Testing with Model Checker: Insuring Fault Visibility

VADIM OKUN PAUL E. BLACK
National Institute of
Standards and Technology
Gaithersburg, MD 20899
USA
{vokun1,paul.black}@nist.gov

YAACOV YESHA
University of Maryland
Baltimore County
Baltimore, MD 21250
USA
yayesha@cs.umbc.edu

Abstract: - To detect a fault in software, a test case execution must enable an intermediate error to propagate to the output. We describe two specification-based mutation testing methods that use a model checker to guarantee propagation of faults to the visible outputs. We evaluate the methods empirically and show that they are better than the previous “direct reflection” approach.

Key-Words: - formal methods; model checking; specification-based testing; mutation testing

1 Introduction

Specification-based testing is a black-box technique, that is, it assumes that internal states of the program implementing the specification are unknown, hence failures can only be detected in external responses. Although model checkers can be used to generate tests [3, 5], existing methods allow the model checker to choose tests that do not cause faults to propagate to the program's output. Further details, references and examples can be found in [13].

Goradia [9] presents typical cases that prevent a fault in an intermediate state from propagating to the output. For example, in a relational expression such as `state_var > z`, an incorrect value of `state_var` may still yield the correct Boolean value of the relational expression.

In this paper, we present two new approaches using model checker to guarantee that tests cause detectable output failures. We briefly introduce model checking, test generation using model checkers, and mutation adequacy criterion here.

1.1 Model Checking

Model checking is a formal technique based on state exploration. Input to a model checker has 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 expressions over states and execution paths. Conceptually, a model checker visits all reachable states and verifies that the temporal logic expressions are satisfied over all paths. If an expression is not satisfied, the model checker attempts to generate a counterexample in the form of a sequence of states.

A common logic for model checking is the branching-time Computation Tree Logic (CTL) [12], which extends propositional logic with temporal operators. For example, a CTL formula `AG safe` means that all reachable states are safe, and `AG (request -> AX response)` means that a request is always followed by a response on the next step.

We use SMV, a CTL symbolic model checker [12]. In SMV, a specification consists of one or more modules. One module, named `main`, is the top level module in SMV. Fig. 1 is an SMV example derived from [14]. We refer to this example throughout the paper. Variables `d`, `b`, and `f` are inputs, `e` and `a` are intermediate variables. The statement `init(e) := 0;` sets `e` to 0 initially. The next value of `e` is `1` if the *guard* `f = On` is true, otherwise it is `0`. The output is the variable `out`, which may be `Low` or `High`. Its value is `High` if `a` is greater than 10, otherwise it is `Low`. The `SPEC` clause states that if `f` is `On`, it is possible to get to some state where `out` is `High`. We often drop the keyword `SPEC` when clear from the context.

1.2 Generating Software Tests

Model checking is being applied to test generation and test coverage evaluation [3, 5]. In both uses, one first

```
MODULE main
VAR
  d: 0..5;     b: 0..11;
  f: {On, Off};
  out: {Low, High};
  a: 0..16;    e: 0..1;
ASSIGN
  init(e) := 0;
  next(e) := case
    f = On : 1;
    1 : 0;
  esac;
  a := e * d + b;
  out := case
    a > 10 : High;
    1 : Low;
  esac;
SPEC AG (f = On -> EF out = High)
```

Fig. 1: An SMV Example

chooses a test criterion [8], that is, decides on a philosophy about what properties of a specification must be exercised to constitute a thorough test.

One applies the chosen test criterion to the specification to derive test requirements, i.e., a set of individual properties to be tested. To use a model checker, these requirements must be represented as temporal logic formulas [2]. To generate tests, the test criterion is applied to yield negative requirements, that is, requirements that are considered satisfied if the corresponding temporal logic formulas are inconsistent with the state machine. For instance, if the criterion is state coverage, the negative requirements are that the machine is never in state 1, never in state 2, etc.

When the model checker finds that a requirement is inconsistent, it produces a counterexample. Again, in the case of state coverage, the counterexamples would have stimulus that puts the machine in state 1 (if it is reachable), another to put the machine in state 2, etc.

The set of counterexamples is reduced, or winnowed, by eliminating duplicates and those that are prefixes of other, longer counterexamples.

1.3 Specification Mutation Criterion

Mutation adequacy [6] is a test criterion that naturally yields negative requirements. The specification-based mutation analysis scheme in [3] applies mutation operators to the state machine or the temporal logic expressions yielding a set of faulty, or mutant, expressions. Some mutation operators are replacing a variable with another variable, replacing an integer variable a with $a + 1$, replacing a conjunction with a disjunction.

Any particular mutant might be consistent or inconsistent with the state machine [2]. A consistent mutant is a temporal logic formula that is true over all possible executions defined by the state machine. Such mutants are not useful and may be discarded. A mutation adequate test set should distinguish between the correct behavior and the behavior of inconsistent mutants.

The rest of the paper is organized as follows. Section 2 briefly reviews similar previous work and the existing specification-based mutation method. Section 3 presents our two approaches: in-line expansion and state machine duplication (SM duplication). Section 4 uses the example in Fig. 1 to compare approaches. In the second part of the section, we evaluate the effectiveness of the approaches at detecting seeded faults in a C program implementing a portion of *TCAS*. Our conclusions are in Section 5.

2 Existing Approaches

First, some terminology. A *fault* is a defect in the code, informally, a bug. A (visible) *failure* is an unacceptable result of execution on some test data; in other words, it is observable incorrect behavior. A failure is caused by one or more faults. A *potential failure,* or potential error, is an intermediate incorrect result.

2.1 Related Work

There is an extensive body of research in program-based testing that studied conditions for detecting a fault from external responses [15, 9]. The RELAY model [15] defines the revealing conditions under which a fault is detected. First, a potential error originates at the smallest subexpression containing the fault. Then the potential error propagates through computations and information flow until a failure is revealed. Test data can be selected to satisfy revealing conditions. In our work we rely on the model checker to achieve error propagation.

Program mutation testing in its original formulation—often referred to as strong mutation—requires the output of a mutant to differ from the original. Weak mutation [10] only requires that the execution of a component of the mutant and the original produce different values. Since in this paper we deal with visible failures, we require strong mutation.

```
AG (f = On -> AX e = 1)
AG (!(f = On) -> AX e = 0)
AG (a = e * d + b)
AG (a > 10 -> out = High)
AG (a <= 10 -> out = Low)
```

Fig. 2: Applying Direct Reflection

Fabbri et. al. [7] categorized mutation operators for different components of Statecharts and provided strategies to abstract and incrementally test the components.

2.2 Direct Reflection

The test criterion we concentrate on in this paper is specification-based mutation adequacy. It is implemented by mutating temporal logic formulas. These formulas may be derived from the state machine by a mechanical process called *reflection* [2, 1].

Fig. 2 contains formulas derived from the assignment statements in Fig. 1. For instance, the `next` clause for the variable `e` in Fig. 1 is reflected into the first two formulas. The formulas directly reflect the state machine transition relation; we refer to this method as *Direct Reflection* to differentiate it from the *In-line expansion* approach which we describe in Section 3.1.

For each mutant, the model checker finds a counterexample that leads to a potential failure if possible. However, there is no guarantee that the potential failure will propagate to a visible output. Consider a mutant of the third formula in Fig. 2:

$$\texttt{AG (a = e * (d + 1) + b)} \qquad (1)$$

Choosing `b = 0`, `d = 0`, and `f = On` shows an inconsistency in an intermediate variable `a`, but not in the output variable `out`. Such a test is of little value.

3 Two New Approaches

In this section we present two new approaches which use a model checker to produce counterexamples that cause faults to be visible.

3.1 In-line Expansion

In this approach, only reflections of the transition relation for output variables are generated and considered for mutation. In these reflected temporal logic formulas, any intermediate variables are replaced with

```
AG (f=On  -> AX(d+b>10 -> out=High))
AG (f!=On -> AX(b>10 -> out=High))
AG (f=On  -> AX(d+b<=10 -> out=Low))
AG (f!=On -> AX(b<=10 -> out=Low))
```

Fig. 3: Applying In-line Expansion

in-line copies of their transition relations. This substitution is performed repeatedly until the formulas are comprised exclusively of input and output variables. Fig. 3 contains formulas derived from the statements in Fig. 1 using in-line expansion method. Since only inputs and outputs appear, the model checker finds counterexamples that affect the outputs. As in direct reflection, all mutants can be checked against the original state machine in a single run.

If there are conditional expressions in the transition relations for intermediate variables, this approach leads to an exponential increase in the number or size of logical formulas: different paths must be specified explicitly. The example in Fig. 1 has two conditional statements, each with two branches, for a total of four possible paths, so there are four formulas in Fig. 3.

3.2 State Machine (SM) Duplication

The rest of Section 3 deals with the other approach: duplicating the state machine. Suppose the model checker compares the external behavior of the original and mutated state machines. Any counterexamples produced must exhibit failures, that is, inputs must be chosen to manifest differences in the outputs. To facilitate this comparison, we begin by duplicating the state machine and insure that the duplicate always takes the same transition as the original. Then we can mutate the duplicate to implement the mutation test criterion.

More formally, let SM be the description of the original state machine. Let SM_d be a duplicate of SM containing a mutation. SM and SM_d have separate sets of outputs. We combine the two machines into a single state machine SM^+. We then assert that the values of the outputs of SM and SM_d are identical over SM^+. If SM_d has an observable fault, the model checker will produce a counterexample leading to the state where SM and SM_d differ in an output value.

From the counterexample, we can construct a test case containing values for inputs and the expected values for the outputs of the original state machine, SM. If the specification allows nondeterministic behavior, the expected outputs might not be adequate as an oracle. Nevertheless, the tests are expected to cause some

```
MODULE original(d, b, f)
VAR
  out: {Low, High};
  a: 0..16; e: 0..1;
ASSIGN
... same transitions as in Fig. 1
MODULE duplicate(d, b, f)
... same as original, to be mutated
MODULE main
VAR
  d: 0..5; b: 0..11;
  f: {On, Off};
  good : original(d, b, f);
  mutant : duplicate(d, b, f);
SPEC AG (good.out = mutant.out)
```

Fig. 4: A Duplication Example

faulty implementation to exhibit failures.

3.3 Handling Nondeterminism

If there are any nondeterministic transitions in the original state machine, SM and SM_d embedded in SM^+ are allowed to make different choices. For example, the statement `var := {1, 2};` assigns `var` the value of 1 or 2.

When a variable is assigned a set of values, all possible values are explored independently of each other. If SM is duplicated naively, SMV could provide a counterexample that chooses one value of a variable in SM and another value of the corresponding variable in SM_d, that is, the "difference" arises from accidental differences or differences in execution, not from semantic differences. We can force SM and SM_d to make the same choices by declaring a new global variable for each nondeterministic choice. We modify both SM and SM_d to choose depending on this common global variable.

While this method is general, it is excessive for variables without explicit transitions, such as inputs. We can simply move their declarations to the `main` module and pass them to SM and SM_d as parameters.

3.4 An Illustrative Example

Consider the sample model in Fig. 1. As Fig. 4 illustrates, we rename `main` to `original`[1], move declarations of input variables into the new `main` module, instantiate the `original` and `duplicate` modules (SM and SM_d, respectively) in the new `main`, and pass inputs as parameters. The CTL formula asserts that outputs of the original and mutant modules are always the same.

[1]If the original state machine description has more than one module, all of them must be renamed for duplication.

Assignment statements in the `duplicate` module from Fig. 4 are candidates for mutation. Some mutations may result in a semantically invalid SMV model. Two cases are common. First, a mutation operator replacing one variable with another may generate a mutant containing a circular dependency. Our tools use SMV's built-in analysis to automatically remove such mutants from further consideration. Second, the value of an expression on the right hand side of an assignment in the mutant may be outside of the range of the variable on the left hand side. Consider a mutant of an assignment for variable `a` in Fig. 1.

```
a := e * (d + 1) + b;        (2)
```

We change the declaration of `a` in the mutant to expand its range when needed.

The example only shows synchronous composition of modules. In case of interleaving, introduced by the keyword `process` in SMV, special care must be taken to ensure that the processes of original and duplicate machines follow each other in an orderly fashion.

3.5 Sharing Independent Variables

Some parts of the model may not depend on the variable affected by a particular mutation. Strictly speaking, for any particular mutation, we need only duplicate the variable whose assignment is being mutated and any dependent variables. Dependency analysis can stop at output variables. Such dependency can be determined using slicing [16]. If the model has many modules, only the module with the mutation and any dependent modules need to be duplicated.

4 Comparison of Approaches

We performed experiments to compare the three approaches. First, we apply direct reflection, in-line expansion and SM duplication to the example in Fig. 1 and compare them by measuring the tests generated for each approach against the other methods. Second, we compare their effectiveness for detecting seeded faults in an implementation of a small portion of *TCAS*.

4.1 Specification-based Coverage

In Table 1, "Mutants" is the total number of syntactically valid mutants, including consistent and duplicate

Method	Mutants	UIMs	UTs
Direct	91	21	9
SM Dupl.	28	21	7
In-line	128	17	10

Table 1: Number of Mutants and Tests.

	Coverage Metric		
Method	Direct	SM Dupl.	In-line
Direct	100%	90%	76%
SM Dupl.	100%	100%	88%
In-line	100%	100%	100%

Table 2: Cross-Scoring of Methods.

mutants. “UIMs” is the number of valid, behaviorally unique, inconsistent mutants. In other words, this excludes all consistent mutants and all but one copy of inconsistent mutants which are semantic duplicates of other mutants. “UTs” is the number of unique counterexamples or tests after duplicates and prefixes of longer counterexamples are removed.

A method can serve both for generation of tests and as a metric for evaluation of existing tests. Specification-based mutation coverage metric was introduced in [2]. We evaluate a method M using a coverage metric C as follows. We generate mutants using method C, but only count unique, inconsistent mutants. Let N be the number of these mutants. We turn the unique counterexamples generated by M into constrained finite state machines (CFSMs) representing individual execution sequences of the state machine [1], then use SMV to find which mutants from C are inconsistent with (*killed* by) at least one CFSM. Let k be the number of mutants killed. The coverage is k/N. A method gets 100% coverage when evaluated against itself as a metric. Table 2 presents cross-coverage of the three methods.

SM duplication method performs better than direct reflection: it kills 100% of direct reflection mutants, while direct reflection kills only 90% of SM duplication mutants. The following example helps explain why.

SM duplication method produces this counterexample to detect the mutant statement (2), Section 3.4:

```
d = 0; b = 0; f = Off;
f = On;
b = 10; f = Off;
```

Each execution step appears on a separate line. Variables not reported are unchanged from the previous step. At the last step, `a` is $1 * 0 + 10 = 10$ and `out` is `Low` in the original state machine, but `a` is $1 * 1 + 10 = 11$ and `out` is `High` in the mutant machine.

Direct reflection method produces this counterexample to detect the corresponding mutant, formula (1), Section 2.2:

```
d = 0; b = 0; f = Off;
f = On;
f = Off;
```

At the last step, the value of the intermediate variable, `a`, is 0, which is inconsistent with the mutant formula. However, when `a` is either 0 or 1, `out` is `Low`. Hence the test will detect the mutant only if intermediate variables are visible.

4.2 Effectiveness in Detecting Faults

Our goal is to reduce the number of faults in programs. Therefore, we evaluate the effectiveness of the methods for detecting seeded faults in a small but realistic program. The subject program is a portion of *TCAS* — aircraft collision avoidance. It is a part of a set of programs that comes originally from [11].

The program consists of 9 procedures and 135 non-blank non-comment lines of C code. There are 12 input variables and one output variable. The program comes with 39 faulty versions derived by manually seeding realistic faults. 26 versions have single mutations, the rest involve either multiple changes or more complex changes.

In Table 3, “Mutants” and “UTs” have the same meaning as in Table 1. “Time” is the time (in seconds) required to generate the tests on a Pentium[2] 4 1.7 GHz PC with 1 GB of RAM running the Linux OS. “Coverage” is the number of faulty versions detected by the method divided by the total number of faulty versions. We used NIST’s Test Assistant for Objects (TAO) [4] to turn the counterexamples into concrete test cases.

Table 3 shows that SM duplication and in-line expansion approaches detect 100% of faulty versions while direct reflection detects only 59% of the faults. We attribute the magnitude of the difference to a relatively large intermediate state of the program.

The in-line expansion method produced by far the largest number of mutants and test cases of the three methods. The SM duplication method generated the smallest number of mutants and test cases, yet it is as effective as the in-line expansion method in detecting

[2]Pentium is a registered trademark of Intel Corporation.

Method	Mutants	UTs	Time	Coverage
Direct	948	83	3.5	59%
SM Dupl.	464	52	9	100%
In-line	3062	139	19	100%

Table 3: Effectiveness in Detecting Seeded Faults

seeded faults. The SM duplication method took considerably longer due to the overhead of starting SMV for every mutant.

5 Conclusion

We presented two new methods, in-line expansion and state machine (SM) duplication, that use a model checker to choose tests which ensure fault propagation to visible outputs. We compared these methods and the previous direct reflection method based on “cross-scoring”. In-line expansion and SM duplication methods got better coverage than direct reflection.

The in-line expansion method is not as useful in practice since it quickly increases the size and number of logic formulas. The SM duplication method duplicates the state machine thus increasing the size of the state space. The running example is tiny and the *TCAS* specification is relatively small, so the limits of scalability have not been addressed. Dependency analysis by slicing is one way to improve scalability.

Our experiments suggest that the SM duplication and in-line expansion methods are much more effective than direct reflection for generating black-box tests. To our knowledge, SM duplication is the first method that relies on a model checker in order to automatically generate tests that guarantee fault propagation to the outputs.

6 Acknowledgments

We thank Paul E. Ammann for encouragement and helpful comments.

References:

[1] P. Ammann, P. E. Black, and W. Ding. Model checkers in software testing. Tech. Rep. NIST-IR-6777, National Institute of Standards and Technology, Feb. 2002.

[2] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proc. Fourth IEEE Internat. High-Assurance Systems Engineering Symp. (HASE 99)*, pp. 239–248. IEEE Computer Society, November 1999.

[3] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. Second IEEE Internat. Conf. on Formal Engineering Methods (ICFEM’98)*, pp. 46–54. IEEE Computer Society, Dec. 1998.

[4] P. E. Black. Modeling and marshaling: Making tests from model checker counterexamples. In *Proc. 19th Digital Avionics Systems Conf. (DASC)*, v. 1.B.3, pp. 1–6, Oct 2000. IEEE.

[5] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proc. 1996 SPIN Workshop*, Aug 1996.

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

[7] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proc. Tenth Internat. Symp. on Software Reliability Engineering*, pp. 210–219, November 1999. IEEE.

[8] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Engineering*, 1(2):156–173, June 1975.

[9] T. Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *Proc. 1993 Internat. Symp. on Software Testing and Analysis*, pp. 171–181, 1993.

[10] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Software Engineering*, 8(4):371–379, July 1982.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Sixteenth Internat. Conf. on Software Engineering*, pp. 191–200, May 1994.

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

[13] V. Okun, P. E. Black, and Y. Yesha. Testing with model checker: Insuring fault visibility. Tech. Rep. NIST-IR, National Institute of Standards and Technology, July 2002.

[14] S. Ranville, 2002. Personal Communication.

[15] D. J. Richardson and M. C. Thompson. An analysis of test data selection criteria using the relay model of fault detection. *IEEE Trans. Software Engineering*, 19(6):533–553, June 1993.

[16] F. Tip. A survey of program slicing techniques. *Programming languages*, 3:121–189, 1995.