necessary to mention `vi` and `vi'` to kill mutations `vi'`, but the partition property of the `bi` and the disjointness of the `vi` ensure that if the target `vi'` is different, any trace t that includes a state where `x` is set to `vi` will surely kill SM'. Such a trace is necessarily provided if, in the previous state, `bi` and `bi'` differ for any mutation `bi'`.

3.2 Uncorrelated Full Predicate Coverage

We define a criterion, Uncorrelated Full Predicate Coverage, that is closely related to the popular multiple condition/decision coverage (MC/DC) criterion. We then explain why MC/DC is difficult to implement with a model checker and show how to implement the new criterion with a model checker. Chilenski and Miller define MC/DC as follows [16]:

> 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. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.

MC/DC coverage has a strong reputation; typically it is part of the justification for FAA certification of critical civil avionics software under DOD-STD-178B. One reason for this reputation may be that there is a correspondence between MC/DC coverage of source code and branch coverage of the underlying (compiled) object code. For the purposes of this paper, it is of interest to see whether we can implement something like MC/DC at the specification level with a model checker. Offutt provides a possible definition with *full predicate* coverage (FP) in the context of a state transition model where predicates are associated with transitions [41]:

> For each predicate P on each transition, T includes tests that cause each clause c in P to result in a pair of outcomes where the value of P is directly correlated with the value of c.

What is interesting about both of these criteria is that they define test coverage in terms of *pairs* of test cases. In the case of MC/DC, the pair of test cases must be related so that the predicate in question evaluates differently at exactly one condition. In essence, this condition toggles between the two test cases. The full predicate criterion generalizes this so that other conditions may change as well, as long as the predicate in question still has its value determined by the single condition. Explicit care is taken in full predicate coverage—via the "correlation" clause—to ensure that decision coverage is maintained; that is, the predicate in question evaluates to true on one test in each pair and false on the other.

Note that any test set that satisfies MC/DC also satisfies full predicate, but that the converse is not true. For example, consider the predicate $P = a\&(b|c)$.

Triples $t_1 = (false, true, false)$ and $t_2 = (true, false, true)$ are full predicate adequate with respect to variable a[8] but not MC/DC adequate. Triples $t_3 = (false, true, false)$ and $t_4 = (true, true, false)$ are both full predicate adequate and MC/DC adequate with respect to variable a.

A model checker builds test cases as counterexamples, and, in general, there is no way to tie the generation of one counterexample to specific characteristics of other counterexamples that have been previously generated.[9] So, on the face of it, implementing either MC/DC or full predicate coverage with a model checker is not possible.

There are two routes out of this dilemma. One is to exploit specific properties of the algorithm used in a specific tool, such as SMV, to generate counterexamples. This is useful, but dangerous from an engineering perspective, since the algorithm for generating counterexamples may change without warning.

The other route is to define a new criterion, that covers as much of MC/DC (or full predicate coverage) as possible, and leaves the remainder of the criterion explicitly aside. We follow this route here.

The new criterion is full predicate coverage without the section that puts requirements on pairs of test cases. We define *uncorrelated full predicate coverage* (UFP) as a modification of FP follows:

> For each predicate P on each transition, T includes tests for each condition c in P such that c is true when P depends on c and c is false when P depends on c.

Essentially, UFP drops the requirement for decision coverage from FP. That is, it is possible for some predicates P to choose a pair of test cases for some condition a such that a assumes both truth values, but P does not. For example, consider P to be $a <-> b$, that is, a equals b. A test set T of $(a,\ b)$ pairs that is $\{(false,\ true), (true,\ false)\}$ is UFP adequate with respect to a, but is neither FP adequate nor MC/DC adequate, since P evaluates to $false$ in both tests. Note that any test case that satisfies full predicate also satisfies UFP, but that the converse is not true.

To implement the UFP with a model checker, it is convenient to use the boolean derivative [2]. Temporal logic clauses are developed as follows. Consider some predicate P that appears in the specification. Suppose that P is a function of condition a and zero or more other unnamed conditions. To achieve UFP, compute the boolean derivative of P with respect to each condition. For a, this yields dP/da, a predicate in the remaining conditions that is true iff the value of a determines the value of P. For UFP, first we need a to assume the value *true* under conditions where a determines the value of P, that is, under conditions defined by dP/da. We do this with a never-claim that says that a is never true under conditions dP/da. This yields the CTL formula:

```
SPEC AG(dP/da -> !a)
```

[8]Tests to yield adequacy with respect to b and c are omitted for clarity

[9]It is possible in SPIN to ask for an entire set of counterexamples to be enumerated, but this capability is not sufficiently powerful for our purposes here.

Given this SPEC clause, the model checker then produces a counterexample with the desired result.

To produce a test case where a is *false* under conditions where a determines the value of P, we write the never-claim:

```
SPEC AG(dP/da -> a)
```

An additional pair of CTL formulae is required for each of the remaining conditions in P. Thus a predicate P in n conditions yields $2n$ CTL formulae. Curiously, even if all of the never-claims are inconsistent, only $n+1$ counterexamples are required for these $2n$ formulae [16].

3.2.1 Mutation Analysis and UFP

There is an interesting tie between the stuck-at mutation operator and UFP:

Theorem: If a test set T is mutation adequate with respect to the stuck-at mutation operator, T is also UFP adequate. Further, if no variable occurs more than once in a given predicate, T is mutation adequate for stuck-at iff T is UFP adequate.

Proof: Suppose for the moment that P has exactly one occurrence of some variable a. Let Pt be P with a replaced by *true* and Pf be P with a replaced by *false*. Consider the two positive test requirements (1) and (2) generated by the stuck-at operator and the two corresponding test requirements (3) and (4) for UFP:

```
(1) EF !(P <-> Pt)
(2) EF !(P <-> Pf)
(3) EF !(dP/da -> a)
(4) EF !(dP/da -> !a)
```

The theorem amounts to showing first that any trace that satisfies (1) also satisfies (3) and second that any trace that satisfies (2) also satisfies (4). The arguments for these two cases are the same, so we only consider the first.

Suppose that trace t satisfies (1). It must be the case that a is *false* in the last state of this trace; otherwise P and Pt would necessarily have the same value. For the same reason, the other variables in P must have values such that P depends on a. This implies that dP/da must be true, and so (3) must also be satisfied by trace t. This shows the first part of the theorem.

Under the assumption that P has one occurrence of a, the reverse argument also holds for exactly the same reasons, and therefore if (3) is satisfied then so must be (1). However, a may occur multiple times in P. The stuck-at mutation operator changes exactly one occurrence of a at a time, whereas UFP considers all occurrences of a at once. Put another way, mutation analysis generates a pair of test requirements for each occurrence of a, but UFP generates exactly just one pair of test requirements for a no matter how may times a appears. Consequently although a trace generated from UFP is guaranteed to satisfy at least one test requirement for the stuck-at operator, there is no guarantee that the trace will satisfy all of them.

For example, consider the predicate $P = a\&b|a\&c$. Note that $dP/da = b|c$. The six test requirements, 4 for stuck-at and 2 for UFP are:

```
(1) EF !(P <-> b | a & c)      -- the first "a" set to true
(2) EF !(P <-> a & c)          -- the first "a" set to false
(3) EF !(P <-> a & b | c)      -- the second "a" set to true
(4) EF !(P <-> a & b)          -- the second "a" set to false
(5) EF !((b | c) -> a)
(6) EF !((b | c) -> !a)
```

Test requirements (5) and (6) can be satisfied with the (a, b, c) triples $t_1 = (false, true, false)$ and $t_2 = (true, true, false)$, respectively. Triple t_1 satisfies test requirement (1); triple t_2 satisfies test requirement (2). Neither t_1 nor t_2 satisfy test requirements (3) or (4).

3.3 Transition Pair Coverage

Offutt defines transition pair testing over a graph G as follows [41]:

> For each pair of adjacent transitions $S_i : S_j$ and $S_j : S_k$ in G, there is a test that traverses the pair of transitions in sequence.

Offutt's definition is developed in the context of SCR specifications, and the graph in question has modes for nodes. The definition can be extended easily to state machine descriptions such as those in SMV. Specifically, consider any variable `x` with possible values `v1` through `vN`. Transition pair coverage with respect to variable `x` simply requires finding tests that start with `x` having each of its possible values in some state, having a specific value `vi` in a successor state, and finally assuming each possible value in a third state. That is, transition pair coverage requires pairs of transitions, one of which reaches a state where `x` equals `vi`, and another where `x` leaves that state. Note that the values of all other variables have been abstracted away, so that the graphs being covered are all of a size determined by domain of the variable under analysis. For this reason, transition pair coverage scales linearly with the size of the specification. Different variants of the criterion depend on whether `x` is allowed to remain in the intermediate state for multiple transitions, and also on the handling of "self-loops," that is, transitions where variable `x` maps to the same value in both the source and target state.

Transition pair coverage, of whatever variant, can be expressed directly with a set of temporal logic never-claims. Specifically, to ensure that a counterexample covers a particular pair of transitions: `x = vi` to `x = vj` to `x = vk` under the assumption that self-loops are not allowed, one would write the never-claim:

```
SPEC AG(x = vi -> AX(x = vj -> AX (!x = vk)))
```

In English, the never-claim is equivalent to saying, "It is always true that if `x = vi` in some state and `x = vj` in the next state, then `x` does not equal `vk` in any state immediately after that." The counterexample, assuming one exists, delivers the desired test case. If no counterexample exists, the model checker reports the SPEC clause to be consistent, thereby implying that the test requirement is infeasible and can safely be ignored.

In summary, the test requirements for transition pair coverage can be expressed directly with temporal logic never-claims, thereby making a model checker a convenient tool for implementing the transition pair criterion.

3.4 Branch Coverage

Gargantini and Heitmeyer achieve branch coverage for SMV and SPIN descriptions by adding labels to the state machine description at each branch [25]. In SMV, this corresponds to each guarded command in a `next` statement. The CTL formulae are then simply never-claims that the labels are unreachable. For counterexamples, the model checker generates traces that reach each guarded command.

As with program-based mutation analysis, specification-based branch coverage is implied by specification-based mutation analysis provided that at least one nonequivalent mutant is generated for each guarded command in a `next` statement. Generally, expoundment is required to achieve this, or else the default case in the `next` statement may not yield a structure amenable to the mutation operators chosen.

3.5 Disconnection and Redirection Faults

Godskesen [26] generates tests for embedded systems by hypothesizing disconnection and redirection faults. Disconnecting an input can be modeled by a mutation that replaces the input variable with $false$. Redirecting an input can be modeled by replacing one input variable with the input to which it is redirected. An output disconnection is equivalent to many large specification faults at once, so are difficult to model with simple mutation operators.

4 Practical applications and examples

In this section we document some of the tools available and give examples.

4.1 Tools

Theoretical approaches without implementations are rarely helpful to practitioners. In fact, there is a significant opportunity for tools that do even part of a task with near-total automation compared with tools that do all of a task, but require a lot of user expertise or time [42].

4.1.1 Deriving Mutation Requirements

The mutation engine is based on the SMV [35, 13] parser and abstract syntax tree manipulation routines. Its overall design is to

1. parse an SMV file and build a corresponding data structure in memory with the state machine descriptions and specifications annotated with pointers to type information,
2. write the unmodified SMV information, and
3. traverse the syntax trees one or more times invoking specific, hard-coded pattern matching routines that recognize an opportunity for a mutation, make a mutant, and write the mutant.

Currently the mutation engine implements six mutation operators similar to those explained in Section 3.1.2: replace enumeration constant, replace defined constant, negate expression, negate only simple expression, replace operator, replace variable, and remove simple expression.

If one were to adhere strictly to mutation theory, one would write one entire SMV file for each separate mutation, yielding thousands of files. However, the overhead of starting SMV, parsing the file, and checking all the specifications that did not change is enormous. Since specifications are checked independently, computing counterexamples is unchanged if we combine hundreds, if not all, of the mutations in one SMV file. Combining the mutations amortizes the overhead to a negligible cost at a slight increase in memory use.

4.1.2 Generating Test Cases

A single program, mugent, automatically generates tests, as diagrammed in Figure 3 and explained in Section 2.2. It runs the mutation engine, to derive mutations as test requirements, runs the model checker, to generate counterexamples, then extracts succinct test cases and reduces them. For now the only reduction is keeping just one copy of syntactically duplicate test cases and discarding any test case that is a prefix of another, longer case.

4.1.3 Turning Test Cases into Executable Tests

The final stage of automatic test generation is to turn each test case, with modeled variable references, into executable test code, including a test harness, drivers, and checking and reporting code. We use TAO (Test Assistant for Objects) [34], a software interface testing environment developed at NIST. TAO integrates two popular test specification technologies, context free grammars and constraint satisfaction, into a single software test generation tool.

4.1.4 Evaluating Test Sets

We use two tools to evaluate test sets. The first is "alltests," which symbolically executes a set of test cases, as described in Section 2.1.1, against a list of requirements. Negative requirements that are found to be inconsistent are satisfied by the test, while positive requirements are satisfied if they are consistent. For example, in mutation analysis the mutants serve as negative requirements which are killed, or satisfied, if they are inconsistent. To reduce the disk use of saving counterexamples from thousands of requirements over hundreds of tests, the program can write just a `T` or `F` depending on whether the requirement is consistent (true) or inconsistent (false).

The second tool is "repcov," which reports the coverage of a test set by listing each requirement and how many tests satisfy it. It also reports the coverage as a ratio between the number of satisfied requirement and the total number of (satisfiable) requirements.

4.2 Examples

Our primary example throughout the paper has been a piece of the specification of the safety system to inject water into a reactor. We also note our work on other models: an automobile cruise control, a Java[10] virtual machine stack, part of a secure operating system, and a flight guidance system.

4.2.1 Safety Injection

To illustrate, we apply our method to the Safety Injection problem. See the Appendix for an SMV specification or Table 1 for a higher-level specification.

The results are shown in Table 2. The top row of the table shows, for each type of mutant, the total number of test requirements generated (in parenthesis), and then the number of feasible test requirements. The second row enumerates the test cases, that is, the unique counterexamples, generated by the model checker. The first number is before minimization; the second number is after minimization. All evaluations of one mutation operator against another are done with respect to the minimized test sets. The remaining rows show these cross comparisons. An operator's row gives the mutation scores for that operator against the test requirements generated by the mutation operators in the various columns. For example, the entry in the third row, labeled `Const`, and second column is `275/276` and `99%`. This means that the eleven tests generated by `Const` satisfied 275 of 276 test requirements generated by `Negate`. The mutation operators are in the same order as Section 3.1.2. The entries on the diagonal are omitted for clarity; these all show complete coverage.

4.2.2 Other Examples

Earlier versions of the method described in this paper have been applied to a variety of applications. The primary reason to mention these applications here

[10] Java is a trademark of Sun Microsystems, Inc. Certain commercial equipment, designs, or software are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the software or equipment identified are necessarily the best available for the purpose.

	Const (168)106	Negate (343)276	Operat (477)270	Vars (296)105	Stuck-at (348)124	Remove (158)53	Total (1790)934
Tests	(20)11	(22)11	(26)14	(17)10	(22)11	(19)10	(28)16
Const	-	275/276 99%	253/270 93%	95/105 90%	123/124 99%	52/53 98%	904/934 96%
Negate	106/106 100%	-	254/270 94%	97/105 92%	124/124 100%	53/53 100%	910/934 97%
Operat	106/106 100%	276/276 100%	-	97/105 92%	124/124 100%	53/53 100%	926/934 99%
Vars	97/106 91%	260/276 94%	234/270 86%	-	112/124 90%	45/53 84%	853/934 91%
Stuck-at	106/106 100%	276/276 100%	254/270 94%	97/105 92%	-	53/53 100%	910/934 97%
Remove	106/106 100%	270/276 97%	250/270 92%	89/105 84%	120/124 96%	-	883/934 94%

Table 2: Comparison of Mutation Operators on Safety Injection

is to bolster the case for the feasibility of our technique. Particularly, the flight guidance example shows that reasonably large examples can be addressed with our technique.

1. Cruise Control: Cruise Control [31] was our first example. Many variations of it exist; we use one from Atlee and Bucklee [6], which was originally coded in Software Cost Reduction (SCR) [29]. Results are reported in [5]. Briefly, mutation analysis generates 24 test cases. We wrote a draft implementation of it so that we could examine the code coverage. The 24 test cases cover 14 of 16 branches. One missed branch was infeasible. The other missed branch was a default case and not explicitly specified. This pointed out the need to expound the specification, as explained in Section 3.1.3. Having expounded the implicit branches, test cases were generated to test the missed branch. The tests did reveal that the draft implementation was incorrectly written level-triggered, while the specification was edge-triggered.

2. Java Stack: In the hardware realm, we applied this technique to a model of the operand stack of a Java virtual machine. We abstracted it to get a model expressible in SMV. The model is a stack with zero, one, two, three, or "many" items plus a variable that prevents counterexamples if the stack size exceeds three, since traces in the model might not apply to the machine. Instructions were abstracted to those that push one item (`push1`), pop one item (`pop1`), or pop two items (`pop2`). In addition to

reflecting the state machine, it includes "use case" specifications such as "`push1` then `pop1` leaves the stack unchanged."

Combining duplicates and discarding prefixes yielded nine unique tests with lengths from three to seven operations. By comparison, exhaustive enumeration yields 45 tests of length seven.

3. Secure Operating System: To generate tests, we modeled part of a Unix-like secure operating system. The simple model consists of two processes and one file. Files have an owner, a group, and a numerical security level. Files also have permission bits to allow a file to be read or written by the owner, anyone in the group, and others. Processes have a user and group, and may have privileges to raise or lower the security level of a file. We modeled the system calls of opening a file, changing the permission (`chmod`), and changing the security level. We specified some "use cases" in addition to reflecting the state machine. A typical use case is if a process can read a file, and any process tries to lower the file's security level, the process can still read the file, regardless of whether the downgrade succeeded.

   Mutation analysis generated 3172 mutants, of which 2845 were inconsistent. We started with a restricted test frame, one process using only the system call to change the security level, and enlarged it in steps to pick the simplest test cases to kill the mutants. After we reduced the test set with "mugent" (Section 4.1.2), we ended up with 193 tests.

4. Flight Guidance: In an effort to see how the mutation approach scales to a larger example, the mutation method in [4] was applied to a portion of a model flight guidance system developed by Rockwell Collins with support from NASA Langley for the express purpose of academic investigation on a realistic example [37]. The portion evaluated was the subset of the model required to analyze one invariant for the system. The invariant states that the autopilot must be engaged when the "reset" signal is received, and that the "flight director" must be on when the autopilot is engaged. This ensures that the current flight guidance modes are displayed to the flight crew. The resulting SMV model has 39 variables, of which 17 were boolean and 22 were "scalar" (integers with a limited range). Mutation analysis produced 10,186 test requirements, of which 6,938 were satisfiable. The tools produced 311 unique counterexamples, which, after minimization, resulted in a set of 166 tests that satisfied these requirements. The time required to complete this analysis—both the generation of the original 311 traces and the evaluation required for minimization of the test set—was about a half a day, concurrent with normal use, on a Computer Science department's UNIX server, a Sun **SPARC**®[11] 30. Our conclusion is that the technique is feasible for systems of interest.

[11] SPARC is a registered trademark of SPARC International, Inc.

5 Abstraction Issues

Even though model checking is increasingly applicable to software analysis, clever abstractions are required for problems of even modest complexity to avoid the state space explosion problem, which renders the model checker useless. Some of these abstractions are informal, although there have been significant formalizations of the abstraction process [3, 15, 8, 33]. Other abstractions [43] are formalized to the extent of being paired with theorem provers and model checkers to calculate and refine them. These abstractions are directed at the analysis problem, that is, determining whether given properties expressed in a temporal logic hold over a given state machine.

We know of several existing, mechanical approaches to reduction or abstraction. To be useful, abstractions must preserve some properties of the original. Two useful measures are soundness and completeness.

In a sound abstraction, properties of the reduced or abstract specification are also properties of the original specification. Soundness avoids false positives. In a complete abstraction, properties of the original specification are also properties of the reduced specification. Completeness avoids false negatives.

Bharadwaj and Heitmeyer [8, 9] formalize an abstraction, RP1, that removes irrelevant entities. Briefly, to check that some property q holds for a specification, one may remove variables and inputs that do not occur in or contribute to q. They also formalize RP2, that abstracts monitored or input variables. That is, if a monitored variable only influences the value of another "summary" variable, the monitored variable may be removed. For instance, `WaterPres`, with a discrete range from 0 to 2000, is immediately quantized to `TooLow`, `Permitted`, or `High` and may be removed. Both abstractions are sound and complete for analysis.

Chan, et. al. [15] use another method to reduce the state size. Some specifications place time bounds on the intervals between events. The obvious specification keeps time as an integer, uses variables to record the times of events, and has predicates on the difference in times. Instead, they keep (bounded) timers measuring the time since events. When the bound is exceeded, the timers enter a "satisfied" (or "unsatisfied") state.

They also use a temporal strength reduction. Suppose there is a predicate on the value from a previous state. Rather than saving the previous value and computing the predicate, just save the value of the predicate. For instance rather than save the previous value of an integer y then compute $prev(y) \geq 1000$, compute $prev(y \geq 1000)$. The abstracted model only need save a boolean value.

Kurshan [33] explains how k verifications may be done on k reductions of a system, each of which is a $1/k$ part of the entire system. Since verification is often exponential in the size of the system, n, a verification of the entire system at once may be proportional to c^n while k verifications take $kc^{n/k}$ work.

In an overview presentation, Rushby [43] advocates "ubiquitous abstraction," that is, using abstractions in several different ways in all parts of analysis. For instance, even for one given problem, different abstractions may be appropriate, depending on the invariants used to prove a goal. The invariants may be automatically strengthened when the proof fails. Another noteworthy approach is calculating transitions of a state abstraction using rules that guarantee correctness, as opposed to taking a hand-crafted abstraction and proving it is sound and complete. Abstractions may be refined automatically using information from static analysis, such as reachable states. In contrast, we are still at the stage of characterizing abstractions in our work, albeit for a different notion of soundness, rather than computing them.

Bensalem, Lakhnech, and Owre [7] explain a semi-automated abstraction in which the analyst chooses a state abstraction and then a conservative (sound) set of corresponding transitions are computed. Construction begins with a complete set of transitions, that is, a transition from every abstract state to every other abstract state. If a transition can be (automatically) proven to be impossible, it is removed. Since such proofs are in general too complex, they combine it with three techniques based on partitioning the abstract variables, substituting, and using the property being investigated.

Since our goal is automatic test generation, rather than property analysis, we can use different abstractions, as explained in [3]. For analysis, abstractions may summarize states and discard details of transitions. An abstracted model may still be quite useful even if it is not precise. To automatically generate tests, we may wish to retain details in order to easily determine if an implementation behaves properly. We can then accumulate sets of tests generated from different precise reductions. In summary an abstraction that is perfectly satisfactory for one purpose, property, or specification may be unusable in another.

One abstraction for test generation is to map variables with large or unbounded domains to a fixed subset of the possible values. For example, an integer variable x might be modeled with a corresponding variable x_{model} with a bounded range of $0, 1,$ and 2. From the test generation perspective, the ranges simply need to cover values that may be interesting when used in actual test cases. We used this abstraction in the Java Stack example in Section 4.2.2.

6 Conclusions

Testing, particularly system testing, consumes a significant portion of the budget for software development projects. Formal methods, typically used in the specification and analysis phases of software development, offer an opportunity not only to reduce the cost of testing, but to increase confidence in the software through formal criteria for test thoroughness. We showed how to use the powerful computation engines of model checking to the problem of evaluating and generating test sets that satisfy a variety of coverage criteria. In either case, we encode each (negative) test requirement for a given specification and a given coverage criterion as temporal logic formula. For evaluation, test cases are encoded as constrained finite state machines and checked against the test requirements. For generation, counterexamples are generated for each satisfiable test requirement. These counterexamples are interpreted as test cases.

We gave special attention to a mutation analysis approach suitable for state machines. Although earlier work [5, 4] applied mutation analysis to model checking specifications, this paper provides the sound formal basis lacking in the earlier efforts. The formal basis allows comparisons between different criteria; for example, we showed a relationship between mutation coverage with the stuck-at operator and the UFP variant of the popular MC/DC criterion.

Acknowledgments

We thank Angelo Gargantini and Connie Heitmeyer of the Naval Research Laboratory for the Safety Injection specifications. We also thank Vadim Okun of the University of Maryland, Baltimore County, for his mutation engine and Jeff Offutt and Aynur Abdurazik for helpful discussions. We give special thanks to Vladimir Yakhnis of Rockwell Collins for supplying the partial SMV model for the flight guidance example.

APPENDIX

This specification of the Safety Injection problem is a modification of one supplied by the Navy Research Laboratory. The specification corresponds to Table 1. See [9] for a closely related specification in all of SCR, Spin, and SMV.

Semantically, the inputs to this model are a reset signal, modeled by `Reset` in the `VAR` section, a blocking signal, modeled by `Block`, and the water pressure, `WaterPres`. Reset may be on or off, as may be the blocking input. The input water pressure, `WaterPres`, ranges from 0 to 200 (no units are given).

The boolean variable `Overridden` is an internal variable representing whether the safety injection system is overridden. Another internal variable is `Pressure`, which is the water pressure classified as three ranges, `TooLow`, `Permitted`, and `High`. The range limits are given by the macros `Low` and `Permit` given in the `DEFINE` section. The model keeps the previous value of each variable in a corresponding variable, prefixed with a `P`, to determine transitions, that is, when a value has just changed.

The `DEFINE` section encodes when the under-pressure safety injection system is activated with the variable `SafetyInjection`. Safety injection is on only if the pressure is too low and the system is not overridden.

The `ASSIGN` section starts the water pressure at 2 and the reset on. At any point either block can turn change, reset can change, or the water pressure can change by up to 3. (The assertions in the `TRANS` section ensure only one of these variables change at a time.)

```
MODULE main

VAR
Reset : {On, Off};
Block : {On, Off};
WaterPres : 0..200;
Overridden : {0,1}; --boolean
```

```
Pressure : {TooLow, Permitted, High};

PReset : {On, Off};
PBlock : {On, Off};
PWaterPres : 0..200;
POverridden : {0,1}; --boolean
PPressure : {TooLow, Permitted, High};

DEFINE
 Low := 90;
 Permit := 100;

SafetyInjection:= case
  Pressure = Permitted: Off;
  Pressure = High:Off;
  Pressure = TooLow: case
    Overridden:Off;
    ! Overridden :On;
  esac;
esac;

ASSIGN
-- ASSIGN init
init(Block):= Off;
init(Reset):= On;
init(WaterPres):= 2;
init(Overridden):= 0;
init(Pressure):= TooLow;

init(PBlock):= Block;
init(PReset):= Reset;
init(PWaterPres):= WaterPres;
init(POverridden):= Overridden;
init(PPressure):= Pressure;

-- ASSIGN NEXT
next(PBlock):= Block;
next(PReset):= Reset;
next(PWaterPres):= WaterPres;
next(POverridden):= Overridden;
next(PPressure):= Pressure;

next(Block):= {On, Off};

next(Reset):= {On, Off};

next(WaterPres):= 0..200;

next(Overridden):=
case
  ((Pressure = TooLow) &
    ((!(Pressure = next(Pressure)) |
    (!(Reset = On) & next(Reset) = On)) |
    (!(!(Pressure = next(Pressure)) |
    (!(Reset = On) & next(Reset) = On)) &
    !(!(Block = On) & next(Block) = On  & Reset = Off) &
    Overridden = 0))) |
```

```
  ((Pressure = Permitted) &
    ((!(Pressure = next(Pressure)) |
    (!(Reset = On) & next(Reset) = On)) |
    (!(!(Pressure = next(Pressure)) |
    (!(Reset = On) & next(Reset) = On)) &
    !(!(Block = On) & next(Block) = On  & Reset = Off) &
    Overridden = 0))) |
  ((Pressure = High) &
    (!(Pressure = next(Pressure)) |
    ((Pressure = next(Pressure)) &
    Overridden = 0)))
    : 0;

  ((Pressure = TooLow) &
    ((!(!(Pressure = next(Pressure)) |
    (!(Reset = On) & next(Reset) = On)) &
    (!(Block = On) & next(Block) = On  & Reset = Off)) |
    (!(!(Pressure = next(Pressure)) |
    (!(Reset = On) & next(Reset) = On)) &
    !(!(Block = On) & next(Block) = On  & Reset = Off) &
    Overridden = 1))) |
  ((Pressure = Permitted) &
    ((!(!(Pressure = next(Pressure)) |
    (!(Reset = On) & next(Reset) = On)) &
    (!(Block = On) & next(Block) = On  & Reset = Off)) |
    (!(!(Pressure = next(Pressure)) |
    (!(Reset = On) & next(Reset) = On)) &
    !(!(Block = On) & next(Block) = On  & Reset = Off) &
    Overridden = 1))) |
  ((Pressure = High) &
    (((Pressure = next(Pressure)) &
    Overridden = 1)))
    : 1;
esac;

next(Pressure):=
case
  (Pressure = TooLow &
    ((!(WaterPres >= Low) & next(WaterPres) >= Low))) |
  (Pressure = Permitted &
    (!(!(WaterPres < Low) & next(WaterPres) < Low) &
    !(!(WaterPres >= Permit) & next(WaterPres) >= Permit))) |
  (Pressure = High &
    ((!(WaterPres < Permit) & next(WaterPres) < Permit)))
    : Permitted;

  (Pressure = TooLow &
    (!(!(WaterPres >= Low) & next(WaterPres) >= Low))) |
  (Pressure = Permitted &
    ((!(WaterPres < Low) & next(WaterPres) < Low)))
    : TooLow;

  (Pressure = Permitted &
    (!(!(WaterPres < Low) & next(WaterPres) < Low) &
    (!(WaterPres >= Permit) & next(WaterPres) >= Permit))) |
  (Pressure = High &
    (!(!(WaterPres < Permit) & next(WaterPres) < Permit)))
```

```
    : High;
esac;

TRANS
 (!(next(Reset) = Reset) & next(Block) = Block & next(WaterPres) =
WaterPres) |
 (!(next(Block) = Block) & next(Reset) = Reset & next(WaterPres) =
WaterPres) |
 (!(next(WaterPres) = WaterPres) &  (next(WaterPres) - WaterPres) <= 3 &
(WaterPres -  next(WaterPres)) <= 3 & next(Reset) = Reset & next(Block)
= Block)

-- The temporal logics
SPEC AG ( -- 1
  (((PPressure = TooLow &
    ((!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) |
    (!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    !(!(PBlock = On) & Block = On  & PReset = Off) &
    POverridden = 0))) |
  (PPressure = Permitted &
    ((!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) |
    (!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    !(!(PBlock = On) & Block = On  & PReset = Off) &
    POverridden = 0))) |
  (PPressure = High &
    (!(PPressure = Pressure) |
    (PPressure = Pressure & POverridden = 0))))
    -> Overridden = 0)
 &
  (!((PPressure = TooLow &
    ((!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) |
    (!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    !(!(PBlock = On) & Block = On  & PReset = Off) &
    POverridden = 0))) |
  (PPressure = Permitted &
    ((!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) |
    (!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    !(!(PBlock = On) & Block = On  & PReset = Off) &
    POverridden = 0))) |
  (PPressure = High &
    (!(PPressure = Pressure) |
    (PPressure = Pressure & POverridden = 0))))
    -> Overridden = 1)
)

SPEC AG ( --2
  (((PPressure = TooLow &
    ((!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
```

```
    (!(PBlock = On) & Block = On  & PReset = Off)) |
    (!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    !(!(PBlock = On) & Block = On  & PReset = Off) &
    POverridden = 1))) |
  (PPressure = Permitted &
    ((!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    (!(PBlock = On) & Block = On  & PReset = Off)) |
    (!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    !(!(PBlock = On) & Block = On  & PReset = Off) &
    POverridden = 1))) |
  (PPressure = High & PPressure = Pressure & POverridden = 1))
  -> Overridden = 1)
 &
  (!((PPressure = TooLow &
    ((!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    (!(PBlock = On) & Block = On  & PReset = Off)) |
    (!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    !(!(PBlock = On) & Block = On  & PReset = Off) &
    POverridden = 1))) |
  (PPressure = Permitted &
    ((!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    (!(PBlock = On) & Block = On  & PReset = Off)) |
    (!(!(PPressure = Pressure) |
    (!(PReset = On) & Reset = On)) &
    !(!(PBlock = On) & Block = On  & PReset = Off) &
    POverridden = 1))) |
  (PPressure = High & PPressure = Pressure & POverridden = 1))
  -> Overridden = 0)
)

SPEC AG ( --3
  (((PPressure = TooLow &
    ((!(PWaterPres >= Low) & WaterPres >= Low))) |
  (PPressure = Permitted &
    (!(!(PWaterPres < Low) & WaterPres < Low) &
    !(!(PWaterPres >= Permit) & WaterPres >= Permit))) |
  (PPressure = High &
    ((!(PWaterPres < Permit) & WaterPres < Permit))))
  -> Pressure = Permitted)
 &
  (!((PPressure = TooLow &
    ((!(PWaterPres >= Low) & WaterPres >= Low))) |
  (PPressure = Permitted &
    (!(!(PWaterPres < Low) & WaterPres < Low) &
    !(!(PWaterPres >= Permit) & WaterPres >= Permit))) |
  (PPressure = High &
    ((!(PWaterPres < Permit) & WaterPres < Permit))))
  -> !(Pressure = Permitted))
)

SPEC AG( --4
```

```
  (((PPressure = TooLow &
    (!(!(PWaterPres >= Low) & WaterPres >= Low))) |
  (PPressure = Permitted &
    ((!(PWaterPres < Low) & WaterPres < Low))))
  -> Pressure = TooLow)
 &
  (!((PPressure = TooLow &
    (!(!(PWaterPres >= Low) & WaterPres >= Low))) |
  (PPressure = Permitted &
    ((!(PWaterPres < Low) & WaterPres < Low))))
  -> !(Pressure = TooLow))
)

SPEC AG( --5
  (((PPressure = Permitted &
    (!(!(PWaterPres < Low) & WaterPres < Low) &
    (!(PWaterPres >= Permit) & WaterPres >= Permit))) |
  (PPressure = High &
    (!(!(PWaterPres < Permit) & WaterPres < Permit))))
  -> Pressure = High)
 &
  (!((PPressure = Permitted &
    (!(!(PWaterPres < Low) & WaterPres < Low) &
    (!(PWaterPres >= Permit) & WaterPres >= Permit))) |
  (PPressure = High &
    (!(!(PWaterPres < Permit) & WaterPres < Permit))))
  -> !(Pressure = High))
)
```

References

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital System Testing and Testable Design.* IEEE Computer Society Press, New York, N.Y., 1990.

[2] S.B. Akers. On a theory of boolean functions. *SIAM Journal*, 7(4), 1959.

[3] 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 (DASC99)*, volume 2, page 10.A.6. IEEE, October 1999. Also NIST IR 6405.

[4] Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248. IEEE Computer Society, November 1999. Also NIST IR 6403.

[5] 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, December 1998.

[6] Joanne M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 280–292, January 1996.

[7] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, June/July 1998.

[8] Ramesh Bharadwaj and Constance Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software*, Paris, France, January 1997.

[9] Ramesh Bharadwaj and Constance L. Heitmeyer. Model checking complete requirements specifications using abstraction. Memorandum Report NRL/MR/5540-97-7999, U.S. Naval Research Laboratory, Washington, DC 20375, November 1997.

[10] Paul E. Black, Kelly M. Hall, Michael D. Jones, Trent N. Larson, and Phillip J. Windley. A brief introduction to formal methods. In *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference (CICC '96)*, pages 377–380. IEEE, May 1996.

[11] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In 15^{th} *IEEE International Conference on Automated Software Engineering (ASE2000)*, pages 81–88. IEEE Computer Society, September 2000.

[12] Paul E. Black, Vadim Okun, and Yaccov Yesha. Mutation of model checker specifications for test generation and evaluation. In W. Eric Wong, editor, *Mutation Testing for the New Century (MUTATION 2000)*, pages 14–20. Kluwer Academic Publishers, October 2000.

[13] Jerry R. Burch, Edmund M. Clarke, Jr., Ken L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[14] John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN Workshop*, Rutgers, NJ, August 1996. Also WVU Technical Report #NASA-IVV-96-022.

[15] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498 – 520, July 1998.

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

[17] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[18] Edmund M. Clarke, Jr., E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[19] Edmund M. Clarke, Jr., Orna Grumberg, and David E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency – Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[20] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[21] Richard A. De Millo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[22] Richard D. Eldred. Test routines based on symbolic logical statements. *Journal of the ACM*, pages 33–36, January 1959.

[23] 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*, pages 384–398. Springer-Verlag, April 1997.

[24] J. Gannon, P. McMullin, and R. Hamlet. Data-Abstraction Implementation, Specification, and Testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[25] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Toulouse, France, September 1999.

[26] Jens Chr. Godskesen. Fault models for embedded systems. In *Proceedings of CHARME'99*, volume 1703 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1999.

[27] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

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

[29] K. L. Heninger. Specifying software requirements for complex systems. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[30] Gerald J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[31] James Kirby, Jr. Example NRL/SCR software requirements for an automobile cruise control and monitoring system. Technical Report TR-87-07, Wang Institute of Graduate Studies, July 1987.

[32] D. Richard Kuhn. Fault classes and error detection in specification based testing. *ACM Transactions on Software Engineering Methodology*, 8(4), October 1999.

[33] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, New Jersey 08540, 1994.

[34] William J. Majurski. Issues in software component testing. To appear in *ACM Computing Surveys*, 1998.

[35] Ken L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[36] Kin C. Yan Mei. Bridging and stuck-at faults. *IEEE Transactions on Computers*, pages 720–727, July 1974.

[37] S. P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Second Workshop on Formal Methods in Software Practice*, Clearwater Beach, FL, March 1998.

[38] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[39] A. Jefferson Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[40] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 100–107, Baltimore, MD, May 1993. IEEE Computer Society Press.

[41] Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE Fifth International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, Las Vegas, NV, October 1999. IEEE Computer Society Press.

[42] Scott Ranville and Paul E. Black. Automated testing requirements-Automotive perspective. In John Penix and Nigel J. Tracey, editors, *Proceedings of the 2nd International Workshop on Automated Program Analysis, Testing and Verification*, May 2001.

[43] John M. Rushby. Ubiquitous abstraction: A new approach to mechanized formal verification. In *Proceedings of Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 176–178. IEEE Computer Society, December 1998.

[44] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11), November 1996.

[45] D. Wu, M. A. Hennell, D. Hedley, and I.J. Riddell. A practical method for software quality control via program mutation. In *Proceedings of the 2nd Workshop on Software Testing, Verification and Analysis*, pages 159–170, Banff, Canada, July 1988.

[46] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.