Testing with Model Checker: Insuring Fault Visibility

Vadim Okun Paul E. Black
National Institute of Standards and Technology
Gaithersburg, MD 20899
{vokun1,paul.black}@nist.gov

Yaacov Yesha
University of Maryland Baltimore County
Baltimore, MD 21250
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; SMV; software engineering; specification-based testing; state machines; test case generation; fault-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, 6, 10, 13, 25, 17], existing methods allow the model checker to choose tests that do not cause faults to propagate to the program's output.

Goradia [16] presents typical cases that prevent a fault in an intermediate state from propagating to the output.

- The faulty state variable does not participate in a computation that affects the output. Consider this code fragment where variable `output` is the only visible outcome:

```
if (condition) {
    output = state_var;
} else {
    output = 10;
}
```

If, in a specific test scenario, `condition` is `false`, an incorrect value of `state_var` does not impact the output.

- The faulty state variable has an impact on the result of the operation that affects the output, but the impact may not be sensitive to the error represented by the incorrect state. For example, in a predicate such as `state_var > z`, the value of `state_var` affects its result. However, an incorrect value of `state_var` may yield the correct Boolean value of the relational expression.

- Cancelling errors. The faulty state variable may interact with another faulty state variable or itself and thereby yield a correct state. For example, in an expression `x * y`, if both `x` and `y` have incorrect signs, the result will have correct sign.

- An algorithm may tolerate errors in the values of certain variables. Consider a numeric algorithm which computes the local minimum of a polynomial in a given interval by using an iterative procedure that terminates when a specific convergence criterion is satisfied. At the end of each iteration, it obtains the next approximation by adding the value of a variable `step` to the previous approximation. If the value of `step` is faulty, the algorithm may still converge to the correct result by changing the number of iterations.

In this paper we present two 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) [8], which extends propositional logic with temporal operators. Here are two typical CTL formulas together with a short interpretation:

- `AG safe`
  All reachable states are safe.

- `AG (request -> AX response)`
  A request is always followed by a response on the next step.

```
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)
```

Figure 1: An SMV Example

We use SMV, a CTL symbolic model checker [20]. In SMV, a specification consists of one or more modules. One module, named `main`, is the top level module in SMV, serving a role similar to that of the function `main` in C programs. Figure 1 is an SMV example derived from [24]. We refer to it throughout the paper. Variables `d`, `b`, and `f` are inputs and are not constrained. The variables `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`. We consider the output to be the variable `out`, which has possible values `Low` and `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

Although model checking began as a method for verifying hardware designs, there is growing evidence that it can be applied to checking specifications of large software systems, such as TCAS II [7]. In addition to verifying properties of software, model checking is being applied to test generation and test coverage evaluation [3, 6, 10, 13, 25, 17].

In both uses, one first chooses a test criterion [14], that is, decides on a philosophy about what properties of a specification must be exercised to constitute a thorough test. Some specification-based test criteria are conjunctive complementary closure partitions [6], branch coverage [13], and mutation adequacy [2].

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, never in state 3, 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, and so forth.

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

Since counterexamples contain both stimulus and expected values, they usually may be automatically converted to a set of complete test cases [31].

A completely different approach to generating counterexamples is modifying the source code of the model checker to find counterexamples with certain qualities. However, this approach is beyond the scope of the paper.

1.3 Specification Mutation Criterion

Mutation adequacy [9] 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 common mutation operators are:

- replacing a variable with another variable of the same type,
- replacing an integer variable a with $a + 1$ and $a - 1$,
- replacing a conjunction with a disjunction.

Any particular mutation of a logic expression 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. Since consistent mutants do not yield counterexamples, they 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 reviews similar work in program-based testing and protocol testing. It also describes the existing specification-based mutation method and its limitations. Section 3 presents our two approaches: in-line expansion and state machine duplication (SM duplication). Section 4 uses the example in Figure 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 [21, 26, 32, 29, 22, 15]. The RELAY model [26] defines the revealing conditions under which a fault is detected. First, a potential error originates at the smallest subexpression containing the fault, that is, the subexpression evaluates incorrectly. Then the potential error propagates through computations and information flow until a failure is revealed in the outputs. Test data can be selected to satisfy revealing conditions.

Program mutation testing in its original formulation—often referred to as strong mutation—requires the output of a mutant program to differ from the original. Weak mutation [18], on the other hand, only requires that the execution of a component of the mutant and the original produce different values. As we are concerned with visible failures in this paper, we require strong mutation.

Fabbri et. al. [11] devised a mutation model for finite state machines and used the mutation analysis criterion to evaluate the adequacy of the tests produced by standard finite state machine test sequence generation methods. In another paper Fabbri et. al. [12] categorized mutation operators for different components of Statecharts specifications and provided strategies to abstract and incrementally test the components.

The discipline of protocol conformance testing [5] involves testing an implementation against the protocol specification. Often, the tester has little or no access to the internal states of a protocol implementation because it is running on remote machines or only its executable code is available (no source code). Tests must be selected to cause discernible results in the visible output.

Similarly, one may be interested in testing just a single component of a modular system. The ccontext of the component represents the rest of the system. Testing the whole system results in an unnecessarily large test set, while testing the component in isolation raises the problems of test executability (e.g., is the test allowed by the context?) and fault propagation (e.g., is a fault tolerated by the context and thus cannot be detected externally?).

[23] addressed these problems for the case of a system modeled as a collection of communicating finite state machines, of which one is the specification of the component to be tested; the rest form the context. Testing in context is reduced to testing in isolation by way of computing an approximation of the specification in context. The approximation is a nondeterministic finite state machine model of the component's properties that can be controlled and observed through

the context; the behavior of every conforming deterministic implementation is included in the approximation. The tests derived from the approximation are executable and guarantee fault propagation. In our work we rely on the model checker to achieve these goals.

Finite state machine models can only specify behaviors within the domain of regular languages. [30] proposed a test suite generation method for protocols specified by extended finite state machines where transitions are associated with actions such as assignments, conditional expressions, input/output, etc. The method assumes that the status of an implementation under test cannot be modified or observed directly, but only by examining the sequences of input and output events. Such observable events are recorded so that the resulting test cases can be applied directly to implementations running on real machines. In this method, an axiom defining the semantics of the actions is associated with each action type. Assertions (preconditions and postconditions) are updated according to the axioms. Assertions consist of a sequence of external (input and output) events appearing along the traversed path, a set of predicates valid at the current state of the extended finite state machine, and variables that need to be observed through the output events in order to confirm correctness of a preselected transition. The method detects any single transition mutants, i.e., mutants where one transition leads to an incorrect state.

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].

Figure 2 contains formulas derived from the assignment statements in Figure 1. For instance, the `next` clause for the variable `e` in Figure 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. These five formulas may be mutated, and the mutants analyzed by a model checker.

```
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)
```

Figure 2: Applying Direct Reflection

A mutant models a fault in the specification. 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 Figure 2:

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

Choosing `b = 0`, `d = 0`, and `f = On` is sufficient to show an inconsistency in an intermediate variable `a`, but not in the output variable `out`. Such a test is of little practical 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 in-line copies of their transition relations. This substitution is performed repeatedly until the formulas are comprised exclusively of input variables. Figure 3 contains formulas derived from the statements in Figure 1 using in-line expansion method. Compare these with Figure 2. Since only inputs and externally visible variables appear, the model checker finds counterexamples that affect the external variables. As in direct reflection, all mutants can be checked against the original state machine in a single run.

```
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))
```

Figure 3: Applying In-line Expansion

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 Figure 1 has two conditional statements, each with two branches, for a total of four possible paths, so there are four formulas in Figure 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, or syntactic change. SM and SM_d have separate sets of output variables. 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 a value for the output.

From the counterexample, we can construct a test case containing values for inputs and the expected values for the outputs from 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 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 in Figure 4 means that the next value for `var` may be either 1 or 2 if `condition` is true.

```
next(var) := case
  condition : {1, 2};
  1 : 0;
esac;
```

Figure 4: Nondeterminism in SMV

Nondeterminism is expressed in the state machine description language of SMV by giving a set of values for the result of an expression. 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 must force SM and SM_d to make the same choices when they have a nondeterministic choice. We achieve this by declaring a new variable globally for each nondeterministic choice. We modify both SM and SM_d to choose depending on this common global variable.

For the assignment statement in Figure 4, we declare a common unconstrained variable: `coin : {1, 2};`. We then modify both SM and SM_d to have this statement:

```
MODULE original(d, b, f)
VAR
  out: {Low, High};
  a: 0..16; e: 0..1;
ASSIGN
... same transitions as in Figure 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)
```

Figure 5: A Duplication Example

```
  next(var) := case
    condition : coin;
    1 : 0;
  esac;
```

While this method is general, it is excessive for variables without explicit transition, such as inputs: there are still no guards or clauses to mutate. In this case, we can simply move declarations of such variables into the `main` module and pass them to SM and SM_d as parameters. Figure 5 is an example of sharing input variables.

3.4 An Illustrative Example

Consider the sample model in Figure 1. As Figure 5 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. If we wish to avoid passing each parameter separately, we can use a feature of SMV that allows to pass an instance of a module (`main` in this case) as a parameter.

The CTL formula asserts that outputs of the original and mutant modules are always the same. If there are several output variables, the assertions can be given in different ways, such as in Figure 6. If there is one `SPEC` clause for each

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

output, as in Figure 6(b), more counterexamples are likely. The conjunction in Figure 6(a) makes the model checker find one counterexample for each mutant. That counterexample needs only have one output differ between the original and the mutant. In contrast, with one clause per output, the model checker tries to find a counterexample for each output for each mutant. Since a mutant rarely affects all outputs, counterexamples would not be found for all mutants and outputs. We have not investigated the number of unique counterexamples produced or any differences in coverage from the two styles.

```
SPEC AG (good.out1 = mutant.out1 & good.out2 = mutant.out2 & ...)
```

(a) A Combined Clause

```
SPEC AG (good.out1 = mutant.out1)
SPEC AG (good.out2 = mutant.out2)
...
```

(b) One Clause per Output

Figure 6: Specifying Multiple Outputs

Assignment statements in the `duplicate` module from Figure 5 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 Figure 1.

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

The right hand side of the mutant may evaluate to a value that is greater than the maximum allowed value of `a`, which was declared to be `16`. To fix this, we change the declaration of `a` in the mutant to expand its range when needed.

3.5 Duplicating processes

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.

We can assign the original and mutant processes unique `id` numbers, for instance, 0 and 1. We pass Boolean variables, `turn` and `valid`, to the processes. `Turn` is initially 0. Each process changes it so that on the next step it is equal to the `id` of the other process. Variable `valid` becomes false if the processes are ever executed out of order, thus telling SMV to disregard other orderings.

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.

The following CTL formula asserts that outputs of the original and mutant modules are the same after the second process executes, if the processes executed in order.

```
AG (turn = 0 & valid -> good.out = mutant.out)
```

3.6 Sharing Independent Variables

Some parts of the model may not depend on the variable affected by a particular mutation. These parts do not need to be duplicated. Strictly speaking, for any particular mutation, we need only duplicate the variable whose assignment is being mutated and any dependent variables. Dependency determinations can stop at output variables. Such dependency can be determined using program slicing [28]. If the model has many modules, only the module with the mutation and any dependent modules need to be duplicated. For large models with limited feedback, this may save enough model checking time to be worth the dependency analysis.

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 small example in Figure 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 mutants, including consistent and duplicate 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.

	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.

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.

The SM duplication and in-line expansion methods perform better than direct reflection: each of them kills 100% of direct reflection mutants, while direct reflection kills only 90% of SM duplication mutants and 76% of in-line expansion 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;
```

For counterexamples, we present only values for the input variables. Each execution step appears on a separate line. Variables not reported are unchanged from the previous step. At the last step in the original state machine `a` is $1*0+10=10$ and `out` is `Low`, while in the mutant machine `a` is $1*1+10=11$ and `out` is `High`.

In the in-line expansion method, there are two corresponding mutants:

```
AG (f=On -> AX (d+1+b > 10 -> out=High))
AG (f=On -> AX (d+1+b <= 10 -> out=Low))
```

The second mutant is consistent with the state machine. The first mutant produces a counterexample that is exactly the same as the counterexample produced by the SM duplication method. Both methods force SMV to choose a value of `b` that causes a visible difference in the output.

In contrast, we saw in Section 2.2 that the direct reflection method produced the following test to detect the corresponding mutant, formula (1):

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

The value of the intermediate variable, `a`, is 0, while the mutant expects it to be 1 in the final step, thus the test shows that the mutant is inconsistent with the state machine. However, when `a` is either 0 or 1, `out` is `Low`. Hence the test will detect a mutant in the implementation only if intermediate variables are visible.

The in-line expansion method has 100% coverage against any of the three metrics. The method can be thought of as a version of path coverage, a very powerful criterion. Consider a mutant of the statement for variable `out` where "`a > 10 : High`" is replaced with "`a < 10 : High`". SM duplication method will produce the following single-step test to detect the mutant:

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

In the in-line expansion method, there are two corresponding mutants:

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

While the second mutant can be detected by the same single-step test, the first mutant requires setting `f = On`. The method requires to test different paths.

It is theoretically possible to combine SM duplication with in-line expansion. The combined method will eliminate the intermediate variables and present the transition relation of the state machine in terms of input and output variables. However, this will have the disadvantage of the in-line expansion method, namely, the formulas can grow exponentially.

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 *TCAS* – aircraft collision avoidance. It is a part of a set of programs that comes originally from Siemens Corporate Research [19] and was subsequently modified by Rothermel and Harrold [27]. These programs are used in research on program testing, so they come with extensive test suites and sets of faulty versions.

The program consists of 9 procedures, 135 non-blank non-comment lines of C code. There are 12 input variables specifying parameters of own aircraft and another aircraft and one output variable—`alt_sep`—a resolution advisory to maintain safe altitude separation between the two aircrafts. The program computes intermediate values and prints `alt_sep` to the standard output. We wrote a formal specification for the program in SMV.

The program comes with 39 faulty versions derived by manually seeding realistic faults. 26 versions have single mutations such as replacing a constant with another constant, replacing $\geq$ with $>$, or dropping a condition. The rest involve either multiple changes or more complex changes.

We ran experiments on TCAS to compare the methods in terms of the number of test cases produced and the effectiveness in detecting seeded faults. In

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

Table 3: Effectiveness in Detecting Seeded Faults

Table 3, "Mutants" is the total number of syntactically valid mutants, including consistent and duplicate mutants, "UTs" is the number of unique counterexamples or tests after duplicates and prefixes of longer counterexamples are removed. "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. When provided with the correspondence between specification variables and function calls on the implementation level, TAO generates code to create new test instances, call the interface functions to set and get values, make sure the specified conditions hold, and report any differences between produced and expected results.

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 seeded faults.

The time required to generate tests using the direct reflection method was 3.5 seconds, SM duplication – 9 seconds, in-line expansion – 19 seconds. We used a 1.7 GHz Pentium 4[2] PC with 1 GB of RAM running the Linux OS. The SM duplication method took considerably longer due to the overhead of starting SMV and building the state machine model for every new 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

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

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 black-box testing, that is, where the tester does not have access to the intermediate variables in the program. 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] Paul Ammann, Paul E. Black, and Wei Ding. Model checkers in software testing. Technical Report NIST-IR-6777, National Institute of Standards and Technology, February 2002.

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

[3] 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.

[4] Paul E. Black. Modeling and marshaling: Making tests from model checker counterexamples. In *Proceedings of the 19th Digital Avionics Systems Conference (DASC)*, volume 1.B.3, pages 1–6, Philadelphia, Pennsylvania, October 2000. IEEE.

[5] G. Bochmann and A. Petrenko. Protocol testing: Review of methods and relevance for software testing. In *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, pages 109–124, 1994.

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

[7] 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.

[8] 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.

[9] Richard A. DeMillo, 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.

[10] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Ed Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer-Verlag, April 1997.

[11] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Mutation analysis testing for finite state machines. In *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, pages 220–229, Monterey, CA, November 1994. IEEE.

[12] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proceedings of the Tenth International Symposium on Software Reliability Engineering*, pages 210–219, Boca Raton, Florida, November 1999. IEEE.

[13] 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.

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

[15] Tarak Goradia. *Dynamic Impact Analysis: Analyzing Error Propagation in Program Executions.* PhD thesis, Dept. of Computer Science, New York University, 1988.

[16] Tarak Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 171–181, 1993.

[17] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Sung Deok Cha. Automatic test generation from statecharts using model checking. Technical Report MS-CIS-01-07, University of Pennsylvania, 2001.

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

[19] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200, May 1994.

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

[21] Larry J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, Aug 1990.

[22] A. J. Offutt. *Automatic Test Generation*. PhD thesis, Dept. of Information and Computer Science, Georgia Institute of Technology, 1988.

[23] A. Petrenko, N. Yevtushenko, G. Bochmann, and R. Dssouli. Testing in context: framework and test derivation. *Special Issue on Protocol Engineering of Computer Communication*, 1997.

[24] Scott Ranville, 2002. Personal Communication.

[25] Sanjay Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checkers. In 8^{th} *Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Washington, DC, April 2001.

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

[27] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, 1998.

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

[29] Jeffrey M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8), August 1992.

[30] Chang-Jia Wang and Ming T. Liu. A test suit generation method for extended finite state machines using axiomatic semantics approach. In R. J. Linn, Jr. and M.U. Uyar, editors, *Protocol Specification Testing and Verification, XII*, pages 29–43. Elsevier Science Publishers B.V. (North-Holland), 1992.

[31] Duminda Wijesekera, Lingya Sun, and Paul Ammann. Relating counterexamples to test cases in CTL model checking specifications. Unpublished, 2002.

[32] Steven J. Zeil. Perturbation techniques for detecting domain errors. *IEEE Transactions on Software Engineering*, 15(6):737–746, June 1989.