Using a Model Checker to Test Safety Properties*

Paul Ammann, Wei Ding, and Daling Xu
ISE Department, MS 4A4
George Mason University
4400 University Drive
Fairfax, VA 22030 USA
+1 703 993 1660
{pammann,wding1,dxu}@gmu.edu

ABSTRACT
In addition to providing a sound basis for analysis, formal methods can support other development activities; in our case the target is specification-based testing at the system level. We use the formal method of model checking to either generate new test sets or analyze existing test sets with respect to safety properties expressed in a temporal logic. We consider two types of tests: failing tests, in which a system must reject (fail) a specific dangerous action, and passing tests, in which a system must accept (pass) a safe action in a context that also includes a plausible dangerous action. We formalize our notion of dangerous actions with a mutation model for model checking specifications, and we develop coverage criteria to assess test sets. The coverage criteria are based on the logic operators from the Computation Tree Logic (CTL) and encompass the idea of scenarios where a dangerous action is either inevitable (A) or possible (E) as of the next state (X) or at some point in the future (F). We demonstrate the feasibility of our approach with an example.

Keywords
Model checking, Mutation analysis, Software testing, Safety

1 INTRODUCTION

It is well understood that software in a safety critical system should not contribute to hazards[21, page 156]. To realize this goal, a variety of approaches are possible, and indeed desirable, since any given approach suffers from some weakness. In this paper, we develop a novel approach to testing specifically tailored to probing behavior relevant to the required safety properties of a given system. In this paper we use the term *safety* in the sense of Leveson [21], and not in the 'computer science' sense, where safety – nothing bad happens – is distinguished from liveness – something good does happen [20]. Our technique gives the test engineer a method to either generate or evaluate a test set with respect to a set of safety-related test coverage criteria.

We strive to exploit a structure built for formal analysis, namely a state machine description in a model checker, to drive the test process. In this work, we assume that the system under test has a useful, finite state model encoded in the model checker SMV [9]. We further assume that safety constraints have been derived separately and are encoded in the temporal logic CTL. A variety of extant case studies that use model checkers to analyze realistic systems lend plausibility to these assumptions. One example case study is TCAS[8], where the "own-aircraft" logic of a traffic collision and avoidance system is specified in SMV. Another example case study is FGS[22], where the mode logic for a flight guidance system has been specified and analyzed in a variety of formal notations, including SMV[28].

Our basic idea is that test engineers should endeavor to place a safety-critical system in circumstances where it could plausibly violate its safety constraints to see, on the one hand, that unsafe actions are rejected, and, on the other, that safe actions are accepted. Our contribution is to provide a new, systematic approach to testing a system with respect to a given safety property. Our contribution's specific attributes are:

- A development of the notion of a *dangerous trace* with respect to a safety property P, and the extension of these traces into passing and failing tests.
- A development of coverage criteria based on dangerous traces and operators from the temporal logic CTL.
- A method to use a model checker to generate automatically test sets that satisfy a given coverage criterion. We also explain how to use a model checker to analyze an existing test set with respect to a given criterion.
- A demonstration of feasibility of our method via application to a small example.

To proceed, we must develop a notion of potentially dangerous yet plausible behavior. We have selected syntactic mutations of the descriptions of state machines for this purpose; related mutation approaches not tailored to safety are

*This work supported in part by the National Institute of Standards and Technology, by a George Mason University Graduate Research Assistant grant, and by the National Science Foundation under grant CCR-99-01030.

presented in [2, 3]. The basic idea behind mutation analysis is that if some variation is made to an artifact – traditionally code, but in this case a specification – then test data should be comprehensive enough to notice the variation and distinguish it from the original. In this case, we use mutations to model incorrect specifications. There are certainly other approaches, such as a mutation model for the state machines themselves or an error seeding approach [23], but syntactic mutations strike a good balance between generality and formal structure.

The paper is organized as follows. In section 2 we develop our basic model of dangerous traces and follow it to a definition of coverage criteria oriented to a given safety property. In section 3, we present how to use a model checker to generate a test set that satisfies a given criterion. In section 4, we show the feasibility of our method on a small example, namely the cruise control example. In section 5, we discuss related work and conclude.

2 MODEL

A model checker specification consists of two parts: a (finite) state machine description and a set of constraints on that description. The state machine description is a Kripke structure; that is, it specifies:

- A set of states. Typically, these states are implicitly defined as the cross product of the possible values of a set of variables. A subset of the states are designated as initial states.
- A transition relation.
- Atomic propositions to label each state. Typically, these are implicit in the variables that define the state.

The constraints are expressed in temporal logic over the atomic propositions. The model checker sees if the finite state machine is a model of the constraints. If not, the model checker tries to produce a counterexample. Counterexamples are useful in the context of this paper because they can be naturally interpreted as test cases [9]. That is, to generate a test case with a model checker, one simply writes a temporal logic constraint that is the *negation* of the desired properties of the test case. The model checker then obligingly generates the result as a counterexample.

To generate a set of test cases that satisfies some given test coverage criterion, one writes a set of temporal logic constraints, one for each test requirement needed to satisfy a given coverage criterion. This possibility is illustrated in figure 1. Boxes in the figure indicate activities, and arrows indicate the flow of test requirements, SMV machines, and test cases into, between, and from these activities. On the left of the figure, the coverage criteria and the (finite) system specification are jointly used to derive the test requirements. The test requirements are then evaluated against the system specification by the model checker. A counterexample from the model checker corresponds to a test case that satisfies a given test requirement. These test cases are collected and reduced to eliminate various types of redundancy [2, 5]. The result is a test set that satisfies all feasible test requirements implied by the selected test criterion with respect to the finite system specification.

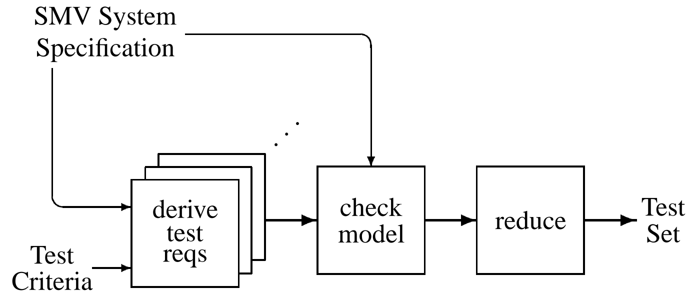

Figure 1: Test Set Generation with a Model Checker

Recognition that a given test set satisfies a particular test coverage criterion is also possible by turning each test case into a constrained finite state machine and using the same set of constraints as used for test generation. This possibility is illustrated in figure 2. On the left of the figure are now three inputs instead of two. Two of these inputs, the SMV system specification and the coverage criterion are used as in figure 1. The new input, the existing test set, is processed so that each test case is turned into a constrained finite state machine that is capable of exactly the behavior of the test case [1, 2]. The model checker is then run to evaluate each constrained machine against the test requirements, and the results are collected into a coverage report.

Recognition of test sets is an important aspect for widescale application of the technique for several reasons. First, most development organizations have invested heavily in regression test sets, and so being able to analyze these existing test sets with respect to a variety of test coverage criteria is a useful activity. Second, although these test sets include additional details not found in the simpler, finite system specification, it is an easier activity to abstract existing test cases to the resolution of the finite system model than it is to add the required details to test cases generated directly from the finite system model. This latter activity typically requires human intervention.

We suppose a set of mutation operators, *Op*, which take state machine descriptions in SMV and produce altered state machine descriptions. For example, one possible operator, the *variable replacement operator* (VRO), takes a single occurrence of some variable x in an SMV state machine description and replaces it with a different variable y of compatible type. Section 4 shows one possible set of operators, *Op*. It is noteworthy that the mutations we consider in this paper take place in the state machine description, and not in the

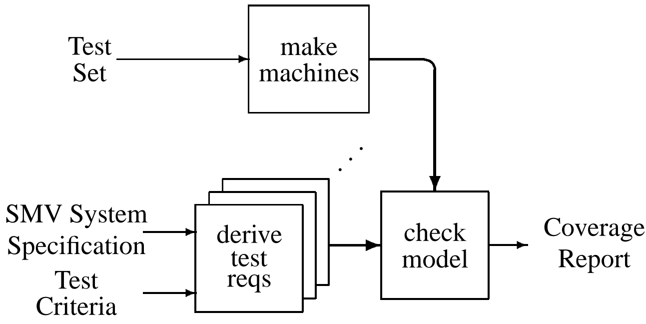

Figure 2: Test Set Evaluation with a Model Checker

temporal logic constraints, as is the case in [2, 3].[1] The systematic application of *Op* to some SMV description results in a set M of mutant machines. An example mutant machine is illustrated in section 4.

The next step is to relate these mutant machines M to the safety predicate P. A *trace* is a sequence of states. Traces in a state machine start in some initial state and include subsequent states as allowed by the transition relation. Consider a trace t allowed by the original State Machine SM. We assume that such a trace necessarily satisfies P since otherwise SM would be known to be unsafe and there would be no reason to test any implementation that refined SM. Now consider a machine SM^+ with a transition relation that is the union of that from SM with that from some mutant machine SM'. If t, the trace from SM, has the property that the last state in t can be extended, first with a transition from SM' but not SM, and possibly further with transitions from SM^+ in such a way that P is violated, then, informally, t is a *dangerous* trace.[2] The precise definition of *dangerous* depends on the way in which t is extended to violate P.

Traces can be dangerous in a variety of interesting ways. At one extreme, a trace is said to be AX *dangerous*, or simply an AX *trace*, if in the additional transitions allowed by the mutant SM', the extended trace violates P in all (A) next (X) states. In other words, an AX trace takes the system to a state where the mutant machine is guaranteed to do something dangerous on the very next transition. A trace is said to be EX *dangerous*, or simply an EX *trace*, if in the additional transitions allowed by the mutant SM', the extended trace violates P in some (E) next (X) state. In other words, an EX trace takes the system to a state where the mutant machine might to do something dangerous on the very next transition.

Similar definitions apply for the future (F) operator. An AF *trace* can be extended with the next trace from SM' and other transitions from SM^+ so that in all (A) futures (F), there is a violation of P. Finally, an EF *trace* can be extended with the next trace from SM' and other transitions from SM^+ so that in some (E) futures (F), there is a violation of P. Precise formulations in CTL for the various types of dangerous traces are presented later in the paper in the context of the associated test cases, which we discuss next.

[1] Defining the mutations on the state machine description instead of a temporal logic reflection yields a cleaner model without some awkward aspects present in [2, 3].

[2] An alternate, and more restrictive, formulation requires the trace t to be in both SM and SM' [11]. We choose the more general route here.

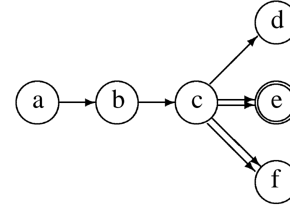

Figure 3: Dangerous EX Trace with Passing/Failing Tests

Dangerous traces lead directly to two notions for test cases: failing and passing tests. In a failing test, the dangerous trace t is extended with the transition relation in SM^+ so that P is violated. In addition, the first transition beyond t is required to come from the mutant machine SM' but not the original SM, thereby making trace t include as many transitions as possible from SM. Further, any valid refinement of SM must reject (fail) the extended trace. In a passing test, the dangerous trace t is extended with the transition relation of SM by one transition. By assumption, this trace cannot violate P, since all of the transitions are from SM. Further, any valid refinement of SM must be able to accept (pass) the extended trace. The situation for failing and passing tests is shown in figure 3. The single arcs in figure 3 represent transitions from SM (and possibly SM'), and the double arcs represent transitions that are in SM' but not in SM. Suppose that the safety property P is violated in state e (indicated by a double circle). Then the trace t equal to a, b, c is an EX trace, since there is an extension of t, namely a, b, c, e where property P is violated. Moreover, t is not an AX trace, due to the existence of a, b, c, f, which does not violate P. A passing test for EX trace t is a, b, c, d. A failing test for EX trace t is a, b, c, e. Note that passing and failing tests always come in pairs; a dangerous trace is extended one way to produce a failing test and another way to produce a passing test.

Coverage criteria derive directly from the definition of dangerous traces: A set of traces T is xy-adequate, where $x \in \{A, E\}$ and $y \in \{X, F\}$, if for each mutant SM' in M, there exists a trace t in T such that t is has a dangerous xy-trace as a prefix. Further, T is passing xy-adequate if T is xy-adequate and for each mutant SM', some dangerous xy-trace is a proper prefix of some trace in T. Correspondingly, T is failing xy-adequate if T is xy-adequate and for each mutant SM', some dangerous xy-trace is a prefix of a trace that leads to a violation of P, as determined by y - that is, either in the next state (X) or in some future state (F).

3 MODEL CHECKER IMPLEMENTATION

In this section, we discuss how to build the test model for AX, AF, EX and EF coverage criteria. It includes two parts: First, we discuss how to build a new state machine from the original and a mutant. Second, we present how to write temporal logic formulae that guide SMV to generate the required traces.

Combining the Original and Mutant State Machine
As noted earlier, a set of mutation operators Op, generates a set of mutant machines M. Each mutant SM' must be combined with the original machine SM into SM^+. In principle, this procedure is simple, but some care is required to implement it correctly in a model checking language such as SMV. Below, we describe how to do this in SMV for the set of mutation operators Op used in the example later in the paper. All of the mutation operators in Op apply to the `next` statements in SMV. The `next` statement is one useful way of specifying a transition relation in SMV.

In the SMV syntax, 1 represents *true*, and 0 *false*. The logical operators *and, or* and *not* are &, | and !, respectively.

Suppose a part of the original state machine SM is

```
next (x) := case
      p1 : v1;
   ...
      pi : vi;
   ...
      pm : vm;
       1 : x;  -- default case
esac;
```

Since case statement has an implicit semantics based on syntactic order, to simplify the problem, we assume, in SM, different order of (pi, vi) pairs does not change the semantics, that is, $p1, ..., pm$ are disjoint (if the guard conditions are not disjoint, it is not difficult to rewrite them to satisfy the assumption). In the above case statement, 1 is the default case.

If a mutation operator is applied to pi we can get the mutant state machine SM':

```
next (x) := case
     p1  : v1;
   ...
     pi' : vi;  -- pi is changed to pi'
   ...
     pm  : vm;
      1  : x;
esac;
```

To get the $AX/AF/EX/EF$ traces, we need a state machine that includes both the traces from the original and mutated state machine. If we just combine the two guard conditions to create state machine SM^+, we may lose some traces, because pi' in SM' may conceal some traces from SM, and Pi may interfere with some traces from SM':

```
next (x) := case
        p1  : v1;
   ...
   pi | pi' : vi;  -- a new line
   ...
        pj  : vj;
   ...
        pm  : vm;
         1  : x;
esac;
```

The relationship between pi and pi' can be:

1) pi' is strictly weaker than pi;

2) pi' is equal to pi;

3) pi' is strictly stronger than pi;

4) pi' is not comparable with pi;

Case 2 is the easiest; no traces are lost. In case 1, some traces from SM are lost. Because the range that pi' constrains is larger than pi does, when a pj from SM that follows $pi|pi'$ just takes the value of the difference of pi and pi', that is, if $pj\&pi'$ is true, it is impossible for a model checker to check pj in SM^+. For example, in figure 4, transition $0 \rightarrow (a = 1) \rightarrow 1$, $1 \rightarrow (a = 1) \rightarrow 1$, and $1 \rightarrow (a = 0) \rightarrow 0$ are missed in the new state machine.

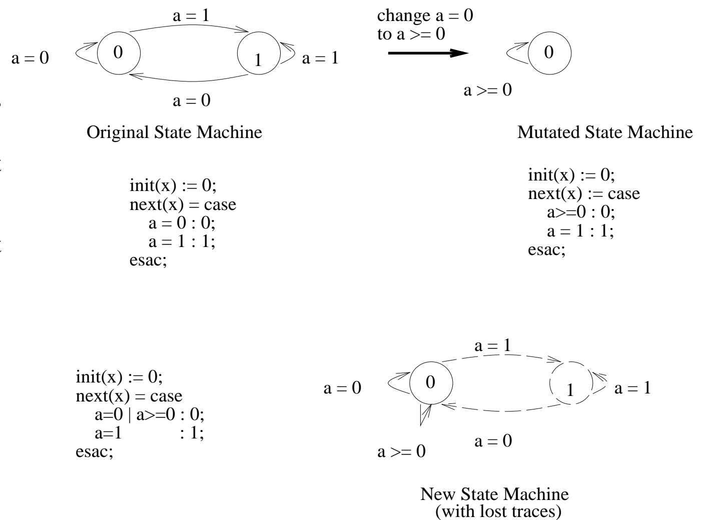

Figure 4: **A State Machine Example**

Similarly, under case 3, some traces from SM' may be lost in SM^+. For case 4, SM^+ loses some traces from both SM' and SM, because not only pi interferes with some traces from SM' but also pi' interferes with some traces from SM.

To make up possible lost traces from SM, for each condition and value pair $pj : vj$ that follows pi',

```
pj & pi' : {vj, vi};
```

is inserted before

```
pi | pi' : vi;
```

Here, SMV treats $\{vj,\ vi\}$ as a nondeterministic choice. The reason to use the value set $\{vj, vi\}$ is that the we need two arcs here, assume the current value of x is $x_{current}$, $x_{current} \rightarrow pj\&pi' \rightarrow vj$ is for the possible lost trace from SM, and $x_{current} \rightarrow pj\&pi' \rightarrow vi$ is for keeping the existing trace from SM'.

To make up possible lost traces from SM',

```
!p1&!p2...&!pi'...&!pm&pi : {vi, x};
```

is inserted before

```
pi | pi' : vi;
```

$!p1\&!p2...\&!pi'...\&!pm$ is the default case 1 in SM'. Because $p1, ..., pm$ are partitioned, the only guard condition that pi may cover is the default case 1.

Thus, a correct SM^+ should be :

```
next (x) := case
   ...
-- start: to make up traces
   !p1&!p2...&!pi'...&!pm&pi : {vi, x};
   pj & pi' : {vj, vi};
   ...
   pi | pi' : vi;
-- end
   ...
        pj  : vj;
   ...
esac;
```

The new state machine includes all the traces. In addition, we need to know when a trace has all of its transitions from the original machine and when a trace has included some trace that only the mutant machine has. A new variable *original* is created to track this behavior. If the trace from the original state machine is followed, *original* is *true*. Once a transition not from the original is followed, the variable original becomes and remains *false*. Hence, the following part is added to the new state machine.

```
VAR
   original: boolean;
...
ASSIGN
   init (original) := 1;
...
next (original) := case
   p1 & next (x) = v1 : original;
   ...
   pi & next (x) = vi : original;
   ...
   pm & next (x) = vm : original;
   !p1 & !p2 & ... & !pm & (next(x)=x)
                      : original;
   1  : 0;   -- transition is not from SM
esac;
```

Expressing Test Requirements in Temporal Logic

Our purpose is to let a model checker generate AX, AF, EX and EF failing and passing tests. If we submit the negation of our testing requirements, the model checker can find a trace that meets our requirements, assuming one exists. Note that due to the finite domain, model checking is complete, and so determining whether a test requirement is satisfiable is decidable.

As an example, for a failing testing trace that covers an EX trace, the variable `original` holds up to some point in the computational tree. Then, in some next state, the variable `original` becomes false (because a transition only in SM' is followed), and P also fails to hold. Expressing this directly in CTL, we get the test requirement:

```
EF(original & EX (!original & !P))
```

The negation we hand to SMV as a `SPEC` clause is simply:

```
SPEC !EF(original & EX (!original & !P))
```

If possible, the model checker will produce a trace violating the `SPEC` clause, and hence satisfying the original test requirement. If there is no counterexample, this simply means that the test requirement cannot be satisfied; in other words, the particular mutant SM' is not EX dangerous.

For a passing testing trace that covers such an EX trace, we want to extend the EX trace with a (safe) transition from the original machine. In temporal logic, the test requirement is (we omit negations from here on):

```
EF (original & EX (original) &
               EX (!original & !P))
```

There is a subtle point here, in that the model checker has a choice of counterexamples from which to choose: one counterexample extends the EX trace to be a passing test; the

other extends the EX trace to be a failing test. The current version of SMV chooses the proper counterexample with the formula as written. (The other can be obtained by switching the order of the two criteria.) Clearly, for robust application to testing, control over which counterexamples to generate would be a useful feature of model checkers.

The analysis above for EX traces gives the general idea; although the other cases are slightly more complex. Formula for failing and passing testing traces that cover all types of traces we consider are:

1) To cover an AX trace:

```
-- Failing
EF(original & EX(!original)
  & AX(!original -> !P))
-- Passing
EF(original & EX(original) & EX(!original)
  & AX(!original -> !P))
```

2) To cover an AF trace:

```
-- Failing
EF(original & EX(!original) &
  AX(!original -> (!P | AF(!P))))
-- Passing
EF(original & EX(original) & EX(!original)
  & AX(!original -> (!P | AF(!P))))
```

3) To cover an EX trace:

```
-- Failing
EF(original & EX(!original & !P))
-- Passing
EF(original & EX(original) &
  EX(!original & !P))
```

4) To cover an EF trace:

```
-- Failing
EF(original & EX(!original) & EF(!P))
-- Passing
EF(original & EX(original) &
  EX(!original) & EF(!P))
```

It is possible that each mutant may provide more than one $AX/AF/EX/EF$ failing/passing trace. The method only generates one of them. One trace is good enough for the testing selection with respect to each mutant, because it differentiates between the original and mutant state machine; that is, such a trace kills the mutant.

Tools for Automatically Generating Tests for Safety
To check the feasibility of our theory discussed in the previous sections, we have developed a set of tools to automatically generate and evaluate tests for safety. As we presented, basically, there are 6 steps to generate and evaluate tests. (Test recognition was discussed earlier.)

1. Select mutation operators Op and produce a set of mutant state machines.
2. For each mutant SM', build the new machine SM^+, including the variable `original` used in the SPEC clauses.
3. Write test requirements for the chosen test criterion in CTL.
4. Model check the result of the prior two steps to yield passing and/or failing test traces.
5. Collect and reduce the counterexamples into test sets [5].
6. Execute the test sets on an implementation.

There exist various prototype tools for these purposes. Okun's mutation engine accomplishes step 1 [6], and mechanism tools from Black and Ammann to do part of step 5 and 6 [3]. As part of her thesis [11], Ding implemented prototype tools for the remaining steps.

4 EXAMPLE

We use the Cruise Control [19] example as our case study. Many variations on this example exist. Table 1 is the mode transition table of the Cruise Control SCR [18]. specification. It shows the events and conditions which transitions from one mode to another. The purpose of the example is to show the feasibility of our method.

The possible states of the cruise control are partitioned into four modes:

- Off: Ignition is off.
- Inactive: Ignition is on, but cruise control is not on.
- Cruise: Ignition is on and the cruise control system is on, controlling the automobile's speed.
- Override: Ignition is on and the cruise control system is on, but not controlling the automobile's speed.

Each line of the table 1 is a transition condition. For example, The trigger event at line 3 can be expressed as

@T(Activate) WHEN [*Ignited* & *EngRun* & *!Brake*]

The third line means, if cruise control is in mode $Inactive$, when *Ignited* is true, *EngRun* is true, *Brake* is false, and if *Activate* changes from false to true, cruise control will change into mode *Cruise*.

The cruise control example has the following mode invariants, which, for the purposes of this paper, we treat as safety

Previous Mode	Ignited	EngRun	Toofast	Brake	Activate	Deactivate	Resume	New Mode
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	-	-	-	-	-	-	Off
	t	t	-	f	@T	-	-	Cruise
Cruise	@F	-	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	-	@T	-	-	-	-	
	t	t	f	@T	-	-	-	Override
	t	t	f	-	-	@T	-	
Override	@F	-	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	t	-	f	@T	-	-	Cruise
	t	t	-	f	-	-	@T	

Table 1: **SCR Specifications for the Cruise Control System**

	Mode	**Property**
1	Off	$!Ignited$
2	Inactive	$Ignited$
3	Cruise	$Ignited$ & $EngRun$ & $!Toofast$ & $!Brake$ & $!Deactivate$
4	Override	$Ignited$ & $EngRun$

Table 2: **Mode Invariants for Cruise Control**

properties. In the following discussion, we identify each property with the sequence number identified in the table 2.

The cruise control SMV specification is derived from one generated by Altee's tool [4]. We manually rewrote the *case* statements to make the guard conditions disjoint.

Using the method presented in in this paper, we automatically generated safety passing tests and safety failing tests that systematically probe the cruise control system's mode invariants. In the following sections, we present the results of this exercise.

Mutation Generation

We used the mutation operators supported by the mutation engine [6]. We illustrate each operator below with the mutant it generates from the following clause, which is the second line in table 1. Changes are emphasized by underlining.

```
CruiseControl=Inactive &
Ignited & !next(Ignited)
```

1. Constant Replacement(CRO): replace one constant by another syntactically legal one, e.g.,

```
CruiseControl=Off & Ignited &
!next(Ignited)
```

2. Variable Replacement(VRO): replace a variable with another variable of the same type, e.g.,

```
CruiseControl=Inactive &
EngRun & !next(Ignited)
```

3. Simple Expression Negation Operator (SNO): replace a simple expression by its negation, e.g.,

```
CruiseControl=Inactive &
!Ignited & !next(Ignited)
```

4. Expression Negation Operator (ENO): replace an expression by its negation, e.g.,

```
!(CruiseControl= Off & Ignited
& !next(Ignited))
```

5. Operator Replacement(LRRO): replace one logical/relational operator with another, e.g., replace "and(&)" with "or(|)", or replace $\geq$ with $\leq$.

```
CruiseControl=Inactive |
Ignited & !next(Ignited)
```

6. Stuck-At Operator(STO): replace a simple expression with True(1) or False(0) respectively. e.g.,

```
1 & Ignited & !next(Ignited)
```

The above set of mutation operators produced a total of 456 mutant cruise control SMV specifications. Of these, 256 had dangerous traces with respect to at least one property. Table 3 shows number of mutants that produced dangerous traces of each possible variety with respect to a given property. As noted earlier, each dangerous trace can be extended into either a passing or a failing test. It is important to note that the number of tests is much smaller than the number of counterexamples indicated by table 3. The reason is that the same test may serve to kill many mutants.

Property	AX	EX	AF	EF
1	15	16	15	16
2	22	23	22	23
3	182	191	182	191
4	44	46	44	46

Table 3: **Mutants with Dangerous xy Traces**

As an illustration of how the method works, consider the application of the SNO operator to the machine SM for cruise control. The mutant state machine that has only EX dangerous traces but no AX dangerous traces. The original machine include the following next statement:

```
(CruiseControl = Inactive & Ignited &
 next(Ignited) & EngRun & !Toofast &
 !Brake & !Enum1=Activate &
  next(Enum1)=Activate) : Cruise
```

In the mutant machine, the SNO operator negates the last condition, `next(Enum1)=Activate`:

```
(CruiseControl = Inactive & Ignited &
 next(Ignited) & EngRun & !Toofast &
 !Brake & !Enum1 = Activate &
 !next(Enum1) = Activate) : Cruise
```

The mutant can violate property 3, because it is possible for mode variable *Inactive* to move to mode *Cruise* when next(Enum1) is not *Activate*. For this particular mutant, an EX trace starts in the initial state, turns on the ignition, starts the engine running, and sets cruise control to Inactive. The EX trace extended with state 1.3 yields failing EX trace as shown below in the output of SMV. This failing test differentiates the original from the mutant.

```
state 1.1:
original = 1
mutant = 1
Ignited = 0
EngRun = 0
Toofast = 0
Brake = 0
Enum1 = Resume
CruiseControl = Off

state 1.2:
Ignited = 1
EngRun = 1
CruiseControl = Inactive

state 1.3:
original = 0
EngRun = 0
CruiseControl = Cruise
```

Using the method of this paper, we generated both passing and failing sets of tests that are xy adequate for each of the types of dangerous traces: EX, AX, EF, and AF. The test sets were reduced with the tools of [3]. The sizes of the resulting test sets are shown in table 4. The first entry in the table is the number of passing tests; the second is the number of failing tests. Notice that in some cases the number of passing and failing tests differs. This is because, on the one hand, a single passing test may serve as a complement for a variety of failing tests. On the other hand, the model checker sometimes chooses from a variety of possible passing tests, thereby increasing the number of passing tests above the absolute minimum. As mentioned earlier, more control over counter example generation would be a very desirable feature in model checkers that are used for test case generation.

Property	AX **Tests**	EX **Tests**	AF **Tests**	EF **Tests**
1	5/5	6/5	5/5	6/7
2	3/4	3/3	3/4	4/4
3	19/21	22/21	19/21	23/33
4	4/5	4/5	4/5	4/7

Table 4: **Sizes of Passing/Failing Test Sets**

As a check, we implemented the cruise control model in Java and used a test driver to automatically execute the passing and failing tests. As expected, the implementation accepts all the passing tests and refuses all the failing tests.

As a further informal check on the test sets generated by the method in this paper, we implemented a Java program of CruiseControl and manually planted 8 different faults in the Java implementations (one fault in each implementation). The faults were manufactured by hand by one of the authors (Xu) before the author started working on the project. Then we ran the Java programs, following each step of each test case, to see whether the variable CruiseControl of the Java program is consistent with the expected values in the test case. The correct Java implementation should be consistent with all the passing test sets and conflict with all the failing test sets. Each incorrect implementation should be exposed by the test sets, that is it should conflict with some of the passing test sets or possibly be consistent with some failing test case.

The result of the correct implementation is as we expected: it passed all the passing tests and rejected all the failing tests. 7 of the 8 faults were exposed by the test sets. The remaining fault not exposed was found to be consistent with the requirements, and so not properly a 'fault'. So our test sets exposed all the non-equivalent faults we planted. Our conclusion is that the test sets can indeed find faults in implementations. Of course, more rigorous study is required to determine the precise effectiveness of test sets that satisfy the new coverage criteria we developed in this paper.

5 RELATED WORK AND 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 apply the powerful computation engine of model checking to the problem of evaluating and generating test sets that satisfy novel coverage criteria targeted at safety predicates.

A broad span of research from early work on algebraic specifications [16] to more recent work such as [26] addresses the problem of relating tests to formal specifications. Testing finite state machines has received considerable attention, e.g. Fujiwara et al's work on conformance testing for protocols [15]. Such work typically addresses states and transitions directly. Here, we work with a syntactic description of the state machine rather than explicit states and transitions. The benefit of our approach is the potential to scale to very large state spaces; the cost is that transitions and states are considered through the abstraction rather than directly.

Counterexamples from model checkers have been recognized as potentially useful test cases. Callahan and Schneider used a model checker to generate tests that cover each block in a certain partitioning of the input domain [7]. Engels *et al* used a model checker to generate network tests [12]. In their work, they used the term 'negative test' for what we call a failing test. Ammann *et al* defined a mutation analysis approach to generating and recognizing tests with a model checker [2, 3]. Gargantini and Heitmeyer used model checkers to generate tests for systems with SCR requirement specifications [17]. Their method yields branch coverage on the SMV description of the requirements; the relationship between branch and mutation coverage at the specification level is essentially the same as it is at the program source code level. Ritchey and Ammann used a model checker to provide comprehensive attack scenarios to test heterogeneous networks [25]; Ramakrishnan and Sekar used a model checker to carry out a related analysis in single host systems [24]. Traditional program mutation analysis [10] is a code-based method for developing a test set that is sensitive to small syntactic changes to the structure of a program. A variety of researchers, including the current authors, have adapted mutation analysis to the specification level [3, 13, 14, 27]. What is new in the present work is the targeting of test cases towards specific safety predicates, with the resulting definition of safety-related coverage metrics. Testing is a small, but important, piece of the safety puzzle. A useful guide to the myriad other techniques necessary is Leveson's text [21].

To summarize, in this paper we developed a notion of a *dangerous trace* with respect to a safety property P, and extended these traces into passing and failing tests. We developed coverage criteria based on dangerous traces and operators from CTL. We showed how to use a model checker to generate test sets that satisfy a given coverage criterion. We also explained how to use a model checker to analyze an existing test set with respect to a given criterion. Finally, we demonstrated the feasibility of our method via application to a small example.

ACKNOWLEDGEMENTS

We acknowledge Paul Black of the National Institute of Standards and Technology for his many contributions to this project. We thank Vadim Okun of the University of Maryland, Baltimore County, for his mutation engine and Jeff Offutt and Aynur Abdurazik for helpful discussions. Anonymous referees provided helpful suggestions for improving our presentation.

REFERENCES

[1] A. Abdurazik, P. Ammann, W. Ding, and J. Offutt. Evaluation of three specification-based coverage testing criteria. In *Proceedings ICECCS 2000: 6th IEEE International Conference on Engineering of Complex Computer Systems*, pages 179–187, Tokyo, Japan, September 2000.

[2] P. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings HASE99: 4th IEEE International Symposium on High Assurance Systems*, pages 239–248, Washington, DC, November 1999.

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

[4] J. 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, Jan. 1996.

[5] P. E. Black. Modeling and marshaling: Making tests from model checker counterexamples. In *Proceedings of the 19th Digital Avionics Systems Conference (DASC00)*, pages 1.B.3–1–1.B.3–6, Philadelphia, PA, October 2000.

[6] P. E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *Proceedings ASE2000: 15th IEEE International Conference on Automated Software Engineering*, pages 81–88, Grenoble, France, September 2000.

[7] J. Callahan and F. Schneider. Specification-based testing using model-checking. Technical report, WVU, August 1996. #NASA-IVV-96-022.

[8] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498 – 520, July 1998.

[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, USA, 2000.

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

[11] W. Ding. Using model checking to generate test cases for critical systems. Master's thesis, George Mason University, Fairfax, VA, August 2000.

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

[13] S. C. P. F. Fabbri, J. C. Maldonado, and M. E. D. an d P. C. Masiero. Mutation analysis testing for finite state machines. In *5th IEEE International Symposium on Software Reliability Eng ineering (ISSRE'93)*, pages 220–229, Monterey, CA, November 1994.

[14] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and E. W. Wong. Mutation analysis applied to validate specifications based o n petri nets. In *Proceedings of the 8th International Conference on Formal De scription Techniques (FORTE'95)*, pages 329–337, Quebec, Canada, October 1995.

[15] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[16] 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.

[17] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European Software Engineering Conference held joinly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 146–162, Toulouse, France, September 1999.

[18] C. 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.

[19] J. 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.

[20] L. Lamport. "Sometime" is sometimes "Not Never" – On the temporal logic of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[21] N. G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, Reading, Massachusetts, 1995.

[22] 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.

[23] H. Mills. On the statistical validation of computer programs. Technical Report FSC 72-6015, IBM Federal Systems Division, Gaithersburg, MD, 1972.

[24] C. Ramakrishnan and R. Sekar. Model-based vulnerability analysis of computer systems. In *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation*, September 1998.

[25] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (Oakland 2000)*, pages 156–165, Oakland, CA, May 2000.

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

[27] M. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *Software Engineering Journal*, pages 211–224, July 1993.

[28] V. Yakhnis, 1999. Personal Communication.