International Journal of Reliability, Quality and Safety Engineering

A SPECIFICATION-BASED COVERAGE METRIC TO EVALUATE TEST SETS

PAUL E. AMMANN[a]
Department of Information and Software Engineering
George Mason University, MS 4A4, Science and Technology II
Fairfax, Virginia 22030-4444 USA

and

PAUL E. BLACK
Information Technology Laboratory
National Institute of Standards and Technology, 100 Bureau Drive, Stop 8970
Gaithersburg, Maryland 20899-8970 USA

Received (received date)
Revised (revised date)

Software developers use a variety of formal and informal methods, including testing, to argue that their systems are suitable for building high assurance applications. In this paper, we develop another connection between formal methods and testing by defining a specification-based coverage metric to evaluate test sets. Formal methods in the form of a model checker supply the necessary automation to make the metric practical. The metric gives the software developer assurance that a given test set is sufficiently sensitive to the structure of an application's specification. We also develop the necessary foundation for the metric and then illustrate the metric on an example.

1. Introduction

There is an increasing need for high quality software, particularly for high-assurance applications such as avionics, medical, and other control systems. Developers have responded to this need in many ways, including improving the process, increasing the attention on early development activities, and using formal methods for describing requirements, specifications, and designs. Although all of these improvements contribute to better software, software still requires testing, and thus precise metrics are useful to evaluate such testing. In this paper we develop a metric based on a mutation adequacy criterion, and we explain how the metric can be evaluated with respect to the software's specifications and existing test sets.

There are many approaches to generating tests.[3,7,16,25,29,36,37] There are also measures of the completeness, adequacy, or coverage of tests on source code.[39] However, there are few objective measures of coverage that are independent of the

[a]Supported in part by the National Institute of Standards and Technology and in part by the National Science Foundation under grant CCR-99-01030.

implementation.[13] We have developed an innovative method that combines mutation analysis and model checking, which is useful for evaluating the coverage of system tests, comparing test generation methods, and minimizing test sets. Most coverage metrics apply to source code, which makes them difficult to apply in cases of conformance or behavioral testing or when developing tests before the code is finished. Since our method measures coverage over specifications, it can be used to evaluate test sets independent of code.

We define a typical notion of coverage criteria and coverage metrics. A *test requirement* is a predicate on an artifact r and a test case t. The artifact r is usually a program, but in our context r is a specification. As an example, one test requirement for statement coverage on a program P is to visit the fifth line. If t visits line 5 in P, the test requirement is satisfied; otherwise, it is not satisfied. A *coverage criterion* C is a predicate over a set of test requirements R for some artifact r and some test set T. If each test requirement in R is satisfied by at least one test case t in T, then C is satisfied. The problems of infeasible test requirements, such as requiring a visit to line 5 in a program in which line 5 is unreachable, have long been recognized; see Frankl and Weyuker's work for an in depth study in the data flow context.[19] We address infeasible test requirements in Sect. 2.1; fortunately the finite context of the model checker renders these problems decidable. At its simplest, a *coverage metric* is a measure of the degree to which a test criterion is satisfied. As we explain in Sect. 3.3, we object to the simple metric of the ratio of satisfied test requirements to the total number of test requirements, and present refinements to the simple metric.

In this paper we categorize mutations of temporal logic formulae with respect to specification coverage analysis (Sect. 2.1). We then explain *reflection*, in which a state machine description is rewritten into a temporal logic (Sect. 2.2), and define *expounding*, in which implicit aspects of a model checking specification are made explicit (Sect. 2.3). We describe how to symbolically evaluate a test set for mutation adequacy (Sect. 2.4). Using the preceding techniques as a foundation, we define the specification coverage metric (Sect. 3). We illustrate these ideas with the Safety Injection example[8,9] (Sect. 4).

1.1. *Background and related work*

Traditional program mutation analysis[15] is a code-based method for developing a test set that is sensitive to any small syntactic change to the structure of a program. A mutation analysis system defines a set of mutation operators. Each operator is a pattern for a small syntactic change. A *mutant program*, or more simply, *mutant*, is produced by applying a single mutation operator exactly once to the original program. The rationale is that if a test set can distinguish the original program from a mutant, the test set exercises part of the program adequately. Applying the operators systematically generates a set of mutants. Some of these mutants may still be equivalent to the original program. A test set is mutation adequate if at least one test in the test set distinguishes each nonequivalent mutant. There are test data generation systems that, except for the ever-present undecidability problem, attempt to automatically generate mutation adequate test inputs.[16] Very little work on mutation analysis for specifications has been reported in the literature; however, Woodward did apply mutation analysis to algebraic specifications.[37]

The example we use in this paper was originally coded using the Software Cost Reduction (SCR) method.[24] SCR uses tables to formally capture and document the requirements of a software system. It is scalable and its semantics is easy to understand. The scalability and semantics account for the use of the SCR method and its derivatives in specifying practical systems.[18,23,34] Research in automated checking of SCR specifications includes consistency checking and model checking. The NRL SCR toolkit includes the consistency checker of Heitmeyer, Jeffords, and Labaw.[22] The checker analyzes application-independent properties such as syntax, type mismatches, missing cases, circular dependencies and so on, but not application-dependent properties such as safety and security. The toolkit also includes a backend translator to the model checker SPIN.[26] Atlee's model checking approach[4,5,6] expresses an SCR mode transition table as a logic model and the safety properties as logic formulae and uses a model checker to determine if the formulae hold in the model. Owre, Rushby, and Shankar[31] describe how the model checker in PVS can be used to verify safety properties in SCR mode transition tables.

The model checking approach to formal methods specifies a system with a state transition relation and then characterizes the relation with properties stated in a temporal logic. Model checking has been successfully applied to a wide variety of practical problems, including hardware design, protocol analysis, operating systems, reactive systems, fault tolerance, and security. Although model checking began as a method for verifying hardware designs, there is growing evidence that it can be applied with considerable automation to specifications for relatively large software systems, such as the 'own-aircraft' logic for TCAS II.[12] The increasing utility of model checkers suggests using them in aspects of software development other than pure analysis, which is their primary role. Mutation analysis of specifications yields mutants from which the SMV model checker generates counterexamples. These counterexamples can be used as test cases.[3] Gargantini and Heitmeyer used a model checker to generate branch coverage tests for SCR specifications.[20] Also using model checkers, Engels, Feijs, and Mauw generated network tests,[17] Ramakrishnan and Sekar generated single host security attacks,[32] and Ritchey and Ammann generated network security attacks.[33]

The chief advantage of model checking over the competing approach of theorem proving is complete automation. Human interaction is generally required in theorem provers 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.[14] We use the SMV model checker. It is freely available from Carnegie Mellon University and elsewhere.

2. Computing a Coverage Metric With a Model Checker

Our method begins with a specification of the system and a set of tests to be evaluated against the specification, as diagrammed in Figure 1. Although the specification need not be a complete description of the system, the more detailed the specification, the more that can be checked. We generate many variants, or mutants, of the original specification. The set of tests are converted to finite state machines and are symbolically executed, one at a time. Some mutants are found to be inconsistent, that is, the model checker finds a difference between the symbolic execution of the test case and this mutant. Since we assume that the test cases are consistent with the original specification, this indicates an inconsistency with the original specification. If a mutant is found to be inconsistent by any of the test cases in the test set, it is considered to be *killed* by the test set. The ratio of killed mutants to total mutants is a coverage metric, similar to that of Wu *et. al.*,[38] but applied to specifications. The higher the ratio, the better or more completely the test set covers the specification. The lower the number, the less complete the covering.

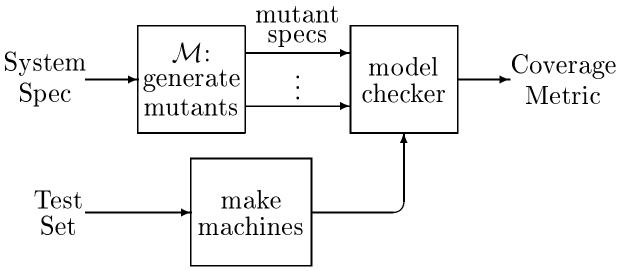

Figure 1: Specification coverage flow

Generally, testing is an attempt to assess the quality of a piece of software. If a test set inadequately exercises some part of the software, the assessment is less accurate. Since the software should correspond with the specification, a test set with better coverage of the specification is likely to more accurately assess the quality of a piece of software.

Program-based mutation analysis relies on the competent programmer hypothesis: programmers are likely to construct programs close to the correct program, and hence test data that distinguish syntactic variations of a given program are, in fact, useful. Here we assume an analogous "competent specifier hypothesis," which states that analysts are likely to write specifications which are close to what is desired. Hence test cases which distinguish syntactic variations of a specification are, in fact, useful.

2.1. *Categories of specification mutations*

A specification for model checking has two parts. One is a state machine defined in terms of variables, initial values for the variables, and a description of conditions under which variables may change value. The other part is temporal logic constraints on valid execution paths. Conceptually, a model checker visits all reachable states and verifies that the invariants and temporal logic constraints are satisfied. Model checkers exploit clever ways of avoiding brute force exploration of the state space, for example, see Burch *et. al.*[11]

Figure 2 illustrates the difference between mutations of logic formulae and mutations of program code. Code mutants are classified as either equivalent or nonequivalent. An equivalent mutant is one which has exactly the same input/output relation as the original program.[b]

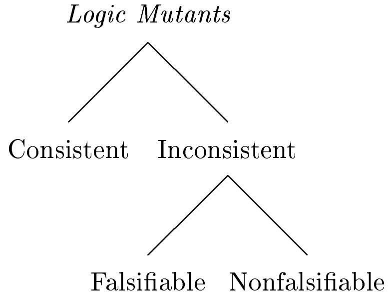

Figure 2: Categories of mutants

Mutations to logic constraints in a model checking specification result in a different situation. Instead of being either equivalent or nonequivalent, mutants are either *consistent* or *inconsistent* with the state machine. A consistent mutant is simply a temporal logic formula that is true over all possible traces defined by the state machine. Just as equivalent mutants cannot be distinguished from the original for program-based mutation analysis,[c] consistent mutants cannot be found false for model checking mutation analysis. Fortunately, consistency is decidable for these temporal logics, and model checkers are specifically designed to efficiently determine whether or not a temporal logic formula is consistent. So in this arena we do not have the problem of undecidability or requiring human judgement.

For inconsistent mutants, there are two possibilities. Some temporal logic formulae can be shown inconsistent with a single trace through the state machine. For example, if the assertion "the East-West light is never green while the North-South light is green" were inconsistent, the inconsistency could be exhibited with an execution trace that ends in a state where both lights are green. We call mutants that are demonstrably inconsistent *falsifiable*. Other temporal logic formulae may be inconsistent with respect to the state machine, but cannot be shown inconsistent

[b] We refer only to the "strong" version of program-based mutation analysis[15] here. In it a test case kills a mutant if execution reaches the mutant (the execution property[35]), the mutant corrupts the internal state (the infection property), and the corrupt internal state eventually results in an incorrect output (the propagation property). For weak mutation testing,[27] only the execution and infection properties are required.

[c] For strong mutation testing, equivalent mutants compute the same input output pairs as the original program. Hence no test case can distinguish an equivalent mutant from the original program.

with a single trace. For example, an inconsistent assertion that "eventually both the East-South and West-North left turn lights are green simultaneously" cannot be shown to be false with any single trace from the state machine. We call mutants that are inconsistent but lack a counterexample *nonfalsifiable*.

2.2. *Expressing specifications in CTL*

In our method, mutations are applied to temporal logic formulae. It is possible, and indeed desirable, to take advantage of existing constraints, such as safety assertions. However, such constraints may not be available, and, in any case, they are typically relatively loose constraints on the state machine. The difficulty with loose constraints is that mutants derived from them may be insensitive to many possible variations of the state machine.

To overcome this problem, we mechanically derive a set of temporal logic formulae for mutation. These formulae restate in temporal logic, or *reflect*, the state machine's transition relation. Although the process is conceptually straightforward, there are subtle issues that require attention. To our knowledge, the literature does not have a comprehensive treatment of this topic for model checkers. Atlee and Buckley faced a similar problem and developed their own solution.[5]

In SMV, there are two ways to specify a state machine transition relation: either procedurally via `next` statements in the ASSIGN section, or via constraints in the TRANS section.

The interesting `next` statements are conditionals of the form:

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

The semantics is typical of a programming language case statement. Boolean expression *b1* is evaluated; if it is true, *v1* is the next value for x. The value *v1* may be a set, thereby allowing for nondeterminism. If *b1* is false, *b2* is evaluated. The case often ends in a default, which is `1`, or true, in SMV.

To express the first case in CTL, we write a formula such as:

```
SPEC AG(b1 -> AX(x = v1))
```

This says that in all states (`AG`), if *b1* is true, all possible next states (`AX`) have $x = v1$. If *v1* is a set, we write:

```
SPEC AG(b1 -> AX(x in v1))
```

For *b2*, the situation is slightly more complicated. The preceding conditions, *b1* in this case, need to be subtracted out:

```
SPEC AG(!b1 & b2  -> AX(x = v2))
```

There are more subtle aspects to the process of determining the guards that we address in Sect. 2.3 below.

2.2.1. *Expressing* `next` *clauses*

In the ASSIGN section, SMV allows the use of the `next` modifier for variable references; it evaluates the variable in the destination state instead of the current state. Unfortunately, the `next` modifier is not allowed in SPEC clauses. There are two routes out of this: create "shadow" variables that track the values from the previous state (this is the solution adopted in Atlee and Buckley[5]), or access the variable after the `X` (next state) operator in CTL.

The first approach is simple, but it increases the number of variables, thereby potentially exploding the size of the state space. The second approach leads to a flat structure with potentially a large number of SPEC clauses. Consider the following `next` clause:

```
next (x) := case
   x=2 & next(y)=3 : 5;
   ...
esac;
```

Using the second approach, we refer to the value of y in the next state.

```
SPEC AG(x=2 -> AX(y=3 -> x=5))
```

The potential for explosion arises when guards reference both current and next values of a variable.

```
next (x) := case
   x=2 & y=next(y) : v1;
   ...
esac;
```

Using the second approach, we must explicitly enumerate the possible values for the variable, and test both before and after the `X` operator.

```
SPEC AG(x=2 & y=1 -> AX(y=1 -> x=v1))
SPEC AG(x=2 & y=2 -> AX(y=2 -> x=v1))
...
```

If we use the first approach instead, we add a new variable, *prevy*, which keeps the previous value. The specification refers to previous values in future states.

```
next (prevy) := y;

SPEC AG(x=2 -> AX(prevy=y -> x=v1))
```

2.2.2. *Expressing* `TRANS` *clauses*

For transition relations specified with the `TRANS` construct, reflection is simpler, since the `TRANS` constructs already are in CTL. Prefixing the predicate with the `AG` operator makes it a `SPEC` clause. The only issue is the use of the `next` operator, which can be handled in the same way as before, either with explicit "previous" values or judicious use of the `X` operator.

2.2.3. *Expressing processes*

SMV also supports a process construct, whereby changes to groups of variables are gathered into modules. The process semantics is that one process is chosen at a time. The process construct conveniently mirrors the notion of an operation or transaction in traditional programming, including the notion of atomicity. The SPEC clauses of the temporal logic do not have an analogous structure associated with them. We suggest identifying explicitly the different processes and using the identifiers to write tight SPEC clauses for the reflection. So, changes to variables in process $p1$ would be captured in the following template for a SPEC clause:

```
SPEC AG(...  -> AX(processID=p1 -> ...))
```

2.3. *Expounding*

As previously noted, the structure of guards must be elaborated when reflecting from the transition relations, since `case` statements have an implicit semantics based on syntactic order, whereas SPEC clauses are unordered. It turns out that for the purpose of mutation testing, more care is needed. In particular, it is easy to overspecify a SPEC clause. An overspecified clause yields a set of mutants that is not as sensitive as it could be. We call the process of elaborating guards *expounding*.

An example may clarify the matter. Consider the following statement from the Safety Injection problem:

```
next(Overridden):= case
  Pressure=TooLow: case
    !(Pressure=next(Pressure) ) : 0;
    !(Reset=On) & next(Reset)=On : 0;
    Block=Off & next(Block)=On & Reset=Off : 1;       -- Third case
    1 : Overridden;
   esac;
...   -- cases for Permitted and High
esac;
```

The third case, marked above, says that if *Block* is off in the current state but is on in the next state and *Reset* is off in the current state, *Overridden* is set to 1 (true) in the next state. Subtracting out the first two cases and simplifying writes the condition of the third case as:

```
Pressure=next(Pressure) & Reset=Off & next(Reset)=Off
    & Block=Off & next(Block)=On : 1;
```

Notice that explicit consideration has been made for *Pressure* not changing, due to the first guard in the case statement, and *Reset* not changing, due to the second guard. The case where *Pressure* = *TooLow* may be reflected in CTL as:

```
SPEC  -- Long version
AG(Pressure=TooLow & Reset=Off & Block=Off ->
    AX(Pressure=TooLow & Reset=Off & Block=On -> Overridden))
```

The following SPEC clause is also consistent because other parts of the specification constrain the way in which variables may change.[d]

```
SPEC  -- Short version
AG(Pressure=TooLow & Block=Off & Reset=Off
      -> AX (Block=On -> Overridden))
```

Shortening specifications increases the precision of a mutation analysis coverage metric because a test set that kills all of the mutants generated from the long version does not necessarily kill all of the mutants generated from the short version. Consider what happens if a mutant operator changes the first occurrence of the predicate *Pressure* = *TooLow* to *Pressure* = *Permitted*. The two resulting SPEC clauses are as follows:

```
-- Mutation of long version
SPEC AG(Pressure=Permitted & Block=Off & Reset=Off ->
    AX(Pressure=TooLow & Reset=Off & Block=On -> Overridden))

-- Mutation of short version
SPEC AG(Pressure=Permitted & Block=Off & Reset=Off
      -> AX (Block=On -> Overridden))
```

The mutation of the long version is still consistent with respect to the state machine, because it is not possible for both *Pressure* and *Block* to change on the same transition. Therefore, no valid test case can kill it. However, the short version mutation is both inconsistent and falsifiable.

How does one get enough redundancy to express the semantics of nonoverlapping alternatives of case statements, without adding redundancy that reduces the number of falsifiable mutants? Procedurally, it is straightforward: systematically drop predicates from the long version and run the model checker on the result. If the result is still consistent, the dropped predicate is redundant and can be omitted during mutation analysis.

An alternate strategy is to use the boolean derivative.[1] Consider a predicate P that contains a boolean condition x. If dP/dx evaluates to false, this implies that P does not depend on x, and x can safely be dropped from P.

As a procedural aside, we found it helpful to use Karnaugh maps to simplify the expressions resulting from expounding. We have used SMV's BDD facility to partially automate this aspect of specification preparation. Also, we have found it to be relatively straightforward using the model checker to examine the manual simplifications.

2.4. *Symbolic execution of test cases*

Conceptually, a test case is a single trace through the state machine. We can express the test case as a *constrained finite state machine*, or CFSM, by adding a special

[d] For details, look at the TRANS specification of the complete example in the appendix. The relevant constraint is that only one of *Pressure*, *Reset*, and *Block* may change on any one transition.

variable, *State*, which controls the machine. Each original variable gets a new value depending solely on *State*. Otherwise it is unchanged.

Expressing a test case as a CFSM allows the model checker to symbolically execute the test case and check specifications for consistency. Consider the following simplified test case, which essentially turns *Reset* on then off again.

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

`ASSIGN` statements that execute this test case are the following. 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*.

```
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;
```

In a reactive system, such as Safety Injection, freezing the state at the end of the test is acceptable. However, consider systems that have no quiescent state, such as a free-running counter. Consistent specifications may indicate that the state always changes. The specification is inconsistent with a CFSM generated as described, since the state freezes, but conceptually the specification is not wrong.

One may elaborate the CFSM with a special variable, *Check*, and set it false when the test ends. All the specifications may be automatically rewritten to include *Check* and evaluate to true when *Check* is false. This rewriting is detailed in Ammann, Black, and Majurski.[2]

3. A Specification Coverage Metric

The specification coverage metric for a test set, T, over a specification, r (for "requirements"), is conceptually simple. It is similar to the test data adequacy of Wu *et. al.*,[38] but must be applied to specifications, not programs. Given a method, $\mathcal{M}$, for creating a set of mutants, the score, S, is the number of mutants killed by the test set, k, divided by the total number of mutants, N, produced by $\mathcal{M}$ on r.

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

When the method, specification, and test set parameters are understood, we omit them, thus $S = k/N$. We usually express the score as a percentage. The lowest, or worst, score is 0% when no mutants are killed. The highest, or best possible, score is 100% when all mutants are killed.

3.1. *Preparing the specification*

The method $\mathcal{M}$ for creating a set of mutants has three parts:

1. preparing the specification,
2. generating mutants, and
3. winnowing the mutants.

We use reflection, which is described in Sect. 2.2, to produce a fully explicit temporal logic description of the state machine's transition relation and any TRANS constraints. A fully explicit specification yields a more precise mutation analysis. We then shorten the resulting clauses as described in Sect. 2.3.

As pointed out there, a test set may kill all falsifiable mutants from overly-specified, long clauses, but still not kill all falsifiable mutants of the shorter versions. Let $\mathcal{M}_s$ be a mutation process that shortens clauses before producing mutants, and $\mathcal{M}_l$ be a mutation process that uses the longer, overly-specified clauses. Suppose a test set, T_1, kills all mutants from $\mathcal{M}_l$, but not all those from $\mathcal{M}_s$, and another test set, T_2, kills all mutants from both. The scores using $\mathcal{M}_s$ show the difference between the two test sets, while using $\mathcal{M}_l$ does not:

$$S(\mathcal{M}_l, r, T_1) = S(\mathcal{M}_l, r, T_2)$$
$$S(\mathcal{M}_s, r, T_1) < S(\mathcal{M}_s, r, T_2)$$

We can see that mutation analysis on the shortened clauses, $\mathcal{M}_s$ in this case, is a more precise metric.

3.2. *Mutation operators*

The heart of mutation analysis is generating mutants. Completely unconstrained changes would yield mostly syntactically incorrect mutants which are entirely meaningless, so a set of mutation operators is used. Each operator specifies a small syntactic change that is likely to be meaningful. For example, the "wrong variable" operator replaces a single occurrence of a variable with another variable of compatible type. The specification $a \wedge b$ might yield $c \wedge b$. The "wrong relational operator" mutation operator replaces any of $<, \leq, =, \neq, \geq$, or $>$ with one of the other five possibilities. The specification $(a < b) \wedge c$ might yield $(a = b) \wedge c$ by replacing $<$ with $=$.

Kuhn showed that some operators subsume others.[28] That is, any test set that kills all mutants of a subsuming operator also kills all mutants of the subsumed operator. Thus if we use a subsuming operator, we do not need to use the subsumed operator. This lets us get the same precision with fewer mutants.

We could get maximum precision by using every conceivable operator that is not subsumed by another operator. However, Black, Okun, and Yesha show that a carefully chosen set of mutation operators yield a fraction of the mutants, but still give excellent precision.[10]

3.3. *Winnowing mutants*

Mutation operators may produce useless or duplicate mutants. To increase the precision and accuracy of the metric, we include a step to discard undesirable mutants. We call this step *winnowing*. There are three distinct types of mutants which one may wish to discard.

1. "Equivalent" mutants, i.e., mutants that are consistent,
2. Trivial mutants, i.e., mutants that are false for all traces, and
3. Duplicate mutants, i.e., different instances of semantically identical mutants.

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

The first type of mutants to winnow are those which are consistent with the specification. For instance, here is a clause from an automobile cruise control specification that says if the cruise control mode is *Override* and the ignition is turned off, the cruise control goes off:

```
AG (CMode=Override -> AX(prevIgnited & !Ignited -> CMode=Off))
```

A "replace constant" mutation operator may change the conditioning mode to *Cruise*, as below, but that is still consistent since turning the ignition off in *Cruise* mode *should* turn the cruise control off.

```
AG (CMode=Cruise -> AX(prevIgnited & !Ignited -> CMode=Off))
```

Since consistent mutants are impossible to kill, leaving them prevents any test set from scoring 100%. If the number of consistent mutants is some fraction, α, of the number of inconsistent mutants, the score is linearly reduced by $1/(1+\alpha)$.

Formally, the more accurate score, that is, computed with only inconsistent mutants is $S = k/N$, by Eq. (1). The new score computed with both consistent and inconsistent mutants, S', has the total number of mutants increased by the number of consistent mutants, N_c. If N_c is some fraction of the number of inconsistent mutants, in other words, $N_c = \alpha N$, we have

$$S' = \frac{k}{N + N_c} = \frac{k}{N + \alpha N} = \frac{1}{1+\alpha}\frac{k}{N} = \frac{1}{1+\alpha}S$$

For mutation analysis we find α to be about 50%, so scores would be reduced by about one third in the context of a model checker. Consistent mutants are easily detected by comparing them with the complete state machine specification in a single run of the model checker.

The next type of mutants to winnow is trivial mutants, that is, mutants which evaluate to false in all conditions. For instance, a "replace variable" mutation operator might replace a variable, p, with another variable, q, which is the semantic negation of p. The resulting mutant, say, `AG (p <-> q)`, is always false or inconsistent, and therefore is satisfied by any test. Leaving such mutants inflates the score. If the number of trivial mutants is some fraction, β, of the number of inconsistent mutants, the score is increased by $\frac{\beta}{1+\beta}(1-S)$, where S is the score without trivial mutants. Formally, the new score, S', has both the number of mutants killed and the total number of mutants increased by the number of trivial mutants, N_t. If $N_t = \beta N$, then by Eq. (1)

$$\begin{aligned} S' &= \frac{k+N_t}{N+N_t} = \frac{k+\beta N}{N+\beta N} = \frac{k+\beta N+\beta k-\beta k}{(1+\beta)N} \\ &= \frac{(1+\beta)k+\beta(N-k)}{(1+\beta)N} = \frac{(1+\beta)k}{(1+\beta)N} + \frac{\beta(N-k)}{(1+\beta)N} \\ &= \frac{k}{N} + \frac{\beta}{1+\beta}\frac{N-k}{N} = \frac{k}{N} + \frac{\beta}{1+\beta}(1-\frac{k}{N}) \\ &= S + \frac{\beta}{1+\beta}(1-S) \end{aligned}$$

Trivial mutants may be detected by comparing all mutants and their negations with the original state machine specification in a single run of the model checker. Suppose a mutant `AG (P)` is always false. The negation, `AG (!P)`, is always true. Thus a pair where `AG (P)` is inconsistent and `AG (!P)` is consistent indicates that `AG (P)` is false. We found β to be about 7%, so a score of 80% is increased to 81%. Since the number of trivial mutants is low, we do not winnow them.

Winnowing duplicate mutants

Finally, we may winnow semantic duplicates, that is, mutants that evaluate to the same value for all possible tests.[e] For instance, suppose the mutation operators "negate expression" and "replace operator" 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 mutants, instead of removing all but one copy, adds more weight to the duplicates. In the extreme, suppose we generate 100 semantically distinct mutants, but one of those mutants occurs 201 times. A test set that kills the 99 unduplicated mutants but does not kill the duplicated mutant gets a score of 99/300 = 33%. However, a test set that only kills the duplicated mutant gets a much higher score of 201/300 = 67%!

We use the approach of Hagwood and Yen[21] to explore the effect of duplicates on the score by examining how the measured score, S', differs from the score if there were no duplicates, S. We begin by defining the vector $r_1, r_2, \ldots, r_N$ as the number

[e]The semantics of mutants is interpreted over the original state machine, not over all possible interpretations. Thus mutants which differ only on infeasible conditions are still considered to be the same semantically.

of mutants with each semantics, where N is the total number of semantically distinct mutants. That is, $r_i = 1$ means mutant i occurs once (no duplicate), $r_i = 2$ means mutant i occurs twice, etc. For $i = 1, 2, \ldots, N$, let

$$X_i = \begin{cases} 1 & \text{if the } i^{th} \text{ mutant is killed by the test set} \\ 0 & \text{otherwise} \end{cases}$$

We assume that the X_i are independent and all have the same probability of being killed, $p = Pr[X_i = 1]$. Note that the expected values of the scores are $E[S] = E[S'] = p$ which are the same! How then can we tell if some distribution of duplicates significantly distorts the score?

Let the total number of mutants be $n = \sum_{i=1}^{N} r_i$. The total number of mutants killed is $\sum_{i=1}^{N} r_i X_i$. Given these and Eq. (1), the scores may be written as

$$\begin{aligned} S &= \frac{\sum_{i=1}^{N} X_i}{N} &= \sum_{i=1}^{N} \frac{1}{N} X_i \\ S' &= \frac{\sum_{i=1}^{N} r_i X_i}{n} &= \sum_{i=1}^{N} \frac{r_i}{n} X_i \end{aligned}$$

A measure of the difference between the estimated score and the observed score is $\rho(S, S') = E[(S - S')^2]$ where E is the expected value. Evaluating this gives

$$\begin{aligned} \rho(S, S') &= E[(\sum_{i=1}^{N} \frac{1}{N} X_i - \sum_{i=1}^{N} \frac{r_i}{n} X_i)^2] \\ &= E[(\sum_{i=1}^{N} (\frac{1}{N} - \frac{r_i}{n}) X_i)^2] \\ &= Var[\sum_{i=1}^{N} (\frac{1}{N} - \frac{r_i}{n}) X_i] + (E[\sum_{i=1}^{N} (\frac{1}{N} - \frac{r_i}{n}) X_i])^2 \\ &= p(1-p) \sum_{i=1}^{N} (\frac{1}{N} - \frac{r_i}{n})^2 \\ &= p(1-p) D(\mu, \mathbf{r}) \end{aligned}$$

where $D(\mu, \mathbf{r})$ is the Euclidean distance between the discrete uniform vector $\mu = \{1/N, \ldots, 1/N\}$ and the probability vector $\mathbf{r} = \{r_1/n, r_2/n, \ldots, r_N/n\}$.

Many combinations of values of $\{r_1, r_2, \ldots, r_N\}$ give the same value of $D(\mu, \mathbf{r})$. Therefore it makes sense to talk about $D(\mu, \mathbf{r})$ as a distribution of values that give rise to a specific value of $D(\mu, \mathbf{r})$. To a good approximation

$$X^2 = \frac{N}{n} \sum_{i=1}^{n} (\frac{1}{N} - \frac{r_i}{n})^2$$

has a chi-squared distribution with $N-1$ degrees of freedom. Using the chi-squared test one can say that $\mathbf{r} = \{r_1/n, r_2/n, \ldots, r_N/n\}$ is significantly different from

$\{1/N, \ldots, 1/N\}$ if

$$\sum_{i=1}^{n} \left(\frac{1}{N} - \frac{r_i}{n}\right)^2 > \frac{1}{nN}\chi(N, 1-\alpha) \tag{2}$$

where $\chi(N, 1-\alpha)$ is the $1-\alpha$ quantile of the chi-squared distribution with N degrees of freedom. That is the scores are statistically different, and thus the distribution of duplicates is significantly nonuniform, if Eq. (2) holds. Individually, if for $i = 1, \ldots, N$

$$\left|\frac{n}{N} - r_i\right| \leq \sqrt{\frac{1}{N}\chi(N, 1-\alpha)}$$

then the scores are statistically equivalent, and duplicates will not significantly change the score.

Duplicate mutants may be detected by essentially comparing every mutant against every other mutant. For instance, if we have mutants `AG m1`, `AG m2` and `AG m3`, check the following.

```
AG (m1 <-> m2)
AG (m1 <-> m3)
AG (m2 <-> m3)
```

Only duplicates are consistent. In practise, we can reduce the number of comparisons by running a few checks to quickly determine mutants that are not duplicates, then comparing possibly-duplicate mutants with each other. We have not characterized the number or distribution of duplicate mutants.

4. Case Study

We applied our method to the Safety Injection problem. See App. 1 for the complete SMV specification. Table 1 is a higher-level specification with three tables. Each line in the mode transition table specifies one condition when the mode changes. If the mode matches the guard in the leftmost column and the condition in the middle, event column becomes true, the new mode will be that given in the rightmost column. Otherwise the mode stays the same. The next two tables give the guard, or mode, for each row. If an event occurs or a condition holds, the variable takes the value given at the bottom of the column. Otherwise the variable is unchanged.

We used three progressively more elaborate preparation methods: the reflected specification, the reflected specification with expounding, and the reflected specification with expounding and TRANS clauses. The expounded clauses were shortened. The appendix shows the specification resulting from the third, most elaborate method. The specification resulting from the first method may be obtained from that in the appendix by dropping all the SPEC clauses tagged with an "e" or reflected from the TRANS relation.

Using the method in Ammann, Black, and Majurski[3] we automatically generated test sets from all three versions of the specification. For comparison, we also manually produced a minimal test set that covered the tables expressed in Table 1. We explain the notion of "table coverage" below.

Table 1: Safety injection tables

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 $Safety\ Injection$.

4.1. *Mutation generation*

We used Vadim Okun's mutation engine with the following operators. We illustrate each operator with a mutant it generates from the following clause. Changes are emphasized by underlining.

```
    AG (Pressure = TooLow & Reset = Off ->
AX (Reset = On -> !Overridden))
```

1. replace_constant: replace one constant with another, for example,

```
    AG (Pressure = High & Reset = Off ->
AX (Reset = On -> !Overridden))
```

2. replace_oper: replace one operator with another operator, for example, replace "and" with "or"

```
    AG (Pressure = TooLow | Reset = Off ->
AX (Reset = On -> !Overridden))
```

3. replace_vars: replace a variable with another variable, for example,

```
    AG (Pressure = TooLow & Block = Off ->
AX (Reset = On -> !Overridden))
```

4. remove_expr: remove a simple expression from conjunctions, disjunctions, and implications, for example,

```
    AG (Pressure = TooLow __ ->
AX (Reset = On -> !Overridden))
```

Table 2: Mutations before and after expounding

	No Expound	Expound no TRANS	Expound with TRANS
Mutant SPEC clauses	188	611	1131
Inconsistent SPEC mutants[f]	121	388	824
Test cases	16	27	36
Test cases after minimizing	10	16	18

The only winnowing we do is to exclude consistent mutants. Table 2 shows the results of applying mutation generation before expounding, after expounding without reflecting the TRANS clause, and after expounding and reflecting the TRANS clause. The table also shows the number of test cases automatically generated by the method in Ammann, Black, and Majurski[3] for each version.

We can use our mutation analysis method to minimize test sets. We analyzed the mutants killed by different test cases and found a smaller set which has the same coverage. The bottom row of the table shows the number of test cases after this minimization. Interestingly, even though the number of mutants grows enormously with expounding and consideration of the TRANS clause, the number of test cases grows relatively modestly.

4.2. *Evaluation of separately developed test sets*

Consider the SCR tables, shown in Table 1, that gave rise to the SMV model. These tables are reproduced from Bharadwaj and Heitmeyer.[9] For comparison, let us construct a test set that satisfies the following criteria: each row in a mode transition table is covered by one or more test cases. In addition, for each mode, the possibility of remaining in that mode is covered by one or more test cases. In an event table, the conditions that cause each event are forced to be true and to be false, if possible, on one or more test cases each. Finally, in a condition table, each condition is forced to be true and to be false, if possible, on at least one test case each. We call this metric *table coverage*. We define any test set that satisfies these criteria to be *table adequate*.

We produced a test set that satisfied the criteria for table coverage.[g] We build a table adequate test set by picking a subset of the test cases generated for the unexpounded version of the safety injection problem. This turned out to be sufficient; otherwise, we could have produced additional tests. The result was four test cases.

Table 3 shows the result of evaluating the table adequate test set and the three automatically generated test sets against the table coverage metric and also

[f] We found that for this example, all inconsistent mutants are falsifiable.

[g] We used SMV to check for table adequacy as follows. For each event in the transition table, a SPEC clause was written stating that the desired transition did *not* happen when the specified event occurred. SMV then generated a counterexample showing that the desired transition did happen if the event occurred. A similar strategy was applied to the event and condition tables. The result was eight SPEC clauses for the mode transition table (two for each row), six for the event table (two for each event except $False$), and two for the condition table (all values for $Overridden$ in mode $TooLow$), for a total of 16 SPEC clauses. That is, a table adequate test set for the tables in Table 1 must satisfy these 16 elements.

Table 3: Test set coverage scores with different criteria

	tests	table coverage 16 elements	No Expound 121 mutants	Expound no TRANS 388 mutants	Expound with TRANS 824 mutants
table adequate	4	100% (16)	89% (108)	77% (301)	70% (577)
No Expound	16	100% (16)	100% (121)	81% (317)	76% (629)
Expound no TRANS	27	100%†	100%†	100% (388)	93% (767)
Expound with TRANS	36	100%†	100%†	100%†	100% (824)

against our metric with mutant generation with no expounding, expounding without TRANS, and expounding with TRANS. The scores marked with † are derived rather than measured. We justify the derived scores because the mutants that define adequacy for the second test set are a subset of those that define the third, and the mutants that define adequacy for the third test set are a subset of those that define the fourth. The 16 tests generated from No Expound scored 317/388 or 81% on mutants from Expound, but no reflection of TRANS clauses. Similarly, the 27 tests generated from Expound, No TRANS scored 767/824 or 93% on mutants from Expound with TRANS. Therefore, the additional mutants from reflection of the TRANS clause and expounding result in a more precise metric.

Although we would not expect table coverage testing of the SCR table to be as thorough as the mutation scheme used in this paper, it is still instructive to evaluate such tests with respect to mutation adequacy. It also demonstrates how our method can be used to evaluate test sets developed by other means. This is important because most existing software systems have large regression test sets associated with them, and it is very useful to analyze such test sets for gaps and redundancy. In the former case, additional tests can be added. In the latter case, redundant tests can be analyzed for removal.

All the test sets generated from mutation analysis turn out to be table coverage adequate. However, we note that a significant number of mutants are not necessarily detected by the table coverage adequate test set. This suggests that our specification-based mutation adequacy coverage metric has practical utility.

5. Summary and Conclusion

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 also to increase confidence in the software through formal metrics for test thoroughness. We pursued this path by applying model checking and mutation analysis to the problem of test set coverage. The resulting coverage metric can be used independently of source code, and is appropriate for "black box" testing.

From an assurance perspective, we would like to say that a test set guarantees some property for the software, but, with some exceptions, this goal is beyond the limits of what testing can show. Instead, test metrics are developed to capture desirable properties of a test set. Most of the testing metrics available in the literature are defined at the unit or source code level. We balance this by offering a metric at the software system level.

In this paper, we developed a metric for evaluating test sets against state transition specifications in the context of model checking. The metric is based on mutation analysis. Mutation adequate test sets are sensitive to the precise structure of the artifact from which the mutations are generated, which in this case is a model checking specification. We developed the necessary foundation for defining the mutation metric, including the roles of reflection, expounding, mutation operators, and winnowing procedures. We showed how to take a set of externally developed test cases, turn each test case into a constrained finite state machine, and score the set against the metric.

Scalability is a concern for all realistic software engineering techniques. The scalability of our technique depends in part on how well model checkers handle large specifications. The successes of SPIN[26] and SMV[12] suggest that a specification-based test coverage metric may apply to a broad variety of software systems.

To evaluate scalability, we plan to apply this technique to a much larger and richer specification, namely the generic Flight Guidance System developed by Rockwell Collins for the academic community and available in a variety of forms.[30] We also plan to explore starting with higher level specifications, say UML or state charts, and automatically generating model checker specifications. Additionally, we will devise other mutation operators and determine which set of mutation operators give the best coverage with the smallest set of tests. At the same time, we will theoretically and experimentally investigate the impact of duplicate mutants on our metric.

Acknowledgments

It is our pleasure to thank Angelo Gargantini and Connie Heitmeyer of the Naval Research Laboratory for supplying the SMV specifications for Safety Injection and for many helpful exchanges of data and ideas. We also thank Vadim Okun of the University of Maryland, Baltimore County, for the use of his mutation engine `mute`.

References

1. S. B. Akers, "On a theory of boolean functions", *SIAM Journal*, 7(4), 1959.
2. P. Ammann and P. E. Black, "Abstracting formal specifications to generate software tests via model checking", NIST IR 6405, 1999.
3. P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications", in *Proc. Second IEEE Int. Conference on Formal Eng. Methods* (IEEE Computer Society, 1998), pp. 46–54.
4. J. M. Atlee, "Native model-checking of SCR requirements", in *Proc. Fourth Int. SCR Workshop*, November 1994.
5. J. M. Atlee and M. A. Buckley, "A logic-model semantics for SCR software requirements", in *Proc. 1996 Int. Symp. on Software Testing and Analysis*, January 1996, pp.

280–292.

6. J. M. Atlee and J. Gannon, "State-based model checking of event-driven system requirements", *IEEE Tran. on Software Eng.*, 19(1), 1993, pp. 24–40.
7. J. Beskin, "Human error simulation in test case generation", Technical report, Reliable Software Technologies Corp., Sterling, Virginia, 1998.
8. R. Bharadwaj and C. Heitmeyer, "Verifying SCR requirements specifications using state exploration", in *Proc. First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.
9. R. Bharadwaj and C. L. Heitmeyer, "Model checking complete requirements specifications using abstraction", Memorandum Report NRL/MR/5540-97-7999, U.S. Naval Research Laboratory, Washington, DC, November 1997.
10. P. E. Black, V. Okun, and Y. Yesha, "Mutation of model checker specifications for test generation and evaluation", in *Proc. Mutation 2000*, October 2000.
11. J. R. Burch, E. M. Clarke, Jr., K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond", *Information and Computation*, 98(2), 1992, pp. 142–170.
12. W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese, "Model checking large software specifications", *IEEE Tran. on Software Eng.*, 24(7), 1998, pp. 498–520.
13. T. S. Chow, "Testing software design modeled by finite-state machines", *IEEE Tran. on Software Eng.*, SE-4(3), 1978, pp. 178–187.
14. E. M. Clarke, Jr., O. Grumberg, and D. 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).
15. 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), 1978, pp. 34–41.
16. R. A. De Millo and A. J. Offutt, "Constraint-based automatic test data generation", *IEEE Tran. on Software Eng.*, 17(9), 1991, pp. 900–910.
17. A. Engels, L. Feijs, and S. Mauw, "Test generation for intelligent networks using model checking", in *Proc. Third Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, ed. E. Brinksma, volume 1217 of *Lecture Notes in Computer Science* (Springer-Verlag, 1997), pp. 384–398.
18. S. R. Faulk, L. Finneran, J. Kirby, Jr., S. Shah, and J. Sutton, "Experience applying the CoRE method to the Lockheed C-130J", in *Proc. 9th Annual Conference on Computer Assurance*, June 1994, pp. 3–8.
19. P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria", *IEEE Tran. on Software Eng.*, 14(10), 1988, pp. 1483–1498.
20. A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications", in *Proc. Joint 7th European Software Eng. Conference and 7th ACM SIGSOFT Int. Symp. on Foundations of Software Eng.*, September 1999.
21. C. Hagwood and J. Yen, June 1999. Personal Communication.
22. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications", *ACM Tran. on Software Eng. and Methodology*, 5(3), 1996, pp. 231–261.
23. C. L. Heitmeyer and J. Mclean, "Abstract requirements specifications: A new approach and its application", *IEEE Tran. on Software Eng.*, SE-9(5), 1983, pp. 580–589.
24. K. L. Heninger, "Specifying software requirements for complex systems", *IEEE Tran. on Software Eng.*, SE-6(1), 1980, pp. 2–13.
25. D. Hoffman, P. Strooper, and L. White, "Boundary values and automated component testing", *Software Testing, Verification & Reliability*, 9(1), 1999.
26. G. J. Holzmann, "The model checker SPIN", *IEEE Tran. on Software Eng.*, 23(5),

1997, pp. 279–295.
27. W. E. Howden, "Weak mutation testing and completeness of test sets", *IEEE Tran. on Software Eng.*, 8(4), 1982, pp. 371–379.
28. D. R. Kuhn, "Fault classes and error detection in specification based testing", *ACM Tran. on Software Eng. Methodology*, 8(4), 1999.
29. P. M. Maurer, "Generating test data with enhanced context-free grammars", *IEEE Software*, July 1990, pp. 50–55.
30. 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*, March 1998.
31. S. Owre, J. M. Rushby, and N. Shankar, "Analyzing tabular and state-transition requirements specifications in PVS", Technical Report CSL-95-12, Computer Science Laboratory SRI International, June 1995. Revised April, 1996.
32. C. R. Ramakrishnan and R. Sekar, "Model-based vulnerability analysis of computer systems", in *2nd Int. Workshop on Verification, Model Checking and Abstract Interpretation*, September 1998.
33. R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities", in *Proc. 2000 IEEE Computer Society Symp. on Security and Privacy*, May 2000.
34. A. J. van Schouwen, D. L. Parnas, and J. Madey, "Documentation of requirements for computer systems", in *Proc. IEEE Int. Symp. on Requirements Eng.* (IEEE Computer Society Press, 1993), pp. 198–207.
35. J. M. Voas, "PIE: A dynamic failure-based technique", *IEEE Tran. on Software Eng.*, 18(8), 1992.
36. E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a boolean specification", *IEEE Tran. on Software Eng.*, 20(5), 1994, pp. 353–363.
37. M. R. Woodward, "Errors in algebraic specifications and an experimental mutation testing tool", *Software Eng. Journal*, July 1993, pp. 211–224.
38. D. Wu, M. A. Hennell, D. Hedley, and I. J. Riddell, "A practical method for software quality control via program mutation", in *Proc. 2nd Workshop on Software Testing, Verification and Analysis*, July 1988, pp. 159–170.
39. H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy", *ACM Computing Surveys*, 29(4), 1997, pp. 366–427.

Appendix A

Safety Injection Problem

The specification given below is a modification of one supplied by the Navy Research Laboratory. The specification corresponds to Table 1. See Bharadwaj and Heitmeyer[9] for a closely related specification in all of SCR, SPIN, and SMV.

The numerical comments to the right of case statement branches below provide a cross reference between the transition relation and the reflection into SPEC clauses. Branches marked with an "e" require expounding prior to reflection. After expounding, it is often convenient to use multiple SPEC clauses for reflection; hence the multiple SPEC clauses for each of the "e" branches. In this example, only the default cases require expounding.

```
MODULE main

VAR
Reset : {On, Off};
Overridden : {0,1}; --boolean
Block : {On, Off};
WaterPres : 0..200;
Pressure : {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
init(Block):= Off;
init(Reset):= On;
init(WaterPres):= 2;
init(Overridden):= 0;
init(Pressure):= TooLow;

next(Block):= {On, Off};
next(Reset):= {On, Off};
next(WaterPres):= 0..200;

next(Overridden):= case
  Pressure = TooLow: case
    !(Pressure = next(Pressure) ) :0;                                   --  1
    !(Reset = On) & next(Reset) = On :0;                                --  2
    !(Block = On) & next(Block) = On  & Reset = Off:1;                  --  3
    1:Overridden;                                                       --  4e
  esac;
  Pressure = Permitted: case
    !(Pressure = next(Pressure) ) :0;                                   --  5
    !(Reset = On) & next(Reset) = On :0;                                --  6
    !(Block = On) & next(Block) = On  & Reset = Off:1;                  --  7
    1:Overridden;                                                       --  8e
  esac;
  Pressure = High: case
    !(Pressure = next(Pressure) ) :0;                                   --  9
    1:Overridden;                                                       -- 10e
  esac;
esac;

next(Pressure):= case
  Pressure = TooLow: case
    !(( WaterPres >= Low )) & ( next(WaterPres) >= Low ) :Permitted;  -- 11
    1:Pressure;                                                       -- 12e
  esac;
  Pressure = Permitted: case
    !(( WaterPres < Low )) & ( next(WaterPres) < Low ) :TooLow;       -- 13
    !(( WaterPres >= Permit )) & ( next(WaterPres) >= Permit ) :High; -- 14
    1:Pressure;                                                       -- 15e
  esac;
  Pressure = High: case
    !(WaterPres < Permit) & ( next(WaterPres) < Permit ) :Permitted;  -- 16
    1:Pressure;                                                       -- 17e
  esac;
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 following SPEC clauses reflect the logic of the transition
-- relation expressed above.
SPEC -- 1
  AG(Pressure=TooLow -> AX(!(Pressure=TooLow) -> !Overridden))
SPEC -- 2
  AG(Pressure=TooLow & Reset=Off -> AX(Reset=On -> !Overridden))
```

```
SPEC -- 3
  AG(Pressure=TooLow & Block=Off & Reset=Off -> AX(Block=On -> Overridden))
SPEC -- 4e:1
  AG(Pressure=TooLow & Reset=On & Overridden -> AX(Pressure=TooLow ->
                                                          Overridden))
SPEC -- 4e:2
  AG(Pressure=TooLow & Reset=On & !Overridden -> AX(Pressure=TooLow ->
                                                          !Overridden))
SPEC -- 4e:3
  AG(Pressure=TooLow & Block=On & Overridden -> AX(Pressure=TooLow &
                                                 Reset=Off -> Overridden))
SPEC -- 4e:4
  AG(Pressure=TooLow & Block=On & !Overridden -> AX(Pressure=TooLow &
                                                 Reset=Off -> !Overridden))
SPEC -- 4e:5
  AG(Pressure=TooLow & Overridden -> AX(Pressure=TooLow & Block=Off &
                                                 Reset=Off -> Overridden))
SPEC -- 4e:6
  AG(Pressure=TooLow & !Overridden -> AX(Pressure=TooLow & Block=Off &
                                                 Reset=Off -> !Overridden))
SPEC -- 5
  AG(Pressure=Permitted -> AX(!(Pressure=Permitted) -> !Overridden))
SPEC -- 6
  AG(Pressure=Permitted & Reset=Off -> AX(Reset=On -> !Overridden))
SPEC -- 7
  AG(Pressure=Permitted & Block=Off & Reset=Off -> AX(Block=On ->
                                                 Overridden))
SPEC -- 8e:1
  AG(Pressure=Permitted & Reset=On & Overridden -> AX(Pressure=Permitted ->
                                                 Overridden))
SPEC -- 8e:2
  AG(Pressure=Permitted & Reset=On & !Overridden -> AX(Pressure=Permitted ->
                                                 !Overridden))
SPEC -- 8e:3
  AG(Pressure=Permitted & Block=On & Overridden -> AX(Pressure=Permitted &
                                                 Reset=Off -> Overridden))
SPEC -- 8e:4
  AG(Pressure=Permitted & Block=On & !Overridden -> AX(Pressure=Permitted &
                                                 Reset=Off -> !Overridden))
SPEC -- 8e:5
  AG(Pressure=Permitted & Overridden -> AX(Pressure=Permitted & Block=Off &
                                                 Reset=Off -> Overridden))
SPEC -- 8e:6
  AG(Pressure=Permitted & !Overridden -> AX(Pressure=Permitted & Block=Off &
                                                 Reset=Off -> !Overridden))
SPEC -- 9
  AG(Pressure=High -> AX(!(Pressure=High) -> !Overridden))
SPEC -- 10e:1
  AG(Pressure=High & Overridden -> AX(Pressure=High -> Overridden))
SPEC -- 10e:2
  AG(Pressure=High & !Overridden -> AX(Pressure=High -> !Overridden))
SPEC -- 11
  AG((Pressure=TooLow) & !(WaterPres >= Low) -> AX((WaterPres >= Low) ->
                                                 Pressure=Permitted))
SPEC -- 12e:1
  AG((Pressure=TooLow) & (WaterPres >= Low) -> AX(Pressure=TooLow))
SPEC -- 12e:2
  AG((Pressure=TooLow) -> AX(!(WaterPres >= Low) -> Pressure=TooLow))
SPEC -- 13
  AG((Pressure=Permitted) & !(WaterPres < Low) -> AX((WaterPres < Low) ->
                                                 Pressure=TooLow))
SPEC -- 14
  AG((Pressure=Permitted) & !(WaterPres >= Permit) ->
                                AX((WaterPres >= Permit) -> Pressure=High))
SPEC -- 15e:1
  AG((Pressure=Permitted) -> AX(!(WaterPres < Low)  & WaterPres < Permit ->
                                                 Pressure=Permitted))
SPEC -- 16
  AG((Pressure=High) & !(WaterPres < Permit) -> AX((WaterPres < Permit) ->
                                                 Pressure=Permitted))
SPEC -- 17e:1
  AG((Pressure=High) & (WaterPres < Permit) -> AX(Pressure=High))
SPEC -- 17e:2
  AG((Pressure=High) -> AX(!(WaterPres < Permit) -> Pressure=High))

-- The following SPEC clauses reflect (an abstraction of) the TRANS section

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

```
SPEC AG(Reset = Off & Pressure = Permit -> AX(Reset = On  ->
                                              Pressure = Permit))
SPEC AG(Reset = Off & Pressure = High   -> AX(Reset = On  ->
                                              Pressure = High))

SPEC AG(Reset = On  & Block = On        -> AX(Reset = Off -> Block = On))
SPEC AG(Reset = On  & Block = Off       -> AX(Reset = Off -> Block = Off))
SPEC AG(Reset = On  & Pressure = TooLow -> AX(Reset = Off ->
                                              Pressure = TooLow))
SPEC AG(Reset = On  & Pressure = Permit -> AX(Reset = Off ->
                                              Pressure = Permit))
SPEC AG(Reset = On  & Pressure = High   -> AX(Reset = Off ->
                                              Pressure = High))

SPEC AG(Block = Off & Reset = On        -> AX(Block = On  -> Reset = On))
SPEC AG(Block = Off & Reset = Off       -> AX(Block = On  -> Reset = Off))
SPEC AG(Block = Off & Pressure = TooLow -> AX(Block = On  ->
                                              Pressure = TooLow))
SPEC AG(Block = Off & Pressure = Permit -> AX(Block = On  ->
                                              Pressure = Permit))
SPEC AG(Block = Off & Pressure = High   -> AX(Block = On  ->
                                              Pressure = High))

SPEC AG(Block = On  & Reset = On        -> AX(Block = Off -> Reset = On))
SPEC AG(Block = On  & Reset = Off       -> AX(Block = Off -> Reset = Off))
SPEC AG(Block = On  & Pressure = TooLow -> AX(Block = Off ->
                                              Pressure = TooLow))
SPEC AG(Block = On  & Pressure = Permit -> AX(Block = Off ->
                                              Pressure = Permit))
SPEC AG(Block = On  & Pressure = High   -> AX(Block = Off ->
                                              Pressure = High))

SPEC AG(Pressure=TooLow & Reset=On  -> AX(!(Pressure=TooLow) -> Reset=On ))
SPEC AG(Pressure=TooLow & Reset=Off -> AX(!(Pressure=TooLow) -> Reset=Off))
SPEC AG(Pressure=TooLow & Block=On  -> AX(!(Pressure=TooLow) -> Block=On ))
SPEC AG(Pressure=TooLow & Block=Off -> AX(!(Pressure=TooLow) -> Block=Off))

SPEC AG(Pressure=Permit & Reset=On  -> AX(!(Pressure=Permit) -> Reset=On ))
SPEC AG(Pressure=Permit & Reset=Off -> AX(!(Pressure=Permit) -> Reset=Off))
SPEC AG(Pressure=Permit & Block=On  -> AX(!(Pressure=Permit) -> Block=On ))
SPEC AG(Pressure=Permit & Block=Off -> AX(!(Pressure=Permit) -> Block=Off))

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