Using Model Checking to Generate Tests from Specifications

Paul E. Ammann Paul E. Black William Majurski
I&SE Dept. *National Institute of*
George Mason University *Standards and Technology*
Fairfax, Virginia 22033 *Gaithersburg, Maryland 20899*
pammann@gmu.edu *paul.black@nist.gov* *bill@nist.gov*

Abstract

We apply a model checker to the problem of test generation using a new application of mutation analysis. We define syntactic operators, each of which produces a slight variation on a given model. The operators define a form of mutation analysis at the level of the model checker specification. A model checker generates counterexamples which distinguish the variations from the original specification. The counterexamples can easily be turned into complete test cases, that is, with inputs and expected output. We define two classes of operators: those that produce test cases from which a correct implementation must differ, and those that produce test cases with which it must agree.

There are substantial advantages to combining a model checker with mutation analysis. First, the generation of test cases is automatic; each counterexample serves as a complete test case. Second, in sharp contrast to program-based mutation analysis, the identification of equivalent mutants is also automatic; the model checker simply reports that the mutant satisfies the constraints, and hence no counterexample is produced. We apply our method to an example specification and evaluate the resulting test sets with coverage metrics on a corresponding implementation in Java.

1 Introduction

The use of formal methods has been widely advocated to reduce the likelihood of errors in the early stages of system development. Some of the chief drawbacks to applying formal methods is the difficulty of conducting formal analysis [5] and the perceived or actual payoff in project budget. Testing is an expensive part of the software budget, and formal methods offer an opportunity to significantly reduce the testing costs. We have developed an innovative combination of mu-

tation analysis, model checking, and test generation which solves some problems previously plaguing these approaches and automatically produces good sets of tests from formal specifications. This section reviews the formal methods and approaches we use.

Generating test inputs, even in sophisticated and constrained combinations, is straight-forward. However deriving the corresponding expected outputs, or equivalently coming up with an oracle to determine if the result is right, is often labor intensive. To be clear about this point, we define a *test case* to be both a set of inputs or stimulus and the expected output or response. We also use the term *complete test case* to emphasize that it includes inputs and outputs. Our approach uses formal specifications to automatically generate complete test cases.

Mutation analysis [9] is a white-box method for developing a set of test cases which is sensitive to any small syntactic change to the structure of a program. The rationale is that if a test set can distinguish a program from a slight variation, the test set is exercising that part of 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 mutation analysis, since identifying equivalent mutants is, in general, an undecidable problem. Typical proofs of this fact reduce equivalent mutant detection to the halting problem.

SCR (Software Cost Reduction) [14] is a method used to formally capture and document the requirements of a software system. The SCR method is scalable and its semantics are easy to understand; this accounts for the use of the SCR method and its derivatives in specifying practical systems [11, 13, 19].

Research in automated checking of SCR specifications includes consistency checking and model checking. The consistency checker of Heitmeyer, Jeffords, and Labaw [12] analyzes application independent properties such as syntax, type mismatch, missing cases, circular dependencies and so on, but not application dependent properties such as safety and security. Atlee's model checking approach [1, 3, 4] addresses the application dependent property of safety in SCR mode transition tables by expressing an SCR mode transition table as a logic model, expressing the safety properties of the specification as logic formulae, and using a model checker to determine if the formulae hold in the model. Owre, Rushby, and Shankar [18] describe how the model checker in PVS can be used to verify safety properties in SCR mode transition tables.

A model checking specification consists of two parts. One part is a state machine defined in terms of variables, initial values for the variables, and a

description of the conditions under which variables may change value. The other part is invariant conditions and temporal logic constraints on possible execution paths. Conceptually, a model checker visits all reachable states and verifies that the invariants and temporal logic properties are satisfied. Model checkers exploit clever ways of avoiding brute force exploration of the state space, for example, see [6]. In cases where a property is not satisfied, the model checker attempts to generate a counterexample in the form a 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 execution path leads to that state, there is no counterexample to exhibit.

The model checking approach to formal methods has received considerable attention in the literature, and readily available tools such as SMV and SPIN [15] are capable of handling the state spaces associated with realistic problems [8]. We selected the SMV model checker. It is freely available from Carnegie Melon University at `http://www.cs.cmu.edu/groups/formal-methods/formal-methods.html`.

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. Human interaction is generally required to prove all but the most trivial theorems; model checkers can explore the state spaces for finite, but realistic, problems without human guidance.

We are not the first to recognize that the counterexamples from model checkers are potentially useful test cases. In particular, Callahan, Schneider, and Easterbrook use the SPIN model checker to generate tests that cover each block in a certain partitioning of the input domain [7]. However, we understand we are the first to define a mutation model in the context of a model checker and automatically generate mutation adequate tests.

Sect. 2 presents the example we used as a vehicle for study. Sect. 3 explains our approach including details of the tools we use. Sect. 4 gives results in terms of tests generated and coverage. Finally Sect. 5 details our plans for future work, and our conclusions are in Sect. 6.

2 Cruise Control Example

We use Cruise Control [16] as a small example with some complexity. Many variations on this example exist; we use one from Atlee [3]. The SMV logic model we use is derived from one generated by Atlee's tool [2].

Table 1 is the state machine description. The safety invariants are given in Table 2, and these are the environmental assumptions.

Current Mode	*Ignited*	*EngRun*	*TooFast*	*Brake*	*Activate*	*Deactivate*	*Resume*	New Mode
Off	@T	—	—	—	—	—	—	*Inactive*
Inactive	@F	—	—	—	—	—	—	*Off*
Inactive	—	t	f	f	@T	—	—	*Cruise*
Cruise	@F	—	—	—	—	—	—	*Off*
Cruise	t	@F	—	—	—	—	—	*Inactive*
Cruise	t	—	@T	—	—	—	—	*Inactive*
Cruise	t	t	f	@T	—	—	—	*Override*
Cruise	t	t	f	—	—	@T	—	*Override*
Override	@F	—	—	—	—	—	—	*Off*
Override	t	@F	—	—	—	—	—	*Inactive*
Override	t	t	f	f	@T	—	—	*Cruise*
Override	t	t	f	f	—	—	@T	*Cruise*

Initial State : Mode = Off, $\neg Ignited$

Table 1: Mode Transition Table for $CruiseControl$

$$EngRun \Rightarrow Ignited$$
$$TooFast \Rightarrow EngRun$$
$$Activate \mid Deactivate \mid Resume$$

We chose to work on the specification at the level of the input to SMV which is a text format, because we want our approach to apply to any situation where a model checker can be used. We specifically do not wish to limit application to a single specification notation, such as SCR. In the SMV logic model, the seventh line of Table 1 is represented as the following.

```
next(CruiseControl) := case
          .
          .
   CruiseControl=Cruise & Ignited & EngRun &
            !Toofast & !Brake & next(Brake) : Override;
          .
          .
esac;
```

Mode	Safety Invariant
Off	$\neg Ignited$
$Inactive$	$Ignited$
$Cruise$	$Ignited \wedge EngRun \wedge \neg Toofast \wedge \neg Brake \wedge \neg Deactivate$
$Override$	$Ignited \wedge EngRun$

Table 2: Safety Invariants for $CruiseControl$

The safety invariants of Table 2 are represented in SMV as follows.

```
SPEC AG(CruiseControl=Off -> !Ignited)
```

```
SPEC AG(CruiseControl=Inactive -> Ignited)
SPEC AG(CruiseControl=Cruise -> (Ignited & EngRun &
                        !Toofast & !Brake & !Enum1=Deactivate))
SPEC AG(CruiseControl=Override -> (Ignited & EngRun))
```

The SMV notation `A` is the universal quantifier, and the notation `G` means "global." Hence, `AG` specifies properties which should hold in every state on all possible traces. That is, `AG` specifies an invariant. Logical conjunction is represented by `&`, `->` is implication, and `!` is logical negation.

The explicit mode transitions are repeated as temporal logic constraints. For instance, here is the logic equivalent to line seven.

```
SPEC AG(CruiseControl=Cruise -> AX((!PBrake & Brake &
                PIgnited & PEngRun & !PToofast)
                                -> CruiseControl=Override))
```

`X` is the next state operator. Variables with a "P" prefix are values from the previous state. They do not appear in the SCR mode transition table, and do not necessarily appear in an implementation.

3 Approach

Figure 1 shows our overall approach. In actual use one would begin with some system specifications and, through finite modeling and possibly other tools, turn them into specifications suitable for a model checker. After this point all processing can be automatic.

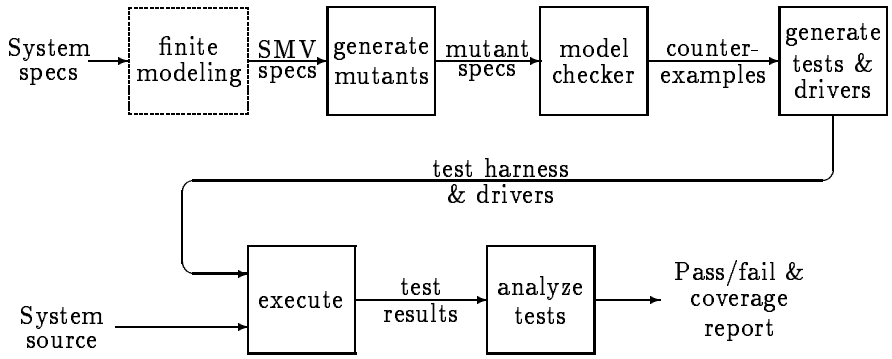

Figure 1: Overall System Flow

We apply mutation operators to the state machine or the constraints yielding a set of mutant specifications. The model checker processes the mutants one at a time. When the model checker 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 dropping any counterexample which is a "prefix" of another, longer counterexample. The counterexamples contain both stimulus and expected values so they may be automatically converted to complete test cases. The test cases generate executable test code, including a test harness and drivers.

This test code is executed with implementation source which is instrumented to record coverage. The test code records which tests pass and which fail. These test coverage results are processed to become a final report of coverage, to show how comprehensive the tests are. We also check that the implementation fails the cases it should fail, and passes those it should pass.

The following sections explain in more detail how we apply mutation analysis for model checkers, the mutation operators we use, how we extract and reduce counterexamples and prepare them to be test cases, the tool for generating tests, and our measures of code coverage.

3.1 Mutation Analysis

Each mutation operator specifies a pattern for a small syntactic change to a program's structure. For example, the "wrong variable" operator replaces a single occurrence of a variable in a program with another variable of compatible type. The program text `if a < b then ...` might be transformed to `if c < b then ...` The "wrong relational operator" mutation operator replaces an occurrence of $<, \leq, =, \neq, \geq$ or $>$ with one of the other five possibilities. The program text `if a < b then ...` might be transformed to `if a = b then ...` by replacing the $<$ operator with the $=$ operator.

Program-based mutation analysis relies on the competent programmer hypothesis, which states that competent programmers are likely to construct programs close to the correct program, and hence test data that distinguish syntactic variations of a given program are, in fact, useful. In the current work, we assume an analogous "competent specifier hypothesis," which states that the specifier of an SMV model is likely to describe a state transition system and set of temporal logic constraints that are close to what is desired, and hence test data that distinguish syntactic variations of a given SMV specification are, in fact, useful.

Equivalent mutants present a serious problem for mutation analysis, since an equivalent mutant cannot be distinguished and should be ignored, but all non-equivalent mutant should be considered. In practice, some analysis by humans is required to distinguish between equivalent and non-equivalent mutants. Alternatively, in an approximation of mutation analysis, some fraction of the

mutants are simply left unexamined. As it turns out, there is a similar undecidability problem lurking in all of the popular coverage metrics, notable the path coverage metrics of statement and branch coverage, data-flow coverage metrics, and condition coverage metrics [20]. One of the interesting aspects of the current work is that we evade the undecidability problem by working in the finite state space of the model checker. Not only is equivalent mutant identification possible in the context of a model checker, but model checkers are optimized to perform this equivalency check efficiently.

A test input distinguishes a mutant from the original program if the mutant and the original program produce different outputs. (Programs are assumed to be deterministic.) A test set is mutation adequate if at least one test in the test set distinguishes each nonequivalent mutant. There are test data generation systems that, modulo the ever present undecidability problem, attempt to automatically generate mutation adequate test inputs [10].

Mutation Analysis for Model Checkers

In the present work, we switch contexts from program based testing to specification based testing, and we consider specifications expressed as finite models within the context of a model checker. Clearly, we require a set of mutation operators that make sense in the model checking context. Two broad categories of mutation operator present themselves:

1. Changes to description of the finite state machine transition. Here, counterexamples are sequences of states from which a good implementation should diverge. That is, when given the sequence of input changes specified by such a test, a correct implementation should respond with different outputs than those recorded in the model checker's counterexample. We refer to these as *failing* tests, or tests which we expect our implementation to fail.

2. Changes to the temporal logic constraints imposed on the model. In this category, counterexamples are sequences of states which are acceptable in the original model, and hence a correct implementation should follow them. We refer to these as *passing* tests: we expect our implementation to pass these tests.

The goal of the present investigation is to show the feasibility of our approach. Specifically, we wish to show that it is possible to define a useful set of mutation operators on an SMV specification and subsequently generate a useful set of tests via counterexamples to mutations of the SMV specification. We define and use the following mutation operators. This list is clearly not exhaustive, but it does produce a set of mutants with reasonable coverage properties, as we document subsequently.

3.2 Mutation Operators

Explicit transitions in SMV have the following form:

```
next (v) := case c1:  e1;   c2:  e2; ...ecase;
```

where each c_i is a predicate on variables declared in the model and each e_i is a corresponding new value for variable v.

We define a mutation operator M_1 that changes condition c_i by conjoining an additional condition $w = e$ for some variable w and some possible value e. We define another mutation operator M_2 that changes those conditions c_i which are multiple conjoined conditions by deleting one of the conditions. Other candidate mutant operators that we may use are changing the new value e_i to some other possible value, deletion of $c_i : e_i$; pairs, and replacing variables in some c_i with other variables of compatible type.

All test cases from M_1 or M_2 are "failing" cases. That is counterexamples are sequences of state transitions from which a correct implementation must diverge at some point. In other words, if the implementation makes exactly the same transitions and has the same values as the test case, the implementation necessarily violates one or more of the temporal logic constraints in the SMV specification.

We also define mutation operators on the invariants and temporal logic constraints over the possible states of the SMV model.

We define mutation operators M_3 and M_4 on mode references. Specifically, operator M_3 replaces constraints of the form:

```
SPEC AG(CruiseControl = mode_i -> ...
```

with

```
SPEC AG(CruiseControl = mode_j -> ...
```

where $mode_i$ and $mode_j$ differ.

We define operator M_4 to replace constraints of the form:

```
SPEC AG(CruiseControl=mode_i & AX(c1 & (CruiseControl=mode_j)))
```

with

```
SPEC AG(CruiseControl=mode_i & AX(c1 & !(CruiseControl=mode_j)))
```

In terms of the SCR mode transition table that gave rise to the SMV specification, operator M_4 replaces the destination mode with an incorrect mode.

Both M_3 and M_4 result in "passing" test cases. That is they yield sequences of state transitions with which a correct implementation must agree.

There are many possible other mutation operators on the temporal logic specifications, such as replacing conjunctions with disjunctions, adding conditions, deleting conditions, etc.

Since we are interested in generating test cases in this work, we limit our attention to temporal logic properties that, when violated, yield explicit counterexamples. If a state machine violates an existential assertion, no useful trace is produced. For instance, suppose the mutated transition table has no way to reach *Cruise* mode but we assert that there must exist some way to reach it. The only possible counterexample would be traces for *every* possible state, and a review to show that *no* execution reached *Cruise* mode. In fact, SMV doesn't produce a counterexample for violations of existential assertions. For efficiency, we removed all existential assertions since they don't produce test cases.

3.3 Preparing Counterexamples

SMV produces one output for each input, or in our case, mutant. A program scans the output for counterexamples. When any are found, they are extracted. Figure 2 is a typical part of an SMV output.

```
-- specification AG(CruiseControl=Off -> AX(!PIgnited... is false
-- as demonstrated by the following execution sequence
state 1.1:
Ignited = 0
EngRun = 0
Toofast = 0
Brake = 0
Enum1 = Resume
CruiseControl = Off
PBrake = 0
PToofast = 0
PEngRun = 0
PEnum1 = Resume
PIgnited = 0

state 1.2:
Ignited = 1
```

Figure 2: Typical Counterexample

Variables with a "P" prefix, such as `PToofast`, are references to temporally previous values; we ignore them. Values of dependent variables, such as `CruiseControl`, must be propagated since SMV only prints values which change. Numbered states are replaced by explicit calls to "step" the implementation.

Duplicate traces are combined. To further reduce the test set any trace for a "passing" test which is a prefix of another passing trace is discarded since it is

unnecessary. Suppose there is a passing trace $\mathcal{A}$, which is `F G`, and that there is another, longer passing trace $\mathcal{B}$, which is `F G H`. Anything exercised by trace $\mathcal{A}$ will also be exercised by trace $\mathcal{B}$, since we are modeling deterministic processes.

For failing traces, prefix traces should be retained and longer traces discarded. This is because if some prefix, say `F G` is incorrect, so is a longer trace, say `F G H`. We have found no prefixes among failing traces. We believe this is because SMV checks shorter sequences first.

3.4 Generating Test Code

TDA (Test Data Assistant) [17] is a software application program interface (API) testing environment under development in the Information Technology Laboratory at NIST. TDA integrates two popular test specification technologies, context free grammars and constraint satisfaction, into a single software test generation tool. The first version of this tool operated on C language APIs. A new version under development tests Java classes.

A TDA test specification is made up of five parts:

1. a reference to the API under test,
2. test scenarios,
3. a test grammar,
4. constraints on inputs, and
5. assertions about outputs.

Information about the API under test is obtained by scanning source files. A test specification is made up of test scenarios, or, sequences of calls on the API or object under test. Each scenario is defined by a start symbol in the test grammar. Traditional grammar notations such as alternation ($|$), Kleene star ($*$), or "one-or-more" ($+$), are used to construct test grammars. Input parameters such as numbers or strings are either specified in the grammar or are generated randomly. Input generation can be directed by sets of constraint equations which describe the relationships between inputs. Assertions describe expected results in terms of outputs and accessible internal state.

Each test scenario defines one test sequence grammar with terminals which are calls to the API. A resultant sequence can build internal state within the software being tested via API calls and validate assertions about internal state or results. Instead of testing individual API calls separately (in isolation), we operate on the whole context of the API or object. Generating and processing test inputs automatically but validating results manually is difficult or infeasible. We refer to the part of a testing system that automatically validates results as the *test oracle*. The assertions section of the test grammar constitutes an oracle, and addresses the oracle problem by generating run-time results checking.

The output of TDA is a collection of Java source code files that embody the tests described in the test specification including the oracle functions and a test harness. The test harness is the controlling program that runs each test scenario, evaluates the outputs, and reports results to the user.

One test scenario is generated for each trace. Since a trace gives an explicit list of calls, the grammar is just the calls with input parameters. Expected outputs are written as constraints on each particular state machine step.

Why use a tool like TDA to orchestrate testing? TDA offers a consistent interface to a rich test environment for evaluating software APIs. Its input language is designed to be an intermediate language between testing tools, although it can be easily written by hand. For our project, it offers a single simple interface to test specification, management, and reporting features independent of the language being analyzed. A graphical interface will be available for TDA in the future.

3.5 Code Coverage

To judge the value of our approach and to determine which mutation operators to use, we need to measure the quality of the generated tests. Coverage analysis on an implementation is a common way to measure the quality and judge test adequacy [20]. There are many different coverage methods and metrics, but the general categories are statement coverage, branch coverage, data-flow coverage, path coverage, and mutation adequacy. Basically statement coverage checks which statements are executed, and branch coverage checks that all possible outcomes of a branch are executed. Data-flow coverage is a measure of executing paths between creation, modification, and uses of data values, and path coverage is a measure of executing some syntactical or semantically defined paths. Mutation adequacy checks that the tests "kill" all non-equivalent mutants of a program.

To briefly explain the differences between these different types of coverage, consider the following example of code.

```
if (P) {
    b = c;
}
if (Q) {
    d = e;
}
```

Statement coverage is satisfied if the two conditionals and the two assignments, `b = c` and `d = e`, are executed. This can be done with one test if `P` and `Q` are both true. Branch coverage is satisfied if both outcomes of the conditionals are executed. This would require at least two tests: for instance, when `P` and `Q` are both true and when they are both false. Simple path coverage would require at least four tests: one for each combination of `P` and `Q`. Data-flow requires that

every path between, say, the definition of e and its use in the assignment are executed. There are many variations and refinements defining different types of data flows and paths, examining the evaluations of predicates, etc.

We chose to examine branch coverage, which subsumes statement coverage, for now. We implemented the cruise control module in Java and used an evaluation copy of Sun Microsystems' JavaScope 1.1 to check coverage[1]. The implementation is less than 100 lines of code including comments. Here is a portion of the code (with coverage results and some omissions); we refer to it in the next section.

```
if (! Ignited) {
    mode = ccMode.Off;
    return;
}

if (! EngRun) {
    mode = ccMode.Inactive;
    return;
}

switch (mode) {
  case ccMode.Off:
    if (Ignited) {
>>>>             ^
>>>>     false branch at line 31 column 17 is NOT covered.
        mode = ccMode.Inactive;
    }
    break;

  case ccMode.Inactive: ...

  case ccMode.Cruise: ...

  case ccMode.Override:
    if (! TooFast && ! Brake) {
        if (lever==ccLever.Activate || lever==ccLever.Resume) {
>>>>             ^
>>>>     false branch at line 54 column 21 is NOT covered.
            mode = ccMode.Cruise;
        }
    }
    break;
}
```

[1] Java and JavaScope are trademarks of Sun Microsystems, Inc.

4 Results

Table 3 gives the coverage for each mutation operator. It also gives the coverage for all failing tests, all passing tests, and all tests run together. The "mutants" column is the total number of mutants generated, and "counterexamples" is the total number of counterexamples found in the SMV runs. "Unique traces" is the number of traces after duplicates and prefixes are removed (see Sect. 3.3 for details). The number of unique tests for failing tests and passing tests is not the sum of those for $M_1 + M_2$ and $M_3 + M_4$ respectively because of duplicates.

Briefly, M_1 and M_2 change the state machine's transitions. Recall that a correct implementation must fail these test cases. M_1 adds a condition to a transition, and M_2 deletes a condition from a transition. (A precise description is in Sect. 3.2) M_3 and M_4 change the temporal logic constraints. A correct implementation must pass these test cases. M_3 changes the condition of a temporal logic constraint, and M_4 negates the new mode.

Operator	Mutants	Counter-examples	Unique Traces	Block (of 12)	Branch (of 16)	Logical (of 21)
M_1	168	192	34	12	13	11
M_2	56	31	27	11	13	16
failing tests	224	223	60	12	14	16
M_3	41	13	5	10	10	10
M_4	10	10	7	12	11	9
passing tests	51	23	11	12	13	14
All	275	246	71	12	14	17

Table 3: Coverage Results of Mutation Operators

The "Block" column gives the number of basic blocks which the tests execute, and "Branch" gives the number of branch possibilities executed. For interest, we also provide the number of logical possibilities covered in the last column. The number of blocks, branch possibilities, and logical possibilities covered for all failing tests, passing tests, and everything do not sum because of overlap.

As discussed earlier, the model checker generates counterexamples that are necessarily mutation adequate with respect to the mutation operators applied. In JavaScope a test set achieves complete branch coverage if the guard in each `if` statement evaluates to true for some test case in the test set and to false for some test case. JavaScope evaluates coverage for `switch` statements separately; all of the test sets achieved full switch coverage, so we do not discuss switch coverage further.

All together the test cases cover 14 of the 16 branches in the implementation. Further analysis revealed that one of the uncovered branches, labeled "`false`

`branch at line 31`," is actually infeasible. That is, *no* test input can cause this branch to be executed, since the false condition is intercepted earlier. Therefore, this branch may be removed from the coverage analysis. Program structures that cannot be covered are a frequent occurrence in any code coverage metric.

The remaining uncovered branch, labeled "`false branch at line 54`," is, in fact, feasible. Why didn't any test case generated by the model checker cover this particular branch? Because it requires a test case with a transition that does not change the mode. However, the original SCR specification is a mode transition table; that is, it explicitly specifies the conditions under which a mode change. The conditions under which a mode does *not* change are implicit. The particular SMV logic model we used for this SCR specification maintains the distinction between explicit and implicit specification. The uncovered branch corresponds to an implicit aspect of the specification, and hence, no mutation of the (explicit) specification is likely to force a test to cover this branch.

Making implicit parts of a specification explicit is a standard aspect of many formal methods. If applied to the SCR mode transition or the corresponding SMV logic model, the result would be mutants that covered this last branch. We conclude that explicit specifications are definitely desirable from the perspective of test coverage.

5 Future Work

We plan to devise other mutation operators and determine which set of mutation operators give the best coverage with the smallest set of tests.

We also note that some mutation operators could change the environmental assumptions. Any counterexamples show when an implementation might behave incorrectly or unsafely in an unexpected environment. If the implementation passes resulting tests, it increases our confidence in its robustness. If the implementation fails, we can reexamine the assumption. We may decide that the system needs to be engineered stronger to assure that the assumption will not fail in the real world.

In the next phase of this project we plan several extensions. We will apply this technique to a much larger and richer specification, the Java SmartCard. We also plan to explore starting with higher level specifications, say in Z or operational semantics, and automatically generating model checker specifications. Finally we are looking for industrial or commercial partners to apply these techniques turning research laboratory scripts into a generally usable tool set.

6 Conclusions

Testing consumes a significant portion of the budget for software development projects. Formal methods, typically used in the specification and analysis phases of software development, offer an opportunity to reduce the cost of the testing phase. We pursued this path by applying model checking to the problem of test case generation.

We defined a mutation model for model checkers, and used the model checker to automatically generate mutation adequate tests that distinguish these mutants from the original model. It is noteworthy that the problem of identifying equivalent mutants, which is a significant issue for program-based mutation analysis, is completely handled by the model checker in our approach. Our approach generates expected outputs for each test input, thus taking care of the "oracle" problem. We defined mutation operators to produce two types of tests, namely tests from which a correct implementation must diverge, and tests which a correct implementation must follow.

We showed the feasibility of our approach on the cruise control example. We turned the counterexamples produced by the model checker into tests with the NIST TDA tool and applied the tests to an implementation in Java. Branch and other coverage analysis shows that the tests are quite good, but not perfect. Analysis of the uncovered branches shows that implicit aspects of the specification are difficult to test, thereby suggesting that methods to make specifications explicit are important from the test case generation perspective.

Acknowledgments

We thank Joanne Atlee for her explanatory e-mail and correct electronic SMV specifications. We appreciate Sun Microsystems, Inc. for making evaluation copies of JavaScope available to the public. Thanks also to Jeff Offutt for suggestions on our presentation.

References

[1] J. M. Atlee. Native model-checking of SCR requirements. In *Proceedings of the Fourth International SCR Workshop*, November 1994.

[2] J. M. Atlee, 1998. Personal Communication.

[3] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 280–292, January 1996.

[4] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[5] P. E. Black, K. M. Hall, M. D. Jones, T. N. Larson, and P. 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.

[6] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^20 states and beyond. In *Proceedings of the Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

[7] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN Workshop*, Rutgers, NJ, August 1996. Also WVU Technical Report #NASA-IVV-96-022.

[8] E. M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency - Reflections and Perspectives*. Springer Verlag, 1994. Lecture Notes in Computer Science 803.

[9] 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.

[10] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[11] S. R. Faulk, L. Finneran, J. Kirby Jr., S. Shah, and J. Sutton. Experience applying the CoRE method to the Lockheed C-130J. In *Proceedings of the 9th Annual Conference on Computer Assurance*, pages 3–8, Gaithersburg, MD, June 1994.

[12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[13] C. L. Heitmeyer and J. Mclean. Abstract requirements specifications: A new approach and its application. *IEEE Transactions on Software Engineering*, SE-9(5):580–589, September 1983.

[14] K. L. Heninger. Specifying software requirements for complex systems. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[15] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[16] J. Kirby Jr. Example NRL/SCR software requirements for an automobile cruise control and monitoring system. Technical Report TR-87-07, Wang Institute of Graduate Studies, July 1987.

[17] W. J. Majurski. Issues in software component testing. To appear in *ACM Computing Surveys*, 1998.

[18] S. Owre, J. Rushby, and N. Shankar. Analyzing tabular and state-transition requirements specifications in PVS. Technical Report CSL-95-12, Computer Science Laboratory SRI International, June 1995. Revised April, 1996.

[19] A. J. van Schouwen, D. L. Parnas, and J. Madey. Documentation of requirements for computer systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 198–207, San Diego, CA, January 1993. IEEE Computer Society Press.

[20] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.