Using Model-Checkers for Mutation-Based Test-Case Generation, Coverage Analysis and Specification Analysis

Gordon Fraser and Franz Wotawa
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2
A-8010 Graz, Austria
{fraser,wotawa}@ist.tugraz.at

Abstract
Automated software testing is an important measure to improve software quality and the efficiency of the software development process. We present a model-checker based approach to automated test-case generation applying mutation to behavioral models and requirements specifications. Unlike previous related approaches, the requirements specification is at the center of this process. A property coverage criterion is used to show that resulting test-cases sufficiently exercise all aspects of the specification. A test-suite derived from the specification can only be as good as the specification itself. We demonstrate that analysis of the test-case generation process reveals important details about the specification, such as vacuity and how much of the model it covers, without requiring additional costly computations.

1. Introduction
Testing is essential in order to achieve sufficiently high software quality. The complexity of both software and the testing process itself yields the desire for automation. One direction to address this issue is model-based testing. There, test-cases are used to determine whether an implementation is a refinement of an abstract model. Many model-checker based test-case generation methods adhere to this idea. On the other hand, if a requirements specification is available, then testing should concentrate on showing that the implementation is correct with regard to the specification.

Centering the testing process around the specification, a resulting test-suite can only be as good as the specification itself. If some behavior is not covered by the specification, then faults within the uncovered parts can escape the test-suite. It is therefore essential to ensure a sufficiently high quality of the specification in addition to the quality of the implementation.

In this paper, we present a model-checker based approach integrating automated test-case generation and specification analysis. This approach is based on mutation of an abstract model and the specification. It is shown that analysis of the mutants that are used for test-case generation can reveal interesting information about the specification:
• How thoroughly is the specification tested?
• How much of the possible behaviors are covered by the specification?
• Does the specification lead to vacuous satisfaction?

This paper is organized as follows: Section 2 introduces test-case generation with model-checkers and outlines our approach. Section 3 describes different approaches to coverage measurement of the resulting test-suites. Then, Section 4 elaborates on insights on the specification completeness and quality that can be gained from an analysis of the mutants. Section 5 lists empirical results. Finally, Section 6 concludes the paper with a discussion of the results.

2. Automated Test-Case Generation with Model-Checkers
Model-checker based approaches to automated testing have been shown to be promising (e.g., [14]). In this section, we recall the principles of test-case generation with model-checkers, and describe our chosen approach.

In general, a model-checker takes as input a state machine based model of a system and a property. It effectively
explores the state space of the model to verify whether the property holds in all states. If a property violation is encountered, a trace (counter-example) is calculated that consists of states leading from the initial state to a state violating the property. Such a trace can be used as a complete test-case, i.e., it consists of both the data to send to an implementation under test (IUT), and the expected results.

In order to systematically exploit the trace generation mechanism for test-case generation, the model-checker needs to be challenged in appropriate ways. There are two main principles to accomplish this. One is based on the use of coverage criteria not for measuring coverage but for test-case generation. A coverage criterion can be used in order to automatically create a test-suite by creating a special test-case generation. Such a trace can be used as a complete test-case, i.e., it consists of both the data to send to an implementation under test (IUT) and the expected results.

Alternatively, traces can be generated using mutation. A mutation describes a syntactic change that may introduce inconsistency, and can be applied to the model or specification [3, 4, 23], or properties extracted from the model [7] in order to create test-cases. Unlike for program mutants, equality of model mutants can be decided efficiently.

Our prototype implementation has shown that the performance of this approach is acceptable, if the model is verifiable in realistic time. Due to recent abstraction techniques we think that this is a valid assumption. The quality of the resulting test-cases is good, although it clearly depends on the quality of the requirements specification. A test-suite covers all possible faults that can lead to a property violation through negative test-cases. In addition, it is ensured that all parts of the specification are tested through positive test-cases. However, in order to judge about the effective test-suite quality, it is necessary to analyze how much of the model it covers and thus how good the specification is.

3. Coverage Analysis

In general, one of the motivations for automated software test selection is that formal verification or exhaustive testing are not feasible. Due to the finite selection of representative test cases, testing cannot guarantee the absence of errors. Therefore, it is of interest to assess the quality and completeness of a test-suite. In general, coverage criteria are defined to measure quality. Code coverage criteria determine how much of certain aspects of the source code (e.g., statements, branches, etc.) are covered, and are measured during test-case execution. Coverage can also be measured with respect to abstract models or specifications.

3.1. Model Coverage

As proposed by Ammann et al. [1], trap properties, as described in Section 2 in the context of test-case generation, can also be used to measure the coverage of a test-suite with the aid of a model-checker. The test-cases are converted to models suitable as inputs for a model-checker by adding a special variable State which is successively incremented. For each test case one corresponding model is created. The values of all other variables are determined only by the value of State. Due to space limitations we have to refer to Ammann et al. [1] for a more detailed description of this process.
When checking such a model against a trap property, the item represented by the property is covered if a counterexample is returned. As the state space of the test-case model is small compared to the original model, this is an efficient process. The overall coverage of a test-suite is evaluated by checking all test-cases converted to models against the trap-properties derived from the coverage criterion.

### 3.2. Mutation Score

The number of mutants a test-suite can distinguish from the original is given as the mutation score. For mutants of properties (reflected transitions, property mutants), the method presented in the previous section can also be used in order to determine the mutation score. It is simply the number of mutant properties identified (killed), i.e., resulting in a counter-example when checked against a test-case model, divided by the number of mutants in total.

The interpretation of checking a negative test-case model against a mutant property is not clear as it is influenced by behaviors the real model does not exhibit. Furthermore, when creating test-cases by checking against the requirements specification, mutation is applied to the original model and not reflected properties. In this case, test-cases need to be symbolically executed.

A test-case model is combined with the system model against which the test-case should be executed (e.g., a mutant model) by defining the mutant model as a sub-module of the test-case, which can easily be done automatically. In the sub-module, input variables are changed to parameters. The sub-module is instantiated in the test-case model, and by using the input variables as parameters it is ensured that the mutant model uses the inputs provided by the test case.

Execution is simulated by adding a property for each output variable that requires the outputs of the mutant model and the test-case to be equal. This equality is only required for the duration of the test-case as the tested model might still change values beyond the end of the test-case. When challenged with this combination, a model-checker returns a trace that shows how the mutant’s behavior differs from the test-case using the input values of the test-case. If the mutant is not detected, then the model-checker claims consistency. The mutation score equals the number of mutants identified divided by the number of mutants in total.

### 3.3. Property Coverage

It is also interesting to measure the extent to which a test-suite covers the specification. A property coverage criterion for temporal logic formulas was introduced by Tan et al. [21]. In their definition, a test case that covers a property should not pass on any model that does not satisfy this property. This coverage criterion is based on ideas of vacuity analysis [5]. Informally, a property is satisfied vacuously by a model, if part of the formula can be replaced without having an effect on the satisfaction.

Tan et al. [21] suggest a method to generate test-suites fulfilling this coverage criterion using special property mutants. This method is subsumed by our property-mutation approach by including an appropriate mutation operator that replaces atomic propositions with true or false.

We now introduce a method to measure property coverage. The test-cases are converted to models as described in Section 3.1. In order to determine the property-coverage, trap properties are required for each atomic proposition of every property. These trap properties are generated by replacing an occurrence of a sub-formula with true or false, depending on its polarity [20]. The property is altered so that it is checked only within the length of the test-case, and true is assumed else. This can simply be done by extending the property with an implication on the state number, which is an explicit variable in a test-case model.

Property coverage of a test-suite can be determined by converting test-cases to models as was described in Section 3.1, and then checking these against trap properties for all sub-formulas of a property. A sub-formula is covered if the corresponding trap property results in a counterexample. Negative test-cases need to be converted to positive test-case first, as they per definition violate the specification. This conversion is easily done by symbolic execution against the correct model similarly to the method in Section 3.2. A positive test-case is created by extracting the outputs of the correct model during the execution.

An important observation is that a test-suite can only cover those sub-formulas that are not vacuously satisfied. If a sub-formula is satisfied vacuously there exists no execution path usable as a test-case for it. Clearly, vacuity is a problem within the specification. The following section therefore takes a closer look at this problem.

### 4. Specification Analysis

Test-cases generated from only an abstract model test whether an implementation conforms to the model. Including the requirements specification in the test-case generation process allows to create test-cases that determine whether the correct behavior has been implemented. However, the quality of a test-suite created this way depends on the completeness of the specification. Test-cases are only created for those parts that are covered by the specification. Errors in the implementation that are outside the scope of the specification might go undetected. It is therefore of interest to show which behaviors are covered by the specification. In addition, it is useful to identify parts of the specification that have no effect on an implementation (i.e., vacuous satisfaction).
4.1. Specification Vacuity

Consider the following example CTL property: $\text{AG}(c_1 \rightarrow \text{AF} \ c_2)$. This property states that in all paths, at all times, if $c_1$ occurs, $c_2$ must eventually be true. If a model satisfies this property, this can be either because $c_2$ really always is true in some future state after $c_1$, or because $c_1$ never occurs. If $c_1$ is never true, then the property is vacuously satisfied by the model. A vacuous pass of a property is an indication of a problem in either the model or the property. While this simple example could easily be rewritten to be only valid if $c_1$ really occurs this is not as easy in the general case. The detection of vacuous passes has recently been considered by several researchers [5, 20]. Vacuity can be detected with the help of witness formulas, where occurrences of sub-formulas are replaced with “true” or “false”, depending on their polarity, similarly to Section 3.3.

A sub-formula is satisfied vacuously iff the formula and its witness formula are both satisfied by a model. Witness formulas can be created with property mutation. Therefore, specification vacuity can be detected by analyzing the test-case generation from relevant mutants.

Property coverage and specification vacuity are complementary. Our method creates test-cases for all properties that are not vacuously satisfied. The total number of atomic propositions comprises of the number of covered propositions and the number of vacuously satisfied propositions.

4.2. Specification Coverage

A state of a model or implementation is covered by a test-case, if it is passed during execution. Similarly, it is of interest to know how much of a model is covered by its specification. However, when verifying a specification a model-checker visits all reachable states. Hence, a different notion of “covered” is necessary. In general, a part of a model is considered to be covered by a specification if it contributes to the result of the verification. Two different approaches to specification coverage based on FSMs were originally introduced: Katz et al. [18] describe a method that compares the FSM and a reduced tableau for the specification. In contrast, Hoskote et al. [15] apply mutations to the FSM and analyze the effect these changes have on the satisfaction of the specification. This idea is extended by Chockler et al. [10] and Jayakumar et al. [17]. These approaches are based on mutations that invert the values of observed signals in states of an FSM. A different approach that modifies the model-checker in order to determine coverage during verification is presented in [25].

The general idea of mutation based coverage measurement approaches is to apply mutations (invert signals) to the FSM and observe the effects. If a signal inversion has no effect on the outcome of the verification, then this signal is not covered in the state where it is inverted. The number of signals and states of the FSM can be very large. Therefore, specialized algorithms to determine the coverage are required. In contrast, we define a coverage criterion for the SMV description of a model. This allows the determination of the coverage of the specification through an analysis of the results of the test-case generation process, without additional computational costs.

In order to determine the model coverage on top of the test-generation without further costly computations, we choose to measure conditions, i.e., atomic propositions. A condition is covered by a property, if negation of the condition leads to a different outcome of the verification. Negation of conditions is a typical mutation operation, and is therefore already included in the test-case generation. The coverage of a property is determined by looking at the results of model-checking each of the model mutants resulting from application of this operator against the property. If a mutant yields a counter-example, then the condition negated in this mutant is covered. The overall coverage of a specification is simply the union of the coverage of all properties contained in the specification.

Compared to FSM-based criteria, the interpretation of condition coverage is more intuitive. Results of the coverage analysis can be directly presented to the developer, who can easily see which part of the model needs more attention.

5. Empirical Results

The presented test-case generation approach has been implemented in a prototype. This section presents results of the described test-case generation and analyses.

<table>
<thead>
<tr>
<th>Example</th>
<th>Size</th>
<th>Mutants Valid/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BDD Vars</td>
<td>BDD Nodes</td>
</tr>
<tr>
<td>Example1</td>
<td>9</td>
<td>102</td>
</tr>
<tr>
<td>Example2</td>
<td>19</td>
<td>714</td>
</tr>
<tr>
<td>Example3</td>
<td>29</td>
<td>3337</td>
</tr>
<tr>
<td>Example4</td>
<td>27</td>
<td>3345</td>
</tr>
<tr>
<td>Example5</td>
<td>53</td>
<td>12969</td>
</tr>
<tr>
<td>SIS</td>
<td>29</td>
<td>7308</td>
</tr>
<tr>
<td>Cruise C.</td>
<td>31</td>
<td>3330</td>
</tr>
<tr>
<td>TCAS</td>
<td>57</td>
<td>4006</td>
</tr>
<tr>
<td>Wiper</td>
<td>91</td>
<td>15.3085</td>
</tr>
</tbody>
</table>

Table 1. Example Models and Mutation Results

Our prototype is based on the input language of the model-checker NuSMV [11] for the model description, and Computation Tree Logic (CTL) for example properties.
Mutation operators for both are essentially identical. The following mutation operators were used for models and specification (see [8] for details): StuckAt (replace atomic propositions with true/false), ExpressionNegation (negate atomic propositions), Remove (remove atomic propositions), LogicalOperatorReplacement, RelationalOperatorReplacement, ArithmeticOperatorReplacement, Number-Mutation (replaces numbers with border cases). All the generations, modifications and instrumentations described in this paper are performed automatically by the prototype. All it needs as inputs are a model and a specification.

Table 1 lists statistics (BDD encoding, number of property) of the example models. Example 1–5 are simple models of a car control at varying levels of complexity. The Safety Injection System (SIS) example was introduced in [6] and has since been used frequently for studying automated test case generation. The Cruise Control model is based on [19] and has also been used several times for automatic testing. TCAS is an aircraft collision avoidance system originating from Siemens Corporate Research [16], later modified by Rothermel and Harrold [26]. Finally, Wiper is the software of an actual windscreen wiper controller provided by Magna Steyr. Due to space limitations the examples cannot be presented in more detail here.

Table 1 also contains details about the number of mutants created from each model and its specification. A mutant is considered valid if it is syntactically correct and inconsistent, such that a test-case can be extracted from it. Table 2 shows the results of the test-case generation. The number of unique test-cases is derived by removing duplicates or subsumed test-cases. Table 3 lists the results of the analysis. DC (decision coverage) is branch coverage determined during execution of the test-cases on implementations, where available, and was measured with the Bullseye Coverage Tool\(^1\). Transition coverage is measured on the model using the method described in Section 3.1, with a trap property for each transition. Property coverage is determined by analysis of the result of the property mutation, as are specification coverage and vacuity. Mutation score is not listed, as each inconsistent mutant results in one or many test-cases, and therefore all inconsistent mutants are detected with this approach. As can be seen, the coverage of resulting test-suites is related to the specification vacuity. For example, Cruise Control has high vacuity and therefore poor test coverage.

### Table 2. Test-cases

<table>
<thead>
<tr>
<th>Example</th>
<th>Test-Cases Unique/Total/Avg. Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>Example 1</td>
<td>33/215</td>
</tr>
<tr>
<td>Example 2</td>
<td>16/53</td>
</tr>
<tr>
<td>Example 3</td>
<td>69/224</td>
</tr>
<tr>
<td>Example 4</td>
<td>558/111</td>
</tr>
<tr>
<td>Example 5</td>
<td>1173/2012</td>
</tr>
<tr>
<td>SIS</td>
<td>281/1104</td>
</tr>
<tr>
<td>Cruise C.</td>
<td>186/1674</td>
</tr>
<tr>
<td>TCAS</td>
<td>438/104</td>
</tr>
<tr>
<td>Wiper</td>
<td>1416/5623</td>
</tr>
</tbody>
</table>

### Table 3. Comparison of Code- and Model-Coverage of Test-Cases and Specification

<table>
<thead>
<tr>
<th>Example</th>
<th>Test Coverage</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DC</td>
<td>Trans.</td>
</tr>
<tr>
<td>Example 1</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Example 2</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Example 3</td>
<td>80%</td>
<td>100%</td>
</tr>
<tr>
<td>Example 4</td>
<td>93%</td>
<td>91%</td>
</tr>
<tr>
<td>Example 5</td>
<td>92%</td>
<td>99%</td>
</tr>
<tr>
<td>SIS</td>
<td>-</td>
<td>97%</td>
</tr>
<tr>
<td>Cruise C.</td>
<td>72%</td>
<td>70%</td>
</tr>
<tr>
<td>TCAS</td>
<td>87%</td>
<td>100%</td>
</tr>
<tr>
<td>Wiper</td>
<td>92%</td>
<td>100%</td>
</tr>
</tbody>
</table>

6. Conclusion

In this paper, we have presented an integrated approach to automated test-case generation and analysis focusing on the requirements specification. Model mutants are used to create test-cases that check for possible errors in the implementation that violate the specification. Specification mutants lead to test-cases that explore all aspects of the specification properties. It is sometimes argued that a sufficient formal requirements specification might not be available. However, there are many conceivable scenarios where detailed specifications are necessary. E.g., safety related software for automotive or avionic systems is subject to rigorous standards that enforce detailed specifications.

In addition to the test-case generation, we have described several possible analyses of resulting test-suites. We have extended previous approaches to apply to our method, and have shown that analysis of which mutants resulted in test-cases can be used to gain additional insights, i.e., property coverage of test-cases, model-coverage of the specification and specification vacuity. This makes it possible to judge about the quality and completeness of the specification.

Even though experiments indicate feasibility of the presented approach, performance problems are likely for bigger models. While all mutations and transformations described in this paper can be done efficiently, the performance bottle neck clearly is the model-checker. Previ-

\(^1\)http://www.bullseye.com
ous case studies (e.g., [14, 2]) show that model-checker based testing scales well even for real-world examples. However, commonly models without numeric variables are used for such experiments. Therefore, as agreed by others researchers[22], abstraction might be necessary.

References


