Property Relevant Software Testing with Model-Checkers

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

Verification is applied to software as a proof method with respect to its requirements. Software testing is necessary due to the fact that verification is often infeasible. Automation is desirable since the complexity and the effort involved are significant. However, automated software testing is commonly used to ensure confidence in the conformance of an implementation to an abstract model, not to its requirement properties. In this paper, we introduce the notion of property relevance of test-cases. Property relevant test-cases can be used to determine property violations. It is shown how to detect the properties relevant to a test-case. New coverage criteria based on property relevance are introduced. Automated generation of test-suites satisfying these criteria is also presented. Finally, feasibility is illustrated with an empirical evaluation.

Categories and Subject Descriptors

D.2.5 [Testing and Debugging]: Testing tools; D.2.5 [Testing and Debugging]: Tracing; D.2.1 [Requirements/Specifications]: Tools

Keywords

Property relevance, software testing, requirements traceability, test-suite analysis, model-checker based testing

1. INTRODUCTION

Testing is an essential part of the software development process. Due to its complexity, automation is highly desirable. A common solution is to derive test-cases from a model of the software, and thereby seek indication whether an implementation conforms to the model or not.

There is a subtle difference between test-cases for model and specification conformance. In order to detect as many faults as possible, model conformance testing aims to maximize confidence in that the implementation is a correct refinement of the model. On the other hand, safety related software, for example, requires that safety related requirements are shown to be fulfilled with the help of test cases. Testing for requirements conformance aims to maximize coverage of the requirement properties. However, no matter how high the achieved coverage is, testing can only increase confidence but not ascertain satisfaction, unless it is done exhaustively. In contrast, a single trace can be sufficient to show a property violation. If a property is violated by a model, then a model-checker returns a counter-example that illustrates the inconsistency. This idea could also be applied to testing. Ideally, failed test-cases should not only indicate non-conformance to a model, but suggest which properties are violated.

In this paper, we use this idea and formally link test-cases with requirement properties. The notion of property relevance is introduced. Under certain conditions, property violation by an implementation can be inferred from a failed relevant test-case. Property relevance enables traceability of requirements to test-cases. Our experience has shown that even when traceability is required, the success of a test-suite is often still measured with structural coverage criteria, because this is supported by many tools. We therefore present new coverage criteria that combine structural coverage and property relevance. We show how these criteria can be measured on existing test-suites, and present a method to automatically create test-suites that satisfy them. Recently, model-checker based testing has been considered in many research projects. The high degree of automation achievable with such approaches is attractive. Furthermore, the ability of model-checkers to detect property violations is useful in the context of the goals this work tries to achieve. Therefore, the introduced methods make use of a model-checker. We illustrate the feasibility of property relevance and coverage by providing empirical results for experiments conducted on a set of examples.

This paper is organized as follows: Section 2 introduces property relevance formally. Coverage criteria based on property relevance and methods to measure them are presented in Section 3. Then, Section 4 describes automated generation of test-cases that satisfy the presented coverage criteria. Section 5 lists empirical results for property relevance and property related analysis techniques applied to a set of example models. Finally, Section 6 concludes the paper with a discussion of the results.

2. PROPERTY RELEVANCE

In general, software testing can only ascertain that a property is fulfilled by an implementation under test (IUT) if the test-suite is exhaustive. However, exhaustive testing is commonly not feasible due to the potentially huge number of possible test-cases. Therefore, testing has to aim at revealing as many property violations...
as possible. Usually, a test-case that locates a fault is used as an indication that the IUT is erroneous without specifying which requirement properties are violated by this fault. We now define property relevance which is a relation between a test-case and a property. With the help of property relevance it is possible to determine whether an implementation that fails a test-case also violates a property for which the test-case is relevant. This information is for example useful with regard to traceability, and helps estimating costs and severities of faults. Furthermore, the coverage and test-case generation techniques presented in later sections are based on property relevance.

2.1 Preliminaries

In model-based testing, test-cases are created for an IUT with respect to a model $M$ and a specification consisting of a number of requirement properties $P_i$. As we are going to use a model-checker for both test-case generation and analysis, we define the model as a Kripke structure:

**DEFINITION 1. Kripke Structure:** A Kripke structure $M$ is a tuple $(S, s_0, T, L)$, where $S$ is the set of states, $s_0 \in S$ is the initial state, $T \subseteq S \times S$ is the transition relation, and $L : S \rightarrow 2^{AP}$ is the labeling function that maps each state to a set of atomic propositions that hold in this state. $AP$ is the countable set of atomic propositions.

An execution sequence of this model is a path (also referred to as trace). We denote the set $P$ of possible execution sequences of a model $M$ as $P = \text{traces}(M)$.

**DEFINITION 2. Path:** A path $p := s_0, s_1, ... >$ of model $M$ is a finite or infinite sequence such that $\forall i > 0 : (s_i, s_{i+1}) \in T$ for $M$.

Properties constrain the allowed paths of a system. Temporal logics are used to express these properties. A temporal logic can either be based on linear time or branching time. The semantics of a linear time formula is given as a sequence, whereas the semantics of a branching time formula is given as a tree. We consider the linear time logic LTL [25], although the ideas in this paper can also be applied to computation tree logic, e.g., CTL [12].

An LTL formula consists of atomic propositions, Boolean operators, and temporal operators. We only consider the temporal operators $U$ and $X$, as all other operators can be expressed with these operators. $X$ refers to the next state. E.g., $X a$ expresses that $a$ has to be true in the next state. $U$ is the until operator, where $a U b$ means that $a$ has to hold from the current state up to a state where $b$ is true. The syntax of LTL is given as follows, with $a \in AP$:

$$\phi ::= a \mid \neg \phi \mid \phi_1 \land \phi_2 \mid X \phi \mid \phi_1 U \phi_2$$ (1)

A property $P$ satisfied by model $M$ or path $p$ is denoted as $M \models P$ or $p \models P$, respectively. A model $M$ violating property $P$ is denoted as $M \not\models P$. The validity of an LTL formula $\phi$ on a path $p$ is defined as follows:

$$p \models a \iff a \in p(0)$$ (2)

$$p \models \neg \phi \iff p \not\models \phi$$ (3)

$$p \models \phi_1 \land \phi_2 \iff p \models \phi_1 \land p \models \phi_2$$ (4)

$$p \models X \phi \iff p_1 \models \phi$$ (5)

$$p \models \phi_1 U \phi_2 \iff \exists t \in \mathbb{N} : p_t \models \phi_2 \land \forall 0 \leq j < t : p_j \not\models \phi_1$$ (6)

Three verifiable types of properties are distinguished:

**Safety Property:** A safety property describes a behavior that may not occur on any path ("Something bad may not happen"). To verify a safety property, all execution paths have to be checked exhaustively.

**Invariance Property:** An invariance property describes a behavior that is required to hold on all execution paths. It is logically complementary to a safety property.

**Liveness Property:** A liveness property describes that "something good eventually happens". With linear time logics, this means that a certain state will always be reached.

Property violation can either be shown by a finite path or with a special kind of infinite path, a lasso-shaped sequence:

**DEFINITION 3. Lasso-shaped sequence [22]:** A sequence $\beta$ is lasso-shaped if it has the form $(s_1, s_2)^\omega$, where $s_1$ and $s_2$ are finite sequences.

A lasso-shaped sequence can prove that a certain state is not reached, and can therefore detect violation of a liveness property. In order to determine violations of any kind of property, we define test-cases such that they are either linear sequences or lasso-shaped:

**DEFINITION 4. Test-Case:** A test-case $t$ is a finite prefix of a path $p_1$, or a lasso-shaped sequence $p_2$ of a model $M$. A test-suite $T$ is a finite set of test-cases.

Informally, a test-case $t = s_0, s_1, ... s_k$ is executed on an implementation $I$ by providing values for all input-variables in $L(s_i)$ to $I$ and comparing the output-variables in $L(s_i)$ with the values returned by $I$, for every state $s_i$. Any variable or atomic proposition provided by the environment and not calculated by the system is considered as input. It is not necessary for the test-case to distinguish between input- and output-variables, it is sufficient if the test-case execution framework does this. When executing a test-case that is a lasso-shaped sequence, the test-case can be converted to a finite sequence as shown in [22].

The implementation can be interpreted as a formal model $I = (S, s_0, T, L)$, such that:

**DEFINITION 5. Test-case execution:** A test-case $t = s_0, s_1, ... s_k$ passes on implementation $I = (S, s_0, T, L)$, if $I$ has a path $p =< s_0, s_1, ... s_k, ... >$ such that for all states $s_i : 0 < i < k$:

$s_i = t_i$, i.e., $t \in \text{traces}(I)$. If $I$ does not have such a path, the test-case $t$ fails on $I$.

As $t$ is a path, we also denote $t \models P$ if property $P$ is satisfied by the path represented by $t$, and $t \not\models P$ is violated by $t$. Two different types of test-cases can be distinguished:

**DEFINITION 6. Positive Test-Case:** A positive test-case detects a fault if its execution fails. $I \oplus t$ denotes a fault free execution of test-case $t$ on the implementation $I$, i.e., the test-case passes. $I \oplus t$ denotes an execution fault, i.e., the test-case fails.

**DEFINITION 7. Negative Test-Case:** A negative test-case detects a fault if its execution passes. $I \oplus t$ denotes an execution of test-case $t$ on the implementation $I$ with fault, i.e., the test-case fails. $I \oplus t$ denotes a fault free execution, i.e., the test-case passes. Therefore, the test-case $t = < t_0, ..., t_j, t'_j, ... >$ contains at least one transition $(t_i, t'_i) \not\in T$.

Positive and negative test-cases are also referred to as passing and failing tests, respectively.
2.2 Relevance of Negative Test-Cases

A single trace can show a property violation, while only an exhaustive set of traces can prove property satisfaction. As an example, model-checkers verify properties on models and return a trace as counter-example if a property violation is found. Consequently, a single negative test-case can be sufficient to show property violation.

If a path \( t \) violates a property \( P \) (i.e., \( t \not\models P \)) then any model \( M \) that contains this path also violates \( P \), i.e., \( \neg P \land t \in \text{traces}(M) \rightarrow P \not\models M \). We call a test-case that represents such a path relevant to \( P \). If a test-case \( t \) is relevant to property \( P \), then \( \forall M : M \models t \rightarrow M \not\models P \).

However, care has to be taken because a test-case is usually only a finite prefix of an execution path, unless it is a lasso-shaped sequence. If it is only a prefix, then the path can be truncated such that a property is violated because of the truncation. For example, consider a condition on the next state \( (X) \) on the final state of the test-case. In such a case, even if \( t \not\models P \), a model \( M \) can satisfy \( P \) because \( t \) is only a prefix of a path that satisfies \( P \), albeit beyond the end of the path. In order to avoid this, \( P \) needs to be transformed so that it is satisfied if the truncation of the path leads to a property violation.

For this, we assume that a test-case \( t \) contains an additional variable \( \alpha \) that carries the number of the current state:

\[
\forall t_i \in t : (s = i) \in L(t_i).
\]

**Definition 8. Prefix Transformation:** The prefix transformation \( P' = \alpha(P, t) \) for an LTL property \( P \) with respect to test-case \( t \) of length \( l \) and with state counter \( s \) is recursively defined as:

\[
\alpha(a) = a
\]

\[
\alpha(\neg \phi) = \neg \alpha(\phi)
\]

\[
\alpha(\phi_1 \land \phi_2) = \alpha(\phi_1) \land \alpha(\phi_2)
\]

\[
\alpha(\phi) = X((s < l) \rightarrow \alpha(\phi))
\]

\[
\alpha(\phi_U \phi_2) = \alpha(\phi_1) U ((s < l) \rightarrow \alpha(\phi_2))
\]

\( \phi \) denotes a temporal logic formula, and \( \alpha \in AP \) denotes a propositional formula.

Now we can define a negative test-case \( t \) to be relevant to a property \( P \) if \( P' = \alpha(P, t) \) is violated by \( t \):  

**Definition 9. Relevant Negative Test-Case:** A negative test-case \( t \) is relevant to property \( P \) if \( t \not\models P' \), where \( P' = \alpha(P, t) \).

For example, in a system with input \( x \) and output \( y \), the negative test-case \( t = \{ (x = 1, y = 0) \} \) is relevant to a property \( P := G(x = 1 \rightarrow X y = 1) \), as \( t \) and \( P \) are not consistent. \( G \) is a shorthand for \( \neg (true \ U \neg x) \) and requires \( x \) to be true at all times. On the other hand, \( t \) is not relevant to \( G(y \not\rightarrow X y = 0) \).

**Theorem 2.1.** \( I \models \neg t \models I \not\models P' \): If an implementation \( I \) passes a negative test-case \( t \) relevant to property \( P \), then \( I \) does not satisfy \( P \). Theorem 2.1 states that if the behavior of an implementation \( I \) does not deviate from a negative test-case \( t \), it is known that \( I \) violates \( P \), if \( t \) is relevant to property \( P \).

2.3 Relevance of Positive Test-Cases

As shown, a negative test-case detects a property violation if it is passed by an IUT. However, if the IUT fails a negative test-case the result is inconclusive. The failing can be caused by either a correct implementation, or by a fault resulting in different outputs than described by the test-case. Positive test-cases, on the other hand, can detect any fault that results in a deviation of the expected outputs. However, neither property satisfaction nor violation can be directly concluded from positive test-cases in general. This section considers property relevance for positive test-cases.

Following the definition of negative property relevance, we say a positive test-case is relevant if an implementation \( I \) can fail with the inputs provided by the test-case such that a property violation results. As an additional prerequisite, the test-case itself has to be consistent with the property. This is required so that the property violation is not caused by the finite truncation.

In order to define relevance this way, it is necessary to identify the input and output variables of a model \( M \). We define \( I \subseteq L \) as the atomic propositions over input variables, and \( O \subseteq L \) as the atomic propositions over output variables.

**Definition 10. Relevant Positive Test-Case:** A positive test-case \( t = \lt t_0, ..., t_i, t_j, ..., t_k \rangle \) for model \( M = (S, s_0, I, L, I, O) \) is relevant to property \( P \) if \( t \models P \) and there exists a \( t' = \lt t_0, ..., t_i, t_j, ..., t_k \rangle \) for any \( 0 \leq j < k \), such that for any \( t_s \in t \) and \( t'_s \in t' \), \( I(t_s) = I(t'_s) \) and \( t' \not\models P \), where \( P = \alpha(P, t) \), and \( (t_s, t'_s) \not\in T \).

For example, in a system with input \( x \) and output \( y \), the positive test-case \( t = \{ (x = 1, y = 0), (x = 0, y = 1) \} \) is relevant to property \( G(x = 1 \rightarrow X y = 1) \) (the second state can be changed so that the property is violated). \( t \) is not relevant to \( G(x = 0 \rightarrow X x = 0 \rightarrow X y = 1) \) because no change in \( t \) can lead to a violation.

As it is not possible to conclude property violation or satisfaction from a positive test-case in general, it is necessary to analyze the execution trace created during the test-case execution and then decide about property violations. This is of course possible in general, the property relevance only narrows down the set of possibly violated properties. If the trace also violates the property, then it is known that the IUT violates \( P \). The complexity of analysis of a finite execution trace with regard to a property is computationally simple (especially compared to verification of the complete implementation).

**Definition 11. Execution Trace:** The trace \( t' \) of the execution of test-case \( t = \lt t_0, ..., t_i \rangle \) on an IUT conceivable as model \( M = (S, s_0, I, L, I, O) \) is a finite sequence \( s_0, ..., s_1 \) of states. For all \( 0 \leq x \leq l : I(t_x) = I(t'_x) \).

If an implementation \( I \) passes a positive test-case \( t (I \models t) \), then the resulting trace \( t' \). If the trace \( t' \) violates the prefix transformed property \( P' = \alpha(P, t') \), i.e., \( t' \not\models P' \), then \( I \not\models P \).

**Theorem 2.2.** \( I \models t \models t' \models P \land t' \not\models P' \rightarrow I \not\models P \): If an implementation \( I \) fails a positive test-case \( t \) relevant to property \( P \), where \( t \) fulfills \( P \), and the resulting trace \( t' \) does not satisfy \( P' = \alpha(P, t') \), then \( I \) does not satisfy \( P \).

While a negative test-case only describes certain property violations, a positive test-case is relevant to all possible violations that can occur during its execution. Therefore, a test-suite of positive test-cases is likely to be relevant to more properties with increasing size. Intuitively, relevance of positive test-cases is easier to achieve. However, we see the greatest use not in measuring property relevance directly, but in derived coverage criteria, as presented in the next section.
3. PROPERTY RELEVANT COVERAGE ANALYSIS

The quality of a test-suite is commonly evaluated by measuring its coverage with respect to certain coverage criteria. These criteria are mostly based on structural items, e.g., lines of code or branches. Because measurement of structural coverage can be done simply by recording which items are passed during execution, there are many such coverage criteria. However, there are only few that consider requirement properties. For example, Callahan et al. [9] define coverage of a partitioning of the execution tree based on properties. Whalen et al. [27] adapt structural coverage criteria to LTL properties. Ammann et al. [5] define a notion of dangerous traces that is similar to property relevance. Li et al. [22] apply ideas from vacuity analysis to testing.

In this section we introduce new coverage criteria based on property relevance. We also describe how test-suites can be evaluated with regard to property relevance and derived criteria. The evaluation follows a model-checker based approach introduced in previous works. Therefore, the section begins with an overview of model-checker based test-suite analysis.

3.1 Test-Suite Analysis with Model-Checkers

We use the syntax of the model-checker NuSMV [11] for all example models in this paper. Listing 1 shows a very simple model that consists of two Boolean variables, where \( x \) is an input variable and \( y \) is true in all states following a state where \( x \) is true (true and false are represented by 1 and 0, respectively).

The general approach is to convert test-cases to verifiable models [2]. For this, a new variable \( \text{State} \) is introduced and successively increased up to the final state of the test-case. All other variables are defined such that their values are determined only by the value of \( \text{State} \). As an example, assume a simple test-case \( t = \{(x = 1, y = 0), (x = 0, y = 1), (x = 1, y = 1)\} \). This test-case is converted to the model in Listing 2.

With this model, a model-checker can be queried about fulfillment of certain properties. For each structural item that should be covered according to the coverage criterion (e.g., states or transitions), a trap property is formulated. The trap property expresses that the structural item cannot be reached. Therefore, when the model-checker is challenged with a model and the trap property, it returns a counter-example if the test-case does reach the structural item. For example, to check whether the example test-case covers the transition \( \{(x = 0, y = 1), (x = 1, y = 1)\} \), the following trap property specified in LTL can be used: \( G(x = 0 \land y = 1 \rightarrow X \neg \{(x = 1 \land y = 1)\}) \). Consequently, a coverage criterion can be represented as a set of trap properties. A test-case model is checked against such a set of trap properties. Each trap property that leads to a counter-example is considered to be covered. The overall coverage of a test-suite is simply determined as the number of trap properties that lead to counter-examples with some test-case out of a test-suite.

3.2 Measuring Property Relevance

Property relevance of a test-case can be measured using the model representation presented above. To determine the relevance of a negative test-case, the test-case model has to be checked against prefix transformations of the properties in question. Any property that leads to a counter-example when its prefix transformation is checked against a test-case model is relevant to the test-case.

Relevance of positive test-cases is more difficult to measure. It is necessary to simulate the possible deviations from the test-case. A naive approach would be to allow any random output to occur along the test-case execution. However, this would allow a test-case to be relevant to too many properties. Instead, we assume an implementation to be close to being correct, as is stated in the competent programmer hypothesis [1], which has been shown to be reasonable.

This still leaves several possibilities to define how the property violation can be reached. In a related approach, Ammann et al. [5] describe a concept of dangerous traces as such traces where a mutant model can reach a property violating state. Following this idea, it would be required to evaluate each test-case against a complete set of mutant models. Alternatively, we could require a single erroneous transition anywhere along the trace to lead to a state where a property violation occurs. This could be evaluated by a slight modification of the test-case model itself and would therefore be efficiently measurable. However, restricting deviations this way makes it impossible for test-cases to be relevant to properties that are violated by lasso-shaped sequences. As a compromise between these different approaches, we choose to allow one erroneous transition, and then continue the trace with states provided by the correct model.

In order to detect property relevance, the test-case has to be symbolically executed against a version of the model that allows an erroneous transition, which we refer to as mutant model. This is illustrated with the example and test-case models from Section 3.1 in Listing 3.

The mutant model is stated as a sub-module of the test-case model. All input variables in the sub-module are changed to parameters. The sub-module is instantiated in the test-case model with its input variables as parameter. The variable \( \text{mutant} \) has a

---

Listing 1: Example model

```plaintext
MODULE main
VAR
  x: boolean;
  y: boolean;

ASSIGN
  init(y) := 0;
  next(x) := case
    x = 1 : 1;
    1 : 0;
    esac;
```

Listing 2: Test-case as model

```plaintext
MODULE main
VAR
  x: boolean;
  y: boolean;

ASSIGN
  init(y) := 0;
  next(x) := case
    x = 1 : 1;
    1 : 0;
    esac;
```

<table>
<thead>
<tr>
<th>STATE</th>
<th>TRANSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>State = 0: 0;</td>
<td>State = 1: 1;</td>
</tr>
<tr>
<td>State = 2: State+1;</td>
<td></td>
</tr>
</tbody>
</table>
non-deterministic choice between 0 and 1 as long as it equals 0. As soon as \texttt{mutant} = 1 it proceeds to the value 2 which does not change anymore. In the mutant model there is a non-deterministic choice for other variables if \texttt{mutant} = 1, i.e., only at one transition.

To determine whether a test-case \( t \) is relevant to a property \( P, P' \) has to be created such that each output variable is replaced by the output variable of the mutant model (e.g., \( y \) to \texttt{Mutant}.\( y \)). Finally, the model-checker is challenged with the combined model and the query whether \( t \) satisfies \( P \) while the mutant model violates \( P' \). If so, a counter-example is returned, showing that \( t \) is relevant to \( P \).

For example, assume property \( P := G(x = 1 \rightarrow X \ y = 1) \) stating that in every state following a state where \( x \) equals 1, \( y \) also equals 1. Accordingly, \( P' \) is \( G(x = 1 \rightarrow X \ \text{Mutant}.y = 1) \).

With the presented methods, the relevance of any set of test-cases with regard to a set of properties can easily be determined. While property relevance is interesting and helpful if a test-suite detects an error, it is not sufficient to assess the overall quality of a test-suite. We therefore introduce new coverage criteria based on property relevance.

### 3.3 Relevance Based Coverage Criteria

While it is definitely reasonable to require a test-suite to have test-cases relevant to all properties, this by itself is not necessarily an adequate coverage criterion. Depending on the properties, a test-suite could theoretically be relevant to all properties while only covering a small subset of the model. Therefore, we extend structural coverage criteria such that they include property relevance:

**Definition 12.** \( X \) Property Relevance Coverage: For any structural coverage criterion \( X \), a test-suite \( T \) achieves \( X \) Property Relevance Coverage, if for each item \( i \) required by \( X \) and each property \( P \) there is a test-case that covers \( i \) and then continues such that it is relevant to \( P \).

For example, Transition Property Relevance Coverage requires that for each transition and each property there is a test-case that executes the transition and then proceeds relevant to the property. This means that in the worst case for \( i \) transitions and \( j \) properties there are \( i \times j \) test-cases necessary to satisfy Transition Property Relevance Coverage. However, one test-case can cover several transitions and be relevant to several properties. Therefore the number of test-cases is significantly smaller than \( i \times j \) in practice.

The usual method of trap properties cannot be directly applied to this coverage criterion. An according trap property \( T_R \) would need to show that a structural state described by another trap property \( T_S \) is reached. Then, the trace violating \( T_S \) would need to continue such that property \( P \) is violated, which is not expressible with a single property. Therefore, there are two options: One is to make the evaluation a two-step process, the other is to define a weaker variant of the criterion.

In the two-step approach, first each test-case is checked against the structural coverage trap properties, recording the state number necessary to violate the trap. For example, the transition \( \{x = 0, y = 1\}, \{x = 1, y = 1\} \) is described by the trap property \( T_r := G(x = 0 \land y = 1 \rightarrow x \neg \sim(x = 1 \land y = 1)) \). A test-case model for test-case \( t \) can be directly checked against this trap property. If the test-case covers \( T_r \), then the model-checker returns a trace of length \( l_t \) that executes the transition.

The second step involves checking each test-case whether a property violation can occur between \( l_t \) and the final state of the test-case. With \( P' \) as the prefix transformation of \( P \), we thus check:

\[
G(\text{State} > l_t \land \text{mutant} < 2 \rightarrow P')
\]

(13)

The proposition \( \text{mutant} < 2 \) ensures that the erroneous transition is taken after \( l_t \). If \( t \) is a positive test-case, the model is combined with a modified model that can violate \( P \) and \( P' \) is rewritten to use the mutant model’s output-variables as \( P'' \), as already presented in Section 3.2, and the resulting trap property is:

\[
G(\text{State} > l_t \land \text{Mutant}.\text{mutant} < 2 \rightarrow P'')
\]

(14)

As a simpler alternative to this approach, we can define a weakened version of the coverage criterion which is measurable with trap properties:

**Definition 13.** Weak \( X \) Property Relevance Coverage: For any structural coverage criterion \( X \), a test-suite \( T \) achieves Weak \( X \) Property Relevance Coverage, if for each item \( i \) required by \( X \) and each property \( P \) there is a test-case that both covers \( i \), and is relevant to \( P \).

The difference between Property Relevance Coverage and Weak Property Relevance Coverage is that the latter makes no assumptions on the order in which the possible property violation and covering of the structural item have to occur. This simplifies the evaluation significantly as it can be done in one step with trap properties.

The structural coverage criterion \( X \) results in a set of trap properties \( T \), and the specification consists of a set of properties \( P \). For each \( T \in T \) and \( P \in P \), a test-case model should result in a counter-example, if \( T \) is not satisfied, and \( P \) can be violated. For negative test-cases, this can be expressed as:

\[
G(\text{mutant} < 2 \rightarrow P) \lor (\text{mutant} = 0 \rightarrow T)
\]

(15)
This property is only violated if the trap $T$ is reached without a mutated transition, and if $P$ is violated by a mutated transition. Positive test-cases require a combination of the test-case model and a modified model that can reach a property violating state, as described in Section 3.2. In that case the mutant-variable of the mutant sub-model has to be used in the trap property:

$$G((\text{Mutant}.\text{mutant} < 2 \rightarrow P''') \vee (\text{Mutant}.\text{mutant} = 0 \rightarrow T))$$

(16)

4. PROPERTY RELEVANT TEST-CASE GENERATION

The previous section introduced new coverage analysis methods and showed how to evaluate test-suites with regard to these criteria. This section first gives a short overview of automated test-case generation with model-checkers, and then presents methods to automatically generate test-suites that satisfy a given property related criterion.

4.1 Automated Test-Case Generation with Model-Checkers

A model-checker takes as input an FSM-based model of a system and a temporal logic property. The model-checker effectively explores the system’s state space in order to determine satisfaction or violation of the property. If a property violation is detected, then a trace is returned that illustrates how this violation was reached (counter-example). A counter-example consists of the values of all variables for each state from the initial state up to the violating state. Such a trace can be used as a complete test-case, i.e., consists of test-data and expected results (test oracle). The input values contained in the trace can be sent to an implementation under test (IUT), and the resulting outputs can be compared to the outputs described by the trace in order to determine the correctness of the execution. The main task involved in automated test-case generation with model-checkers is how to systematically select counter-examples. In general, a trace can result from inconsistency. Therefore, either a correct model has to be modified, or special properties that are inconsistent with the model have to be posed on the model. Two main approaches are distinguished:

Coverage Based Methods: Coverage criteria are used to measure how thoroughly a system is exercised by a test-suite, e.g., with statement or branch-coverage. Such coverage criteria can also be used for test-case generation, as introduced by Gargantini and Heitmeyer [14]. For each structural item of a model that should be covered according to the criterion (e.g., states or transitions), a trap property is formulated (These are the same trap properties used to evaluate test coverage). The trap property expresses that the structural item cannot be reached. Therefore, when the model-checker is challenged with a model and the trap property it returns a counter-example that shows how to reach (i.e., cover) this item. Several similar approaches have been presented [26, 15, 16, 20]. It has been shown that simple coverage criteria such as state or transition coverage can lead to inefficient test-suites [19, 18]. Therefore, more complex coverage criteria have been proposed. Commonly, coverage criteria usable for such an approach are based on the model, but there have been other attempts to base coverage on requirement properties [9, 10].

Mutation Based Methods: A simple syntactic change in a textual representation is a mutation. Mutation analysis was introduced in order to evaluate the effectiveness of test-suites. It is based on the competent programmer hypothesis [1] and the coupling effect hypothesis [13]. The former states that programmers are likely to write programs that close to being correct, and the latter states that tests that can reveal simple faults are likely to also detect more complex faults. Specification mutation was introduced in [2] in the context of coverage analysis, and the use for test-case generation was initially suggested in [4]. There are several options to utilize mutation for test-case generation:

- A mutated model can be checked against its specification. If the mutant violates a property, then a resulting trace can be used as a test-case that a correct implementation should fail.
- The specification can be mutated and checked against its model. If the resulting mutant property is inconsistent with its model and is falsifiable, then a trace is returned by the model-checker. This trace shows correct behavior, thus a correct implementation is expected to behave as described by the trace.
- The model can be mutated and checked against the original model, resulting traces illustrating the difference in behavior. Ammann and Black [7] proposed to reflect the model transitions as temporal logic properties, apply mutation to these and then check the resulting mutant properties against the original model. Okun et al. [24] suggested to combine original and mutant models in order to illustrate the different behavior on output variables, thus creating test-cases only for observable faults (state machine duplication).

The performance and feasibility of any of these approaches is determined by the model size. Approaches that enhance the model (e.g., state machine duplication) enlarge the model complexity. Approaches such as direct model mutation introduce additional time penalties for model-encoding as opposed to methods that only check one model against a set of properties. If the model size turns out to be problematic for a model-checker, then experience shows that performance is so bad that none of the different approaches at all can be used. Fortunately, case studies (e.g., [17, 3]) indicate that model checker based test-case generation scales reasonably well, even for real-world examples, although the need for abstraction for complex models is conceivable.

4.2 Property Relevant Test-Case Generation

Most approaches to automated test-case generation do not systematically produce property relevant test-cases. Exceptions are model mutation based approaches that create negative test-cases by checking the mutants against the specification [4, 5].

In this section, we present an approach that automatically creates test-suites that satisfy a given property relevant coverage criterion. A test-case for such a criterion consists of a path that leads to the coverable structural item, and then a postfix that is relevant to a property. The presented approach consists of two steps: First, a test-suite that achieves structural coverage is created and then extended in a property relevant way.

The approach is straightforward: First, a test-suite $T_1$ satisfying the coverage criterion $X$ is created by model-checking the trap properties $\mathcal{T}$ defined by the structural coverage criterion $X$ against the model. The resulting test-cases can be optimized by removing duplicates and prefixes. That way, the number of property relevant extensions that have to be calculated is minimized.

In order to create positive test-cases that are property relevant we need a behavior that can violate a property, and at the same time the according behavior of the correct model. Therefore, the model is duplicated such that there is one correct model and one model that can lead to an erroneous state (mutant model). Both
models are provided with the same input values. This is achieved by stating the mutant model as a sub-module of the original model, and changing all input variables in the sub-module to parameters. The sub-module is instantiated in the original model with its input variables as parameter. As an example, the modifications to the simple model introduced in Section 3.1 are illustrated in Listing 4.

\[
\begin{align*}
\text{MODULE} & \quad \text{model}(x) \\
\text{VAR} & \quad y: \text{boolean}; \\
& \quad \text{mutant}: 0..2; \\
\text{ASSIGN} & \quad \text{init}(y):=0; \\
& \quad \text{next}(y):=\begin{cases} \\
& \quad \text{mutant}=1: \{0,1\}; \\
& \quad \text{next}(y):=\begin{cases} \\
& \quad x=1: l; \\
& \quad l: 0; \\
& \quad \text{esac}; \\
& \quad \text{next}(\text{mutant}):=\begin{cases} \\
& \quad \text{mutant}=0: \{0,1\}; \\
& \quad \text{next}(\text{mutant}):=\begin{cases} \\
& \quad l: 2; \\
& \quad \text{esac}; \\
& \quad \text{init}(\text{mutant}):=0; \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{align*}
\]

Listing 4: Combined model and mutant

In a second step, for each \( t \in T_1 \) and for each property \( P \) a new trap property is specified. The final state of \( t \) is represented as a conjunction of value assignments \( t_f := v1 = value_1 \land v2 = value_2 \). The resulting trap property is, with \( P \) rewritten to \( P'' \) using the mutant’s output variables (regard Section 3.2):

\[
G(t_f \land \text{Mutant.mutant} < 2 \rightarrow P'') \quad (17)
\]

The model \( M \) is combined with a mutant sub-module named \( \text{Mutant} \), and checked against the new set of trap properties. The actual values of the mutant sub-module in the trace can be discarded, only the fact that a violating state can be reached and the values of the correct model are relevant to a positive test-case. In addition, if the model can not be rewritten such that the initial state equals \( t_f \), then the prefix leading from the initial state to \( t_f \) is discarded. This results in a set of test-cases \( T_2 \). Finally, the test-cases \( T_2 \) need to be extended by their counterparts in \( T_2 \).

Negative test-cases can be created in a similar way. However, the combination of correct and mutant model is not necessary. Instead, the mutant model can be directly checked against the trap property:

\[
G(t_f \land \text{mutant} < 2 \rightarrow P) \quad (18)
\]

5. EMPIRICAL RESULTS

The techniques presented in this paper were empirically analyzed with a set of example models. This evaluation tries to show whether property relevant test-case generation and analysis are feasible, and how they perform in comparison to other methods.

Table 1 lists statistics about the model encoding sizes and numbers of properties of the examples used in the evaluation. Example 1–5 are 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. Cruise Control is based on [21] and has also been used several times for automated test-case generation, e.g., [4, 5]. Finally, Wiper is the software of an actual windshield wiper controller provided by Magna Steyr. Due to space limitations the examples cannot be presented in more detail here. The model-checker NuSMV [11] was used for our prototype implementation.

<table>
<thead>
<tr>
<th>Example</th>
<th>BDD Size</th>
<th>Properties</th>
<th>Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>Nodes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example1</td>
<td>113</td>
<td>7</td>
<td>98</td>
</tr>
<tr>
<td>Example2</td>
<td>248</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>Example3</td>
<td>332</td>
<td>28</td>
<td>91</td>
</tr>
<tr>
<td>Example4</td>
<td>27</td>
<td>22</td>
<td>319</td>
</tr>
<tr>
<td>Example5</td>
<td>53</td>
<td>29</td>
<td>545</td>
</tr>
<tr>
<td>SIS</td>
<td>29</td>
<td>18</td>
<td>357</td>
</tr>
<tr>
<td>Cruise C.</td>
<td>31</td>
<td>30</td>
<td>748</td>
</tr>
<tr>
<td>Wiper</td>
<td>91</td>
<td>25</td>
<td>2828</td>
</tr>
</tbody>
</table>

Table 1: Models and mutation results

In order to evaluate the quality of the different test-suites, a set of model mutants was created for each model. In addition to some model features, Table 1 also lists the number of inconsistent mutant models that were created for each example model. Such a mutant results from a simple syntactic change of the SMV model. The following mutation operators were used (see [8] for details): StuckAt (replace atomic propositions with true/false), Expression-Negation (negate atomic propositions), Remove (remove atomic propositions), LogicalOperatorReplacement, RelationalOperatorReplacement, ArithmeticOperatorReplacement, NumberMutation (replaces numbers with border cases). All resulting mutants that are inconsistent with the specification are used as a reference. The test-cases are executed symbolically against these mutants by adding the mutants as sub-modules to the test-case models, and replacing input variables in the mutants with the input values provided by the test-case. A property that states that all output variables of the test-case and the mutant should equal for the duration of the test-case simulates execution with a model-checker. If the output values of the mutant differ at some point during the execution, a counter-example is created. If a counter-example is created, the mutant is identified (killed).

<table>
<thead>
<tr>
<th>Example</th>
<th>T</th>
<th>D</th>
<th>TP</th>
<th>TE</th>
<th>DE</th>
<th>TPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example1</td>
<td>3</td>
<td>4</td>
<td>12</td>
<td>16</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>Example2</td>
<td>3</td>
<td>4</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Example3</td>
<td>3</td>
<td>8</td>
<td>13</td>
<td>174</td>
<td>287</td>
<td></td>
</tr>
<tr>
<td>Example4</td>
<td>6</td>
<td>9</td>
<td>18</td>
<td>78</td>
<td>116</td>
<td>230</td>
</tr>
<tr>
<td>Example5</td>
<td>19</td>
<td>48</td>
<td>33</td>
<td>226</td>
<td>425</td>
<td>387</td>
</tr>
<tr>
<td>SIS</td>
<td>8</td>
<td>16</td>
<td>10</td>
<td>67</td>
<td>158</td>
<td>88</td>
</tr>
<tr>
<td>Cruise C.</td>
<td>8</td>
<td>21</td>
<td>23</td>
<td>109</td>
<td>246</td>
<td>284</td>
</tr>
<tr>
<td>Wiper</td>
<td>32</td>
<td>40</td>
<td>84</td>
<td>529</td>
<td>642</td>
<td>1351</td>
</tr>
</tbody>
</table>

Table 2: Numbers of test-cases generated

We considered the structural coverage criteria Transition (T), Decision (D) and Transition-Pair [23] (TP). Table 2 lists the results of the test-case generation. First, a test-suite was created for each coverage criterion. Then, these test-suites were extended (TE, DE, TPE). The numbers given in Table 2 represent optimized test-suites, i.e., duplicates and redundant prefixes were removed. As
can be seen, the number of test-cases increases significantly for the extended test-suites, while staying within realistic bounds. In average, 52% of all test-cases created from structural coverage criteria for the example models are redundant, while only 27% of the test-cases created for property related coverage are redundant. This indicates that the test-cases explore quite diverse behavior, which is good.

To assess the effectiveness at detecting faults, the test-suites were executed against a set of mutants known to violate the specification. The mutant score is calculated as \( s = \frac{k}{m} \), where \( k \) is the number of killed mutants, and \( m \) is the total number of mutants. The results of this analysis are presented in Table 3. For all the example models, Transition Pair coverage results in the best test-suite without property relevance. As expected, the number of killed mutants is significantly higher for property relevant test-suites. Of course it has to be considered that these mutants are automatically generated and might not be representative for real errors.

Finally, we applied several different automated test-case generation methods to the Wiper-example, and evaluated the resulting test-suites with different coverage criteria. Test-suites were generated with the transition and decision coverage criteria, and also property coverage [22], which requires tests to show the affect of each atomic proposition within a property. In addition, test-suites were created by property mutation (checking mutants of the properties against the original model), mutation of reflected transition relations [7] and a state-machine duplication [24] approach. For these mutation based approaches, the same set of mutation operators used to create the mutants for the mutant score analysis were used. Table 4 lists the results of this experiment. In this table, the test-suites created with the method presented in this paper are referred to as extended. The coverage of these test-suites is very good in comparison to other test-suites, which is not surprising considering they are significantly larger. Interestingly, the coverage with regard to the property relevant criteria is comparable for all test-suites that are created without property relevant methods. The property relevant extensions of transition and decision test-suites achieve significantly better results in this regard.

On the downside, property relevant test-case generation and analysis are more complex than generation and analysis for structural coverage. The addition of a mutant model that can take an erroneous transition to the correct model increases the model complexity. Both test-case generation and coverage analysis depend on a combination of trap properties and requirement properties. Therefore, the number of computations necessary is potentially high.

For example, the test-case generation for transition, decision and transition-pair coverage of the biggest example model (Wiper) on a PC with Intel Core Duo T2400 processor and 1GB RAM take 9s, 23s and 362s, respectively. Extending these test suites takes 33m43s, 89m54s and 41m3s, respectively. A significant part of the complexity is caused by the model duplication necessary for creating positive traces. Extending the same three test-suites with negative test-cases does not require model duplication, and therefore only takes 15m41s, 44m7s and 20m34s respectively.

However, the additional computational effort is still within realistic bounds, especially when considering a main application for safety related systems. As with all model-checker based approaches, the applicability generally depends on the model complexity. If a model-checker fails to verify a model in realistic time, then no model-checker based approach can be used to generate test-cases.

<table>
<thead>
<tr>
<th>Test-Suite</th>
<th>Without Relevance</th>
<th>Relevant Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>D</td>
</tr>
<tr>
<td>Transition</td>
<td>100%</td>
<td>73%</td>
</tr>
<tr>
<td>Decision</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Property</td>
<td>93%</td>
<td>91%</td>
</tr>
<tr>
<td>Mutation</td>
<td>100%</td>
<td>99%</td>
</tr>
<tr>
<td>Reflection</td>
<td>67%</td>
<td>84%</td>
</tr>
<tr>
<td>SM Duplication</td>
<td>92%</td>
<td>86%</td>
</tr>
<tr>
<td>Transition Ext.</td>
<td>100%</td>
<td>73%</td>
</tr>
<tr>
<td>Decision Ext.</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

6. CONCLUSION

In this paper we have introduced the notion of property relevance. We have formally linked test-cases and properties via property relevance, and presented a method to evaluate the property relevance of a test-case. Based on property relevance, we have introduced a novel combination of structural coverage and property relevance. In addition to an evaluation method we have presented a method to automatically create test-suites that satisfy these criteria and create test-cases traceable to requirement properties specification.

A related approach was taken by Ammann et al [5], describing the notion of dangerous traces similar to property relevant traces in this paper. However, the test-case generation is not based upon structural coverage but uses different mutants. These mutants are merged with the original model in order to improve the performance. Their method does not allow generation of relevant positive test-cases for all kinds of properties.

The approach in this paper applies to all properties that can be model-checked. The combination of structural coverage and property relevance guarantees thorough testing even if the requirement properties are poor. Of course, the improvement of property relevant test-case generation in contrast to structural coverage test-case generation is related to the quality of the requirements specification. The approach is neither specific to black- or white-box testing. Functional properties that refer only to input- and output-variables lead to black-box tests, while the internal state of the model can also be used if properties refer to it. On the downside, both the test-case generation and the analysis of property relevant coverage are considerably more complex than related coverage approaches that do not use property relevance. The number of created test-cases is also significantly higher. While this is good with regard to fault sensitivity, a large number of test-cases can incur high costs for test-execution.

Our experiments show encouraging results. The number of property violating mutants that can be detected with property relevant coverage test-suites is significantly larger than with test-suites created from purely structural coverage criteria. The question remains
7. REFERENCES


