Abstracting Formal Specifications to Generate Software Tests via Model Checking

Paul Ammann*[1] and Paul E. Black[2]

[1] Information & Software Engineering Dept., George Mason University,
Fairfax, Virginia 22033 USA
pammann@gmu.edu
[2] Information Technology Laboratory, NIST, Gaithersburg, Maryland 20899 USA
paul.black@nist.gov

Abstract. A recent method combines model checkers with specification-based mutation analysis to generate test cases from formal software specifications. However high-level software specifications usually must be reduced to make analysis with a model checker feasible.
We propose a new reduction, parts of which can be applied mechanically, to soundly reduce some large, even infinite, state machines to manageable pieces. Our work differs from other work in that we use the reduction for generating test sets, as opposed to the typical goal of analyzing for properties. Consequently, we have different criteria, and we prove a different soundness rule. Informally, the rule is that counterexamples from the model checker are test cases for the original specification. The reduction changes both the state machine and temporal logic constraints in the model checking specification to avoid generating unsound test cases. We give an example of the reduction and test generation.

1 Introduction

The use of formal methods has been widely advocated to reduce the likelihood of errors in early stages of system development. Some of the chief drawbacks to applying formal methods are the difficulty of conducting formal analysis [6] and the perceived or actual payoff in project budget. Testing is a substantial part of the software budget, and formal methods offer an opportunity to significantly reduce testing costs, thereby making formal methods more attractive from the budget perspective.

The authors developed an innovative combination of mutation analysis, symbolic model checking, and test generation which solves some problems previously plaguing these approaches and automatically produces good sets of tests from formal specifications [1, 2]. This approach is useful only if there is a specification amenable to model checking for a given application. Here we seek to widen the class of applications for which automatic test generation via a model checker is

* Partially supported by the National Science Foundation under grant number CCR-99-01030

feasible. Our approach is to define a reduction from a given specification to a smaller one that is more likely to be tractable for a model checker. We tailor the reduction for test generation, as opposed to the usual goal of analysis.

A broad span of research from early work on algebraic specifications [13] to more recent work such as [21] addresses the problem of relating tests to formal specifications. In particular, counterexamples from model checkers are potentially useful test cases. In addition to our use of the Symbolic Model Verifier (SMV) model checker [19] to generate mutation adequate tests [2], Callahan, Schneider, and Easterbrook use the Simple PROMELA Interpreter (SPIN) model checker [16] to generate tests that cover each block in a certain partitioning of the input domain [8]. Gargantini and Heitmeyer use both SPIN and SMV to generate branch-adequate tests from Software Cost Reduction (SCR) requirements specifications [14].

The model checking approach to formal methods has received considerable attention in the literature, and readily available tools such as SMV and SPIN are capable of handling the state spaces associated with realistic problems [11]. Although model checking began as a method for verifying hardware designs, there is growing evidence that model checking can be applied with considerable automation to specifications for relatively large software systems, such as the Traffic Alert & Collision Avoidance System (TCAS) II [9]. The increasing usefulness of model checkers for software systems makes them attractive targets for use in aspects of software development other than pure analysis, which is their primary role today.

Model checking has been successfully applied to a wide variety of practical problems, including hardware design, protocol analysis, operating systems, reactive system analysis, fault tolerance, and security. The chief advantage of model checking over the competing approach of theorem proving is complete automation. Whereas human interaction is generally required to prove all but the simplest theorems, model checkers can explore the state spaces for finite, yet realistic, problems without human guidance.

A model checking specification consists of two parts. One part is a state machine defined in terms of variables, initial values for the variables, environmental assumptions, and a description of the conditions under which variables may change value. The other part is temporal logic constraints over states and execution paths. Conceptually, a model checker visits all reachable states and verifies that the temporal logic properties are satisfied over each possible path. Model checkers exploit clever ways of avoiding brute force exploration of the state space, for example, see [7]. If a property is not satisfied, the model checker attempts to generate a counterexample in the form of a trace or sequence of states. For some temporal logic properties, no counterexample is possible. For example, if the property states that at least one possible execution path leads to a certain state and in fact no path leads to that state, there is no counterexample to exhibit.

Even though model checking are powerful formal “compute engines,” clever abstractions are required for problems of even modest complexity to avoid the state space explosion problem, which renders the model checker useless. Some of these abstractions are informal, although there have been significant formalizations of the abstraction process [4, 9, 18]. Other abstractions [20] are formalized to the extent of being paired with theorem provers and model checkers to calculate and refine them. These abstractions are directed at the analysis problem, that is, determining whether given properties expressed in a temporal logic hold over a given state machine.

In this paper, we focus on test generation instead of analysis, and therefore test requirements drive our abstraction. The basic property needed for test generation, which we term *reduction soundness*, is that if a counterexample is generated in the abstraction, the counterexample is also a test case in the original specification. Reduction soundness does not necessarily hold for abstractions designed for analysis, although other soundness properties may hold.

Our contributions in this paper are:

1. We provide a new reduction called *finite focus* for abstracting state machine specifications.
2. We develop a notion of soundness that is suitable for test generation, and we show that the finite focus reduction is sound.
3. We define a notion of mutation adequacy for mutation testing of model checking specifications, and describe the subset of mutants for which finite focus generates a mutation adequate test set. Informally, they are mutants that can be distinguished via traces from the finite focus reduction.

Section 2 is an overview of the mutation analysis approach for generating tests and measuring coverage with a model checker described in [1, 2], and Sect. 3 explains how the finite focus reduction fits into this approach. Section 4 formally defines the reduction and proves soundness properties. Although significant parts of the reduction are theoretically underconstrained, Sect. 5 describes additional considerations, particularly for mutation adequacy. Section 6 presents an example of using finite focus to generate tests for validation. Section 7 gives our plans for future work. Finally, we present our conclusions in Sect. 8. Appendix A is a full proof of the rewriting rules used in the reduction. Appendix B gives the model used in the example and an alternate model, and App. C gives the tests.

2 Automatic Test Generation

Figure 1 shows the overall approach explained in detail in [1, 2]. One begins with some system specifications and, through finite modeling and with the aid of automated tools, turns them into specifications suitable for a model checker. After this point all processing can be automatic.

2.1 Background on Mutation Analysis

Standard mutation analysis [12] is a method based on program source code to develop a set of test cases which is sensitive to small syntactic changes to a

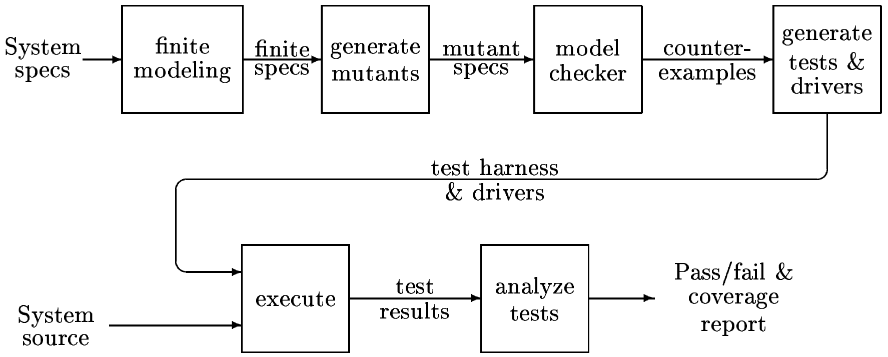

Fig. 1. Automatic Test Generation in [1]

program. The rationale is that if a test set can distinguish a program from each of its slight variations, the test set is exercising the program adequately.

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. Applying the set of operators systematically generates a set of mutant programs. Some of these mutants may be semantically equivalent to the original program. That is, a mutant and the original may compute the same function for all possible inputs. Such mutants are termed *equivalent*. Equivalent mutants present a serious problem for program-based mutation analysis, since identifying equivalent mutants is, in general, an undecidable problem.

2.2 Test Generation Via Mutation Operators

The specification-based mutation analysis scheme in [1, 2] is decidable since its domain is the finite state space of a model checker specification. To provide a richer set of constraints, the state machine specification is "reflected" in the constraints. For instance, a transition from state `s1` to `s2` on condition `c` becomes the constraint `SPEC AG s1 & c -> AX s2`. Mutation operators are applied to the constraints yielding a set of mutant specifications. A "condition substitute" operator yields `SPEC AG s1 & b -> AX s2`, among other mutant specifications, when applied to the above constraint. Other operators change constants, variables, or boolean operators, drop conditions, etc.

The model checker compares the (assumed-good) state machine with the mutants. When it finds an inconsistency, it generates a counterexample. Equivalent mutants produce no counterexamples and therefore are automatically disregarded.

The set of counterexamples is reduced by eliminating duplicates and counterexamples which are "prefixes" of other, longer counterexamples. The counterexamples contain both stimuli and expected output values so they may be automatically converted to complete test cases. The test cases generate executable test code, including a test harness and drivers. For a given set of mutation operators, the procedures in [1, 2] generate a mutation-adequate set of test cases.

3 Practical Test Generation

The preceding approach uses model checkers to process specifications. Unfortunately, symbolic model checkers can only handle finite state machines. In fact, spaces with more than a few thousand states must often be handled in special ways. Yet specifications of realistic software often have enormous, even infinite, state spaces.

It is often straight-forward for an analyst to come up with a smaller model if the original model is too large for the model checker. However, it is generally impractical to require large amounts of human time to devise smaller models. To leverage scarce human expertise, we want reductions and abstractions which are highly automated.

3.1 Other Reduction Approaches

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

In a sound abstraction, properties of the reduced or abstract specification are also properties of the original specification. Soundness avoids false positives. That is, any error found in the abstract specification (a "positive" result) is also an error in the original specification. In a complete abstraction properties of the original specification are also properties of the reduced specification. Completeness avoids false negatives. That is, all errors in the original specification will be found in the abstract specification.

Heitmeyer, et. al. [15] formalize an abstraction which removes irrelevant variables. Briefly, to check that some property q holds for a specification, one may remove variables and inputs which do not occur in or contribute to q. Another abstraction removes monitored or input variables which only contribute directly to one other variable. Finally, they also describe a method which abstracts monitored variables. That is, if only certain values or ranges of a monitored variable influence the values of other variables, the monitored variable may be replaced with an abstract variable. For instance, consider an input variable, `Water Pressure`, with a discrete range from 0 to 2000 whose influence is constant over low values, over a range of moderate values, and over high values. This variable may be replaced with a quantized variable which has the values Too_Low, In_Range, or Too_High. All these abstractions are sound for analysis. The first is complete, and the other two are complete under conditions which frequently hold in practise.

Chan, et. al. [9] use another method to reduce the state size. Some specifications place time bounds on the intervals between events. The obvious specification keeps time as an integer, uses variables to record the times of events, and has predicates on the difference in times. Instead, they keep (bounded) timers measuring the time since events. When the bounds are exceeded, the timers enter a "satisfied" (or "unsatisfied") state.

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

Kurshan [18] explains how k verifications may be done on k reductions of a system, each of which is a $\frac{n}{k}$ part of the entire system. Since verification is often exponential in the size of the system, a verification of the entire system may be proportional to c^n while k verifications take $kc^{\frac{n}{k}}$ work.

In an overview presentation, Rushby [20] advocates "ubiquitous abstraction," that is, using abstractions in several different ways in all parts of analysis. For instance, even for one given problem, different abstractions may be appropriate, depending on the invariants used to prove a goal. The invariants may be automatically strengthened when the proof fails. Another noteworthy approach is calculating transitions of a state abstraction using rules that guarantee correctness, as opposed to taking a hand-crafted abstraction and proving it is sound and complete. Abstractions may be refined automatically using information from static analysis, such as reachable states. In contrast, we are still at the stage of characterizing abstractions in our work, albeit for a different notion of soundness, rather than computing them.

Bensalem, Lakhnech, and Owre [3] explain a semi-automated abstraction in which the analyst chooses a state abstraction and then a conservative (sound) set of corresponding transitions are computed. Construction begins with a complete set of transitions, that is, a transition from every abstract state to every other abstract state. If a transition can be (automatically) proven to be impossible, it is removed. Since such proofs are in general too complex, they combine it with three techniques based on partitioning the abstract variables, substituting, and using the property being investigated.

3.2 A New Reduction

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

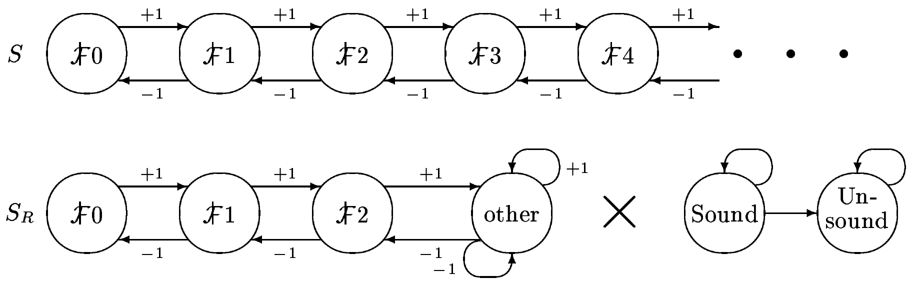

Fig. 2. Focus on a Finite Subset of States

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

Consider the example of a bank balance in an imaginary currency, the Florin (ƒ), with operations to deposit and withdraw one Florin. The complete model, depicted by the top row and labeled S in Fig. 2, uses type `natural`. However, the model cannot be automatically examined by a model checker. To use the analytical resources of a model checker, we must drastically reduce this model to some finite size. For instance, a human analyst may naturally focus on what happens when the balance is close to zero and ignore, for the moment, large balances. Can we formalize this focusing on a subrange so that the analyst need not worry about making an unsound reduction?

Suppose we choose to accurately model balances of ƒ0, ƒ1, and ƒ2, and map anything greater than two to "other." We need to indicate that the model checker should ignore any set of operations in which the balance exceeds ƒ2, since they may not be sound, i.e., may not be accurately represented.

Consider having one constraint on accounts with a balance of ƒ3 and a different constraint on those with a balance of ƒ4. Both of these balances are mapped to "other" in the reduction. This loss of accuracy indicates that any execution path entering "other" is suspect. We record this by adding a "soundness" state variable which becomes *unsound* if the state becomes "other," such as a deposit when the balance is ƒ2. The bottom row, labeled S_R in Fig. 2, illustrates this reduction. We can then have the model checker ignore any unsound inconsistencies so that it returns only those which are problems in the full model. We formalize the idea behind this example as a reduction we call "finite focus."

3.3 Finite Focus and Test Generation

For our purposes, a system specification is a pair (S, T), where S is a state machine description and T is a set of temporal logic constraints. S may be unbounded, for instance, part of the state may be an integer.

To generate test cases using the method described in Sect. 2, we must be able to analyze the specifications with a symbolic model checker. Figure 3 illustrates the steps to apply the reduction for finite focus or RFF. For test case generation, the state machine, S, is reflected as temporal logic constraints to provide a description for subsequent mutation analysis. Any existing temporal logic constraints, τ in the figure, may be added to the reflected constraints which describe the state machine.

Some finite number of states, focused around the initial state, are mapped to states in the reduced specification. All other states are mapped to a single "other" state. The source and destination of each transition are mapped likewise. The function RFF_T maps temporal logic constraints, and RFF_S maps the state machine. The two functions, along with constraint rewriting (CR) for soundness, explained below, constitute RFF.

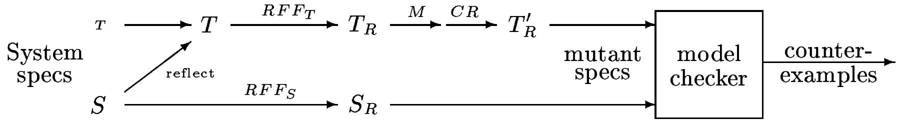

Fig. 3. Specification Transformations

RFF_S also adds a separate state machine with the initial state *sound*. Whenever the reduced state machine ends in the "other" state, this added machine goes *unsound*. It remains *unsound* thereafter. This step yields a reduced state machine, S_R.

RFF_T yields reduced temporal logic constraints, T_R, but is less rigidly determined than RFF_S. Together S_R and T_R answer to the finite specifications of Fig. 1.

To generate counterexamples, we repeatedly apply various mutation operators, M in Fig. 3, to the temporal logic constraints. Then, in order to prevent unsound counterexamples, we rewrite the constraints so they are always satisfied when the state is *unsound*. This constraint rewriting, CR, yields mutated constraints, T'_R. Together S_R and T'_R are given to the model checker which efficiently computes a number of counterexamples. We prove the soundness of our reduction in Sect. 4 as the central result of this paper. Soundness for test generation means that any counterexample of the reduced specification (S_R, T'_R) is a valid trace of the original state machine specification, S.

3.4 Preventing Unsound Counterexamples

Along with the reductions RFF_T and RFF_S, we must make sure no counterexamples are produced if the state becomes unsound. We can prevent counterexamples by modifying the temporal logic specifications so there is no violation if the model is unsound. Here we give a set of rewriting rules which force any temporal logic expression to evaluate to true (or false, as appropriate) if the state is *unsound.*

Let s be the variable representing soundness: if the state is sound, s is true. If the machine takes an unsound transition, s becomes false. Note that once s becomes false, it remains false; RFF soundness relies on this.

Specifications in computation tree logic (CTL) [10] are changed according to the following rules, which we refer to as the constraint rewriting rules, CR. If a CTL formula does not begin with a temporal operator, it is rewritten as an implication so that it has the value $True$ when the soundness variable is false. Otherwise, the temporal operator rewriting rule is applied.

$$CR(f) = \begin{cases} cr(f, True) & \text{if } f \text{ begins with a temporal operator} \\ s \mathrel{-\!\!>} cr(f, True) & \text{otherwise} \end{cases}$$

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

$$\begin{aligned}
&cr(!\, f, v) &&= !\, cr(f, {\sim}v) \\
&cr(f \,\&\, g, v) &&= cr(f, v) \,\&\, cr(g, v) \\
&cr(f \mid g, v) &&= cr(f, v) \mid cr(g, v) \\
&cr(f \mathrel{-\!\!>} g, v) &&= cr(f, {\sim}v) \mathrel{-\!\!>} cr(g, v) \\
&cr(f \mathrel{<\!\!-\!\!>} g, v) &&= cr(f, {\sim}v) \mathrel{-\!\!>} cr(g, v) \,\&\, cr(g, {\sim}v) \mathrel{-\!\!>} cr(f, v)
\end{aligned}$$

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

$$\begin{aligned}
cr(\mathtt{OP}\ f, True) &= \mathtt{OP}\ s \mathrel{-\!\!>} cr(f, True) \\
cr(\mathtt{OP}\ f, False) &= \mathtt{OP}\ s \& cr(f, False)
\end{aligned}$$

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

$$\begin{aligned}
cr(\mathtt{OP} g \mathtt{U} f, True) &= \mathtt{OP} g \mathtt{U} s \mathrel{-\!\!>} cr(f, True) \\
cr(\mathtt{OP} g \mathtt{U} f, False) &= \mathtt{OP} g \mathtt{U} s \& cr(f, False)
\end{aligned}$$

If the formula is none of the above, say, a variable, it is unchanged.

$$cr(f, v) = f$$

For example, the following specification states that an instruction which pushes one item on the stack of a Java[1] virtual machine, increases the stack size by one in the next state.

[1] Java is a trademark of Sun Microsystems, Inc.

```
SPEC AG(instr=push1 -> AX(StkSize=PStkSize+1))
```

Rewriting for soundness yields the formula below which reads: if the current state is sound and the instruction pushes one item, the stack is one larger in the next state if it is (still) sound.

```
SPEC AG(s -> instr=push1 -> AX(s -> StkSize=PStkSize+1))
```

Our models and specifications for a Java virtual machine stack are in App. B.

We now want to argue a theorem that states, roughly, when s is false, every constraint evaluates to true. First we define a finite execution path of a state machine, over which constraints are evaluated. We begin with a reference definition of a state machine.

Definition 1 (after [4]). *A* state machine *S is a 4-tuple $(\Sigma, s_0, E^m, \mathcal{T})$, where Σ is a set of states, $s_0 \in \Sigma$ is the initial state, E^m is the set of input events, and $\mathcal{T}$ describes the state transitions. The transitions $\mathcal{T}$ are a relation $s \times e \to s$ where $s \in \Sigma$, and $e \in E^m$.*

Since $\mathcal{T}$ is a relation, this definition includes non-deterministic state machines. Input events correspond to monitored variables in [4, 5].

Definition 2 (after [10]). *A* finite path *is a finite sequence of states $(s_0, s_1, \ldots, s_n)$ such that $\forall i \mid 0 \leq i < n \Rightarrow (s_i, s_{i+1}) \in \mathcal{T}$.*

Definition 3. *A finite path is* irreducible *if after removing the last state, the path does not violate any constraint.*

That is, a finite path $(s_0, s_1, \ldots, s_n)$ is irreducible if $(s_0, s_1, \ldots, s_{n-1})$ does not violate any constraint.

Theorem 1. *Suppose that CR is applied to a set of constraints, T_R. Any irreducible path that violates a constraint in $CR(T_R)$ has s equal true in each state.*

We give a proof sketch here; see App. A for a more formal proof. We first argue the rules for universally quantified expressions. The AG rule describes invariants on states; clearly this rule exempts a particular state if s is false in that state. The AF rule describes a property of some future state; if s is false, it remains false in all future states, and therefore the property is true for all future states. The AX rule is a special case of the AF rule where the future state is simply the next state.

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

In case we need to rewrite the expression to be false, for instance, if the specification is `SPEC ! AG p`, the rewriting rules for AG, AF, and A . . . U force the value to be false when s is false. For AX we need the requirement on CTL structures that every state has at least one outgoing transition. Otherwise, if a state had no next state, AX False would be vacuously true.

The rules for rewriting existential quantifiers to be true or false follow similarly, except for rewriting EX expressions to be true. If a state had no next state, EX True would be false, since there is no next state at all. Since every state has a next state, the rewriting works in all cases.

4 Proof of Reduction Soundness

In this section we formally define the properties which any reduction for finite focus must have. We define soundness for our ultimate purpose of test generation and prove that counterexamples from finite focus reductions are sound.

Definition 4. *A* trace *of an execution is a list of inputs to a state machine and the resultant states. Formally* $t = [(e_1, s_1), \ldots, (e_n, s_n)]$ *is a trace of* S *if* $\forall i \mid 1 \leq i \leq n \Rightarrow \mathcal{T} : (s_{i-1}, e_i) \rightarrow s_i$.

Note that this definition allows non-deterministic state machines. However, any particular trace is completely unambiguous, even if the state machine is non-deterministic. That is, the particular sequence of transitions yielding a trace may be unambiguously reconstructed from the trace: each transition is $(s_{i-1}, e_i) \rightarrow s_i$ where $1 \leq i \leq n$.

Definition 5. *A* counterexample*, c, from a state machine,* S*, and temporal logic constraints,* T*, is an irreducible trace of* S *with a constraint violation of* T.

In other words, model checkers produce counterexamples just long enough to demonstrate some combination of inputs and states which cause one or more constraints to be false. If the model checker produced counterexamples longer than necessary, it may find a contradiction in a sound state, then arbitrarily continue the trace and eventually report an unsound state.

Some model checkers can generate counterexamples quite efficiently. The actual counterexamples often elide much of the redundant trace information, such as values which stay the same in a new state. When we say that a counterexample is produced from some S and T, we mean a trace which demonstrates a constraint violation is found and reported.

Definition 6. *A state machine reduction for finite focus,* RFF_S*, has the following properties:*

1. *The reduction accurately copies the initial state. It may also copy some finite number of states around the initial state.*
   (a) *The initial state is mapped one-to-one:* $s_0 \overset{RFF_S}{\longrightarrow} s_0$.
   (b) *Other states are mapped one-to-one, also.*
   (c) *Any remaining states are mapped to a new state, "other":* $s_i \notin \Sigma_R \Rightarrow s_i \overset{RFF_S}{\longrightarrow} other$.

2. *Input events map identically:* $E^m \xrightarrow{RFF_S} E^m_R$.
3. *The sources and destinations of transitions are mapped according to the above,* $\mathcal{T} : (s, e) \to s' \xrightarrow{RFF_S} \mathcal{T}_R : (RFF_S(s), e) \to RFF_S(s')$
4. *The reduction adds a new "soundness" variable with two states:* sound *and* unsound.
   (a) *The initial state is* sound.
   (b) *For every unsound transition* ($\mathcal{T}_R : (s, e) \to other$)*, add a transition conditioned on the source state and event from* sound *to* unsound*, more formally,* $(sound, s \times e) \to unsound$.
   (c) *Any other transition from* sound *remains in* sound.
   (d) *All transitions from* unsound *remain in* unsound.

The temporal logic reduction $T \xrightarrow{RFF_T} T_R$ is nearly unconstrained. Theoretically, soundness is preserved by *any* reduction, as long as the resulting constraints are valid (e.g., no undefined variables or constants) in the state machine, S_R. In Sect. 5 we describe practical requirements to achieve coverage.

Definition 7. *A state* s *in* S_R *is* sound *if it is faithfully copied to* S_R.

That is, if $s \in \Sigma \xrightarrow{RFF_S} s \in \Sigma_R$, s is *sound*.

Lemma 1. *Counterexamples include no unsound states.*

Proof: Assume there is a first unsound state in the counterexample trace.

1. There was no contradiction in a previous state, by Definition 5.
2. Since this state is unsound, it is not part of a contradiction, by Theorem 1.
3. Since subsequent states are also unsound (part 4d of the definition of RFF), they cannot be part of a contradiction, either.

So there is no contradiction at all. But this conflicts with the definition that counterexamples have contradictions. Therefore there is no first unsound state.

Complementing Lemma 1, we argue that all sound transitions from soundly reachable states may be used. Consider all sound states which are reachable from the initial state through other sound states. Any transition from a soundly reachable sound state to another sound state may appear in counterexamples.

To say which transitions actually appear in counterexamples would require us to characterize the state machine duplication, possible mutation operators, the temporal logic reduction function, and the model checker's counterexample selection scheme. We discuss a related issue, mutation adequacy, in Sect. 5.

Theorem 2. *No soundly reachable sound transition is necessarily excluded from counterexamples.*

Proof:

1. By definition the initial state is reachable. Since it is also sound, it is soundly reachable. Any sound state reachable from a soundly reachable state is soundly reachable, too.

2. Since the machine always remains sound, any sound transition from a soundly reachable state is not precluded by our scheme.

We now present our main result, namely that any reduction following the above is sound, or that it produces only *sound counterexamples.* In other words, the counterexample trace can be mapped back to a (valid) trace in the original specification with a simple inverse mapping RFF_S^{-1}.

The inverse mapping from sound states s_R in S_R back to states s in S is $RFF_S^{-1} = \{(s_R, s) \mid s_R \neq \mathit{other} \wedge (s, s_R) \in RFF_S\}$. Note that there is no mapping for the unsound state, "other." Since RFF is otherwise one-to-one, RFF_S^{-1} is a (partial) function. Events in S_R are the same as those in S, so they map identically. The inverse mapping of a trace is the inverse mapping of each state in the trace.

Theorem 3. *Any counterexample, c, produced from* $S_R = RFF_S(S)$ *and* $T'_R = CR(M(T_R))$ *is sound.*

Proof: By the definition of counterexample soundness, we must prove $RFF_S^{-1}(c)$ is a trace of S. If a state appears in c, it is sound, by Lemma 1. Since all states are sound, RFF_S^{-1} maps them back to S, and the S_R transitions implied by c are also transitions in S.

5 Mutation Analysis and Utility

Definition 6 does not require RFF to map *any* additional states beyond the initial state. However, the more states which are mapped one-to-one, the more traces there are in the reduced model. To be useful, the additional states must also be soundly reachable. (If any state is reachable only through an unsound state, no counterexample includes it.) Thus an analyst is free to reduce a specification to as few states as necessary for effective model checking, but may wish to make the reduced specification as large as possible.

Theorem 3 describes conditions under which counterexamples generated by a model checker from S_R are traces of S. Since RFF_T and M precede CR, they are unconstrained by soundness; indeed, *any* transforms may be used without invalidating Theorem 3. (Intuitively, as long as the temporal logic constraints are rewritten with CR, they can only induce more or fewer sound counterexamples.)

However, we wish to use the counterexamples for test cases for an implementation of S as outlined in Sect. 2. Clearly, the utility of test cases produced by this method depends on the transforms RFF_T and the set of mutation operators $\mathcal{M}$ from which M is drawn. In the remainder of this section, we describe constraints on RFF_T which yield coverage with respect to $\mathcal{M}$.

We first describe the ideal situation, which is different from the order depicted in Fig. 3, and then introduce practical considerations. Consider applying some mutant operator $M \in \mathcal{M}$ to T, which produces T'. Then T' is transformed via RFF_T and CR to produce T'_R. The model checker is run on (S_R, T'_R), (possibly) producing a counterexample t_R. The existences of counterexample traces may be characterized as follows:

If there is a trace t in S such that

1. t is a counterexample[2] with respect to T', and
2. $RFF_S(t)$ is a sound trace in S_R,

then some t_R does in fact exist for the model checker to find, and the trace $RFF_S^{-1}(t_R)$ is a counterexample with respect to T'.

Informally, this says that the set of counterexamples generated by the model checker from S_R is as close to being mutation adequate with respect to S, T, and $\mathcal{M}$ as possible. In other words, if a mutant T' can be distinguished by a sound trace from S_R, the model checker finds such a sound trace.

From a practical perspective, the chief drawback of the above characterization is that the reduction RFF_T is applied after each mutation operator is applied. If RFF_T can be completely automated for some application, this is not a serious problem. If, however, some human intervention is required to apply RFF_T, multiple transformations are an expensive proposition. Instead, it may be more practical to transform T with RFF_T once, then apply mutation operators to T_R instead of T. The result of test generation is then a mutant adequate set of tests with respect to S_R, T_R, and $\mathcal{M}$, rather than S, T, and $\mathcal{M}$.

6 Example

To validate the above proof, we apply finite focus to an example: the stack of a Java virtual machine. Abstracting the stack to just the number of items on the stack (stack size) and grouping instructions into those which push one item (`push1`), pop one item (`pop1`), or pop two items (`pop2`) still leaves an unbounded model much like the bank balance at the top of Fig. 2. It also includes specifications such as "`push1` then `pop1` leaves the stack unchanged."

We applied finite focus to get a usable model of a stack with zero, one, two, three, or "many" items plus a variable which goes unsound if the stack size exceeds three. We used two mutation operators. M1 changes constants in phrases such as `VAR = CONST`, and M2 negates boolean expressions. M1 gave 279 mutants, and M2 gave 129 mutants, for a total of 408. These produced 254 counterexamples. Combining duplicates and discarding prefixes yielded 9 unique tests with lengths from three to seven operations. By comparison, exhaustive enumeration yields 45 tests of length seven.

The stack specifications are in App. B, and the nine tests are in App. C.

7 Future Work

Since the goal is test generation, other soundness definitions and thus useful counterexamples are possible. In the bank account example in Fig. 2, we could

[2] Note that, by assumption, we have no efficient means of computing such a counterexample directly from S and T'. If there were an efficient means, we could simply run the model checker on S instead of S_R and therefore avoid the finite focus transform.

map counterexamples with the "other" state to a loose test that the balance is more than two. The inverse mapping, RFF_S^{-1}, would be more complex, but it may allow more information to be derived from reductions.

We plan to investigate different sets of mutation operators. These experiments, along with theoretical considerations of predicate test domination [17], should help us develop good classes of operators.

8 Conclusions

Previously, we showed how to use a model checker to measure test set coverage [1], and to develop mutation adequate tests for software specifications [2]. Mutation analysis in the finite domain of the model checker avoids many problems which plague program mutation analysis. But applying it depends critically on the feasibility of model checking specifications for realistic software system.

Here we address model checking feasibility for test generation, and present a reduction called finite focus for it. We define soundness for test generation: counterexamples generated from the reduced specification are test cases for the original specification. We prove that finite focus is sound, and experimentally show that it soundly reduces an unbounded model to a model which yields a small, yet mutation adequate, test set.

Soundness only constrains the part of the finite focus reduction that transforms the state machine and rewrites the temporal logic constraints. To maximize utility, we develop constraints on the transform of the temporal logic aspect which improve mutation adequacy in the original specification.

Acknowledgments

We thank Vadim Okun and Yaacov Yesha for their specification mutation engine. We appreciate reviewers' comments which guided us to fix flaws.

References

1. Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248. IEEE Computer Society, November 1999. Also NIST IR 6403.
2. Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, December 1998.
3. Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, June/July 1998.

4. Ramesh Bharadwaj and Constance Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software*, Paris, France, January 1997.
5. Ramesh Bharadwaj and Constance L. Heitmeyer. Model checking complete requirements specifications using abstraction. Memorandum Report NRL/MR/5540-97-7999, U.S. Naval Research Laboratory, Washington, DC 20375, November 1997.
6. Paul E. Black, Kelly M. Hall, Michael D. Jones, Trent N. Larson, and Phillip J. Windley. A brief introduction to formal methods. In *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference (CICC '96)*, pages 377–380. IEEE, May 1996.
7. Jerry R. Burch, Edmund M. Clarke, Jr., Ken L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
8. John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN Workshop*, Rutgers, NJ, August 1996. Also WVU Technical Report #NASA-IVV-96-022.
9. William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498 – 520, July 1998.
10. Edmund M. Clarke, Jr., E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
11. Edmund M. Clarke, Jr., Orna Grumberg, and David E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency – Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
12. Richard A. De Millo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
13. 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.
14. Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Toulouse, France, September 1999. To Appear.
15. Constance Heitmeyer, James Kirby, Jr., Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.
16. Gerald J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
17. D. Richard Kuhn. Fault classes and error detection in specification based testing. *ACM Transactions on Software Engineering Methodology*, 8(4), October 1999.
18. Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, New Jersey 08540, 1994.
19. Ken L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

20. John M. Rushby. Ubiquitous abstraction: A new approach to mechanized formal verification. In *Proceedings of Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 176–178. IEEE Computer Society, December 1998.
21. Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11), November 1996.

A Proof that Rewrites Prevent Checking

In this appendix we prove that when rewritten with the constraint rewriting rules, CR, any expression evaluates to $True$ or $False$ as appropriate in any state in which s is $False$. We use definitions of temporal logic operators from Clarke, et. al. [10].

Proof that $cr(\texttt{AX}\ f, True) = \texttt{AX}\ s \mathrel{-\!\!>} cr(f, True)$ yields $True$ when s is $False$.

$$\begin{aligned} u \models \texttt{AX}\ False \mathrel{-\!\!>} cr(f, True) &= u \models \texttt{AX}\ True \\ &= \forall v \mid (u,v) \in \mathcal{T} \rightarrow v \models True \\ &= \forall v \mid (u,v) \in \mathcal{T} \rightarrow True \\ &= \forall v \mid True \\ &= True \end{aligned}$$

Proof that $cr(\texttt{AX}\ f, False) = \texttt{AX}\ s \& cr(f, False)$ yields $False$.

$$\begin{aligned} u \models \texttt{AX}\ False\ \&\ cr(f, False) &= u \models \texttt{AX}\ False \\ &= \forall v \mid (u,v) \in \mathcal{T} \rightarrow v \models False \\ &= \forall v \mid (u,v) \in \mathcal{T} \rightarrow False \\ &= \forall v \mid (u,v) \notin \mathcal{T} \\ &\text{and since every state has at least one outgoing transition} \\ &= False \end{aligned}$$

Proof that $cr(\texttt{EX}\ f, True) = \texttt{EX}\ s \mathrel{-\!\!>} cr(f, True)$ yields $True$.

$$\begin{aligned} u \models \texttt{EX}\ False \mathrel{-\!\!>} cr(f, True) &= u \models \texttt{EX}\ True \\ &= \exists v \mid (u,v) \in \mathcal{T} \wedge v \models True \\ &= \exists v \mid (u,v) \in \mathcal{T} \wedge True \\ &= \exists v \mid (u,v) \in \mathcal{T} \\ &\text{and since every state has at least one outgoing transition} \\ &= True \end{aligned}$$

Proof that $cr(\texttt{EX}\ f, False) = \texttt{EX}\ s \& cr(f, False)$ yields $False$.

$$\begin{aligned} u \models \texttt{EX}\ False \& cr(f, False) &= u \models \texttt{EX}\ False \\ &= \exists v \mid (u,v) \in \mathcal{T} \wedge v \models False \\ &= \exists v \mid (u,v) \in \mathcal{T} \wedge False \\ &= \exists v \mid False \\ &= False \end{aligned}$$

Proof that $cr(\mathtt{A}g\mathtt{U}f, True) = \mathtt{A}g\mathtt{U}s \mathrel{-\!\!>} cr(f, True)$ yields $True$. Note that we are quantifying over paths; s_i means "the i^{th} state of the path."

$$\begin{aligned}
u \models \mathtt{A}g\mathtt{U}\ False \mathrel{-\!\!>} cr(f, True) = u \models \mathtt{A}g\mathtt{U}True & \\
= \forall path(s_0 = u, s_1, \ldots, s_n)\ [& \\
\exists i\ [i \geq 0 \wedge s_i \models True\ \wedge \forall j\ [0 \leq j < i \rightarrow s_j \models g]]] & \\
= \forall path(s_0 = u, s_1, \ldots, s_n)\ [& \\
\exists i\ [i \geq 0 \wedge \forall j\ [0 \leq j < i \rightarrow s_j \models g]]] & \\
\text{choosing } i = 0 & \\
= \forall path(s_0 = u, s_1, \ldots, s_n)\ [& \\
0 \geq 0 \wedge \forall j\ [0 \leq j < 0 \rightarrow s_j \models g]] & \\
= \forall path(s_0 = u, s_1, \ldots, s_n)\ [\forall j\ [False \rightarrow s_j \models g]] & \\
= \forall path(s_0 = u, s_1, \ldots, s_n)\ [True] & \\
= True &
\end{aligned}$$

Proof that $cr(\mathtt{A}g\mathtt{U}f, False) = \mathtt{A}g\mathtt{U}s \& cr(f, False)$ yields $False$.

$$\begin{aligned}
u \models \mathtt{A}g\mathtt{U}\ False \& cr(f, False) = u \models \mathtt{A}g\mathtt{U}False & \\
= \forall path(s_0 = u, s_1, \ldots, s_n)\ [& \\
\exists i\ [i \geq 0 \wedge s_i \models False\ \wedge \forall j\ [0 \leq j < i \rightarrow s_j \models g]]] & \\
= \forall path(s_0 = u, s_1, \ldots, s_n)\ [& \\
\exists i\ [i \geq 0 \wedge False\ \wedge \forall j\ [0 \leq j < i \rightarrow s_j \models g]]] & \\
= \forall path(s_0 = u, s_1, \ldots, s_n)\ [\exists i\ [False]] & \\
= \forall path(s_0 = u, s_1, \ldots, s_n)\ [False] & \\
= False &
\end{aligned}$$

The proofs of $cr(\mathtt{E}g\mathtt{U}f, True) = \mathtt{E}g\mathtt{U}s \mathrel{-\!\!>} cr(f, True)$ and $cr(\mathtt{E}g\mathtt{U}f, False) = \mathtt{E}g\mathtt{U}s \& cr(f, False)$ are the same as the proofs of $cr(\mathtt{A}g\mathtt{U}f, True)$ and $cr(\mathtt{A}g\mathtt{U}f, False)$ respectively, except that they use $\exists path$ instead of $\forall path$.

Proof that $cr(\mathtt{AF}\ f, True) = \mathtt{AF}\ s \mathrel{-\!\!>} cr(f, True)$ yields $True$.

$$\begin{aligned}
u \models \mathtt{AF}\ False \mathrel{-\!\!>} cr(f, True) &= u \models \mathtt{A}True\mathtt{U}False \mathrel{-\!\!>} cr(f, True) \\
&= True
\end{aligned}$$

Proof that $cr(\mathtt{AF}\ f, False) = \mathtt{AF}\ s \& cr(f, False)$ yields $False$.

$$\begin{aligned}
u \models \mathtt{AF}\ False \& cr(f, False) &= u \models \mathtt{A}True\mathtt{U}False \& cr(f, False) \\
&= False
\end{aligned}$$

Proof that $cr(\mathtt{EF}\ f, True) = \mathtt{EF}\ s \mathrel{-\!\!>} cr(f, True)$ yields $True$.

$$\begin{aligned}
u \models \mathtt{EF}\ False \mathrel{-\!\!>} cr(f, True) &= u \models \mathtt{E}True\mathtt{U}False \mathrel{-\!\!>} cr(f, True) \\
&= True
\end{aligned}$$

Proof that $cr(\mathtt{EF}\ f, False) = \mathtt{EF}\ s \& cr(f, False)$ yields $False$.

$$\begin{aligned}
u \models \mathtt{EF}\ False \& cr(f, False) &= u \models \mathtt{E}True\mathtt{U}False \& cr(f, False) \\
&= False
\end{aligned}$$

Proof that $cr(\texttt{AG}\ f, True) = \texttt{AG}\ s \mathbin{->} cr(f, True)$ yields $True$.

$$\begin{aligned} u \models \texttt{AG}\ False \mathbin{->} cr(f, True) &= u \models \texttt{AG}\ True \\ &= u \models \texttt{\~{}EF}\ \texttt{\~{}}True \\ &= u \models \texttt{\~{}EF}\ False \\ &= u \models \texttt{\~{}E}True\texttt{U}False \\ &= u \models \texttt{\~{}}False \\ &= u \models True \\ &= True \end{aligned}$$

Proof that $cr(\texttt{AG}\ f, False) = \texttt{AG}\ s\&cr(f, False)$ yields $False$.

$$\begin{aligned} u \models \texttt{AG}\ False\&cr(f, False) &= u \models \texttt{AG}\ False \\ &= u \models \texttt{\~{}EF}\ \texttt{\~{}}False \\ &= u \models \texttt{\~{}EF}\ True \\ &= u \models \texttt{\~{}E}True\texttt{U}True \\ &= u \models \texttt{\~{}}True \\ &= u \models False \\ &= False \end{aligned}$$

Proof that $cr(\texttt{EG}\ f, True) = \texttt{EG}\ s \mathbin{->} cr(f, True)$ yields $True$.

$$\begin{aligned} u \models \texttt{EG}\ False \mathbin{->} cr(f, True) &= u \models \texttt{EG}\ True \\ &\text{similar to } \texttt{AG}\ True \text{ case} \\ &= True \end{aligned}$$

Proof that $cr(\texttt{EG}\ f, False) = \texttt{EG}\ s\&cr(f, False)$ yields $False$.

$$\begin{aligned} u \models \texttt{EG}\ False\&cr(f, False) &= u \models \texttt{EG}\ False \\ &\text{similar to } \texttt{AG}\ False \text{ case} \\ &= False \end{aligned}$$

We proved that expressions whose outermost operators are temporal operators always evaluate to the desired $True$ or $False$ value when rewritten according to the rules. However, specifications may be some boolean function of temporal operator expression. For instance, for a specification such as `SPEC ! AG p` to evaluate to $True$, the temporal operator expression must evaluate to $False$. (Thus the need for rewriting so the expression will be $True$ or it will be $False$.) Assuming subsequent rewrites make the subformulae $True$ or $False$, we show that these rewrites make the formula $True$ or $False$, as appropriate.

Proof that $cr(!\, f, v) =!\, cr(f, \tilde{\ } v)$ yields $True$ or $False$ appropriately.

$$\begin{aligned} cr(!\, f, True) &= !\, cr(f, \tilde{\ } True) \\ &= !\, cr(f, False) \\ &= !False \\ &= True \end{aligned}$$

$$\begin{aligned} cr(!\, f, False) &= !\, cr(f, \tilde{\ } False) \\ &= !\, cr(f, True) \\ &= !True \\ &= False \end{aligned}$$

Proof that $cr(f \;\&\; g, v) = cr(f, v) \,\&\, cr(g, v)$ yields $True$ or $False$ appropriately.

$$\begin{aligned} cr(f \;\&\; g, True) &= cr(f, True) \,\&\, cr(g, True) \\ &= True \,\&\, True \\ &= True \end{aligned}$$

$$\begin{aligned} cr(f \;\&\; g, False) &= cr(f, False) \,\&\, cr(g, False) \\ &= False \,\&\, False \\ &= False \end{aligned}$$

Proof that $cr(f \mid g, v) = cr(f, v) \mid cr(g, v)$ yields $True$ or $False$ appropriately.

$$\begin{aligned} cr(f \mid g, True) &= cr(f, True) \mid cr(g, True) \\ &= True \mid True \\ &= True \end{aligned}$$

$$\begin{aligned} cr(f \mid g, False) &= cr(f, False) \mid cr(g, False) \\ &= False \mid False \\ &= False \end{aligned}$$

Proof that $cr(f \;{-}{>}\; g, v) = cr(f, \tilde{\ } v) \;{-}{>}\; cr(g, v)$ yields $True$ or $False$ appropriately.

$$\begin{aligned} cr(f \;{-}{>}\; g, True) &= cr(f, \tilde{\ } True) \;{-}{>}\; cr(g, True) \\ &= cr(f, False) \;{-}{>}\; cr(g, True) \\ &= False \;{-}{>}\; True \\ &= True \end{aligned}$$

$$\begin{aligned} cr(f \;{-}{>}\; g, False) &= cr(f, \tilde{\ } False) \;{-}{>}\; cr(g, False) \\ &= cr(f, True) \;{-}{>}\; cr(g, False) \\ &= True \;{-}{>}\; False \\ &= False \end{aligned}$$

Proof of $cr(f <\!\!-\!\!> g, v) = cr(f, {}^{\sim}v) -\!\!> cr(g, v)$ & $cr(g, {}^{\sim}v) -\!\!> cr(f, v)$.

$$\begin{aligned}
cr(f <\!\!-\!\!> g, True) &= cr(f, {}^{\sim}True) -\!\!> cr(g, True) \\
&\qquad\qquad \&\ cr(g, {}^{\sim}True) -\!\!> cr(f, True) \\
&= cr(f, False) -\!\!> cr(g, True)\ \&\ cr(g, False) -\!\!> cr(f, True) \\
&= False -\!\!> True\ \&\ False -\!\!> True \\
&= True\ \&\ True \\
&= True
\end{aligned}$$

$$\begin{aligned}
cr(f <\!\!-\!\!> g, False) &= cr(f, {}^{\sim}False) -\!\!> cr(g, False) \\
&\qquad\qquad \&\ cr(g, {}^{\sim}False) -\!\!> cr(f, False) \\
&= cr(f, True) -\!\!> cr(g, False)\ \&\ cr(g, True) -\!\!> cr(f, False) \\
&= True -\!\!> False\ \&\ True -\!\!> False \\
&= False\ \&\ False \\
&= False
\end{aligned}$$

B Java Virtual Machine Stack

This section gives the model and specification we used to generate tests. We enumerate the stack size. Following this we give a second SMV file which models the stack size as a number.

```
-- $ Id: javaStack.smv,v 1.2 1999/08/05 13:50:20 black Exp $
-- *created  "Fri Jun 26 11:20:23 1998" *by "Paul E. Black"
-- *modified "Thu Aug  5 09:46:25 1999" *by "Paul E. Black"
-- first try at state machine abstraction of
--      Java Smart Card virtual machine
-- this just models the first few places of the operand stack

MODULE main
VAR
    -- system inputs
    instr : {in_push1, in_pop1, in_pop2};
    -- internal states
    Sound : boolean;
    StackSize : {size0, size1, size2, size3, sizeBig,
                                              sizeUndefined};
                -- SKIMP stack overflow is an exception which
                -- is not caught and terminates the program.

ASSIGN
init(Sound) := 1; -- state begins sound
next(Sound) := case
    -- abstraction looses accuracy
    StackSize=size3 & instr=in_push1 : 0;
    1 : Sound; -- otherwise soundness is unchanged
esac;

-- allow only instructions which don't cause stack underflow
-- Java compilers should ensure this
init(instr) := in_push1;
next(instr) := case
    next(StackSize)=size0 : in_push1;
    next(StackSize)=size1 : {in_push1, in_pop1};
    1 : {in_push1, in_pop1, in_pop2};
esac;

init(StackSize) := size0; -- stack begins empty
next(StackSize) := case
    -- push one item on the stack
    StackSize=size0 & instr=in_push1 : size1;
    StackSize=size1 & instr=in_push1 : size2;
    StackSize=size2 & instr=in_push1 : size3;
    StackSize=size3 & instr=in_push1 : sizeBig;
    StackSize=sizeBig & instr=in_push1 : sizeBig;
    -- pop one item from the stack
    StackSize=size1 & instr=in_pop1 : size0;
    StackSize=size2 & instr=in_pop1 : size1;
    StackSize=size3 & instr=in_pop1 : size2;
    -- Size after popping from a "big" stack is nondeterministic
    -- since we lost information.
    StackSize=sizeBig & instr=in_pop1 : {size3,sizeBig};
    -- pop two items from the stack
    StackSize=size2 & instr=in_pop2 : size0;
    StackSize=size3 & instr=in_pop2 : size1;
    -- Size after popping from a "big" stack is nondeterministic
    -- since we lost information.
    StackSize=sizeBig & instr=in_pop2 : {size2,sizeBig};
    -- anything else is undefined
    1: sizeUndefined;
esac;

-- These are erroneous in JVM.  They should never be generated by
--        compilers and should be caught by the verifier.
TRANS
    StackSize=size0 -> !(instr=in_pop1)
TRANS
    StackSize=size0 -> !(instr=in_pop2)
TRANS
    StackSize=size1 -> !(instr=in_pop2)


SPEC AG(Sound -> (! StackSize=sizeUndefined))
-- push one item on the stack
SPEC AG(Sound ->(StackSize=size0 & instr=in_push1 ->
                              AX(Sound ->(StackSize=size1))))
SPEC AG(Sound ->(StackSize=size1 & instr=in_push1 ->
                              AX(Sound ->(StackSize=size2))))
SPEC AG(Sound ->(StackSize=size2 & instr=in_push1 ->
                              AX(Sound ->(StackSize=size3))))
SPEC AG(Sound ->(StackSize=size3 & instr=in_push1 ->
                              AX(Sound ->(StackSize=sizeBig))))
SPEC AG(Sound ->(StackSize=sizeBig & instr=in_push1 ->
                              AX(Sound ->(StackSize=sizeBig))))
-- pop one item from the stack
SPEC AG(Sound ->(StackSize=size1 & instr=in_pop1 ->
                              AX(Sound ->(StackSize=size0))))
SPEC AG(Sound ->(StackSize=size2 & instr=in_pop1 ->
                              AX(Sound ->(StackSize=size1))))
SPEC AG(Sound ->(StackSize=size3 & instr=in_pop1 ->
                              AX(Sound ->(StackSize=size2))))
SPEC AG(Sound ->(StackSize=sizeBig & instr=in_pop1 ->
               AX(Sound ->(StackSize=size3 | StackSize=sizeBig))))
-- pop two items from the stack
SPEC AG(Sound ->(StackSize=size2 & instr=in_pop2 ->
                              AX(Sound ->(StackSize=size0))))
SPEC AG(Sound ->(StackSize=size3 & instr=in_pop2 ->
                              AX(Sound ->(StackSize=size1))))
SPEC AG(Sound ->(StackSize=sizeBig & instr=in_pop2 ->
               AX(Sound ->(StackSize=size2 | StackSize=sizeBig))))
-- push1, pop1 returns stack to the same state
SPEC AG(Sound ->(StackSize=size0 & instr=in_push1 ->
                   AX(Sound ->(instr=in_pop1 ->
                              AX(Sound ->(StackSize=size0))))))
SPEC AG(Sound ->(StackSize=size1 & instr=in_push1 ->
                   AX(Sound ->(instr=in_pop1 ->
                              AX(Sound ->(StackSize=size1))))))
SPEC AG(Sound ->(StackSize=size2 & instr=in_push1 ->
                   AX(Sound ->(instr=in_pop1 ->
                              AX(Sound ->(StackSize=size2))))))
SPEC AG(Sound ->(StackSize=size3 & instr=in_push1 ->
                   AX(Sound ->(instr=in_pop1 ->
                              AX(Sound ->(StackSize=size3))))))
-- push1, push1, pop2 returns stack to the same state
SPEC AG(Sound ->(StackSize=size0 & instr=in_push1 ->
                   AX(Sound ->(instr=in_push1 ->
                         AX(Sound ->(instr=in_pop2 ->
                              AX(Sound ->(StackSize=size0))))))))
SPEC AG(Sound ->(StackSize=size1 & instr=in_push1 ->
                   AX(Sound ->(instr=in_push1 ->
                         AX(Sound ->(instr=in_pop2 ->
                              AX(Sound ->(StackSize=size1))))))))
SPEC AG(Sound ->(StackSize=size2 & instr=in_push1 ->
                   AX(Sound ->(instr=in_push1 ->
                         AX(Sound ->(instr=in_pop2 ->
                              AX(Sound ->(StackSize=size2))))))))
SPEC AG(Sound ->(StackSize=size3 & instr=in_push1 ->
                   AX(Sound ->(instr=in_push1 ->
                         AX(Sound ->(instr=in_pop2 ->
                              AX(Sound ->(StackSize=size3))))))))
```

```
-- $ Id: javaStack2.smv,v 1.3 1999/08/05 13:51:05 black Exp $
-- *created  "Fri Jun 26 11:20:23 1998" *by "Paul E. Black"
-- *modified "Thu Aug  5 09:48:18 1999" *by "Paul E. Black"
-- first try at state machine abstraction of
--        Java Smart Card virtual machine
-- this just models the first few places of the operand stack
-- This version models the stack size with an integer subrange so
--    we can succinctly model next size (just +1, -1, or -2).
--    However we must save previous values of StackSize since
--    SPEC clause values are no longer based entirely on cases.
MODULE main
VAR
    -- system inputs
    instr : {in_push1, in_pop1, in_pop2};
    -- internal states
    Sound : boolean;
    StackSize : 0..5; -- 4 is Big, 5 is undefined
    PStackSize : 0..5;
    PPStackSize : 0..5;
    PPPStackSize : 0..5;
                -- SKIMP stack overflow is an exception which
                -- is not caught and terminates the program.
```

```
DEFINE
    sizeBig := 4;
    sizeUndefined := 5;
ASSIGN
init(Sound) := 1; -- state begins sound
next(Sound) := case
    -- abstraction looses accuracy
    StackSize=3 & instr=in_push1 : 0;
    1 : Sound; -- otherwise soundness is unchanged
esac;
-- allow only instructions which don't cause stack underflow
-- Java compilers should ensure this
init(instr) := in_push1;
next(instr) := case
    next(StackSize)=0 : in_push1;
    next(StackSize)=1 : {in_push1, in_pop1};
    1 : {in_push1, in_pop1, in_pop2};
esac;
init(StackSize) := 0; -- stack begins empty
next(StackSize) := case
    -- push one item on the stack
    StackSize<sizeBig & instr=in_push1 : StackSize+1;
    StackSize=sizeBig & instr=in_push1 : sizeBig;

    -- pop one item from the stack
    StackSize>0 & StackSize<sizeBig & instr=in_pop1 :
                                             StackSize - 1;
    -- Size after popping from a "big" stack is nondeterministic
    -- since we lost information.
    StackSize=sizeBig & instr=in_pop1 : {3,sizeBig};
    -- pop two items from the stack
    StackSize>1 & StackSize<sizeBig & instr=in_pop2 :
                                             StackSize - 2;
    -- Size after popping from a "big" stack is nondeterministic
    -- since we lost information.
    StackSize=sizeBig & instr=in_pop2 : {2,sizeBig};
    -- anything else is undefined
    1: sizeUndefined;
esac;
-- maintain "Previous" values of stack size
next(PStackSize) := StackSize;
next(PPStackSize) := PStackSize;
next(PPPStackSize) := PPStackSize;
-- These are erroneous in JVM.  They should never be generated by
--        compilers and should be caught by the verifier.
TRANS
    StackSize=0 -> !(instr=in_pop1)
TRANS
    StackSize=0 -> !(instr=in_pop2)
TRANS
    StackSize=1 -> !(instr=in_pop2)

SPEC AG(Sound -> (! StackSize=sizeUndefined))
-- push one item on the stack
SPEC AG(Sound ->(StackSize<sizeBig & instr=in_push1 ->
                          AX(Sound ->(StackSize=PStackSize+1))))
SPEC AG(Sound ->(StackSize=sizeBig & instr=in_push1 ->
                          AX(Sound ->(StackSize=sizeBig))))
-- pop one item from the stack
```

```
SPEC AG(Sound ->(StackSize>0 & StackSize<sizeBig
                                      & instr=in_pop1 ->
                          AX(Sound ->(StackSize=PStackSize - 1))))
SPEC AG(Sound ->(StackSize=sizeBig & instr=in_pop1 ->
                  AX(Sound ->(StackSize=3 | StackSize=sizeBig))))
-- pop two items from the stack
SPEC AG(Sound ->(StackSize>1 & StackSize<sizeBig
                                      & instr=in_pop2 ->
                          AX(Sound ->(StackSize=PStackSize - 2))))
SPEC AG(Sound ->(StackSize=sizeBig & instr=in_pop2 ->
                  AX(Sound ->(StackSize=2 | StackSize=sizeBig))))
-- push1, pop1 returns stack to the same state
SPEC AG(Sound ->(instr=in_push1 ->
        AX(Sound ->(instr=in_pop1 ->
                          AX(Sound ->(StackSize=PPStackSize)))))
-- push1, push1, pop2 returns stack to the same state
SPEC AG(Sound ->(instr=in_push1 ->
               AX(Sound ->(instr=in_push1 ->
                   AX(Sound ->(instr=in_pop2 ->
                       AX(Sound ->(StackSize=PPPStackSize)))))))
```

C JVM Stack Tests

Table 1 shows the nine tests which resulted from applying our test generation method. We only show the instruction for each step: the stack size is easily determined. In the model, the type of instruction in a step changes the stack size in the next step. Thus the final step of each test checks the stack size, and the choice of final instruction is irrelevant. The SVM model checker [19] chose push1 as the last instruction of each test; we leave out that last instruction from this table.

Test	*Instruction types*
1	push1 pop1
2	push1 push1 pop1 push1 pop2
3	push1 push1 pop1 pop1
4	push1 push1 pop2
5	push1 push1 push1 pop1 pop2
6	push1 push1 push1 pop1 push1 pop2
7	push1 push1 push1 pop1 pop1
8	push1 push1 push1 pop2 pop1
9	push1 push1 push1 pop2 push1 pop2

Table 1. Generated Stack Tests

Although there are similarities between tests, we need all nine tests for 100% mutation coverage.