Positive and Negative Testing with Mutation-Driven Model Checking

Zhenyu Chen\textsuperscript{1} and Axel Hollmann\textsuperscript{2}
\textsuperscript{1}School of Computer Science and Engineering, Southeast University, China
\textsuperscript{2}Department of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Germany
zychen@seu.edu.cn, hollmann@adt.upb.de

Abstract: Mutation-driven test case generation with model checking has been proposed to reduce the costs of specification-based mutation analysis. Most of the existing work focuses on verifying the expected behaviors in the original model, i.e. positive testing. In this paper negative testing is introduced to check the unexpected behaviors. Mutants are divided into three types: increscent, decrescent, and cross ones. Both, positive and negative testing is proposed to guarantee the detection of these mutants. A non-trivial example illustrates and validates our approach.

1 Introduction and Related Work

Mutation analysis is a commonly accepted technique of fault-based testing that considers faults that cause small changes to the system under test [1, 2]. Mutation analysis has been primarily used for code-based testing techniques, but it has been extended to specification-based testing in recent years. In this context, model checking can be used to compare the mutated specification with the original one to automatically generate test cases [3].

Model checking and specification-based mutation analysis is also used to automatically produce test cases to measure test coverage [4]. Model checking can also be used to detect equivalent mutants that result in redundant test cases. Recently, a technique has been presented to generate test cases from Abstract State Machines specifications to detect specific fault classes [5]. In [6], a new method of mutant minimization is proposed for reducing the number of mutants and thereby the size of the test-suites.

Mutants result in different system behavior: Mutants lose some expected behaviors of the original model or mutants contain some unexpected behavior of the original model. As a novelty of our paper, we will show that these two types of behavior need different test cases: Test cases from the original model are generated to test expected behavior, i.e. positive testing. Test cases from a mutant are generated to test unexpected behavior, i.e. negative testing. Additionally, we divide mutants into three types (increscent, decrescent, and cross mutants). Positive and negative testing is proposed to detect these three kinds of mutants. The concept of positive and negative testing has been adopted from [7, 8].

The paper is structured as follows: In section 2 basic definitions for model checking and mutation analysis are given. Section 3 introduces the concept of positive and negative testing with mutation-driven model checking. Furthermore, a non-trivial example illustrates and validates our approach in section 4. The paper is concluded with section 5.
2 Combining Mutation Analysis and Model Checking

Model checking typically depends on a state space, which is generated based on a set of initial states and transition relations. The state space is usually represented by a graph structure. It is common to use Kripke models introduced as follows:

Definition 1 (Kripke Model [9]) A Kripke model (or model) is a 3-tuple $M = (S, R, I)$, where: (1) $S$ is a finite set of reachable states. (2) $R \subseteq S \times S$ is a transition relation, that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in R$. (3) $I$ is a set of initial states.

Roughly speaking, a mutant is a model $M^* = (S^*, R^*, I)$ which is similar to the original one $M = (S, R, I)$. We assume that $I = I^*$. $S \cup S^*$ is considered as the set of states both in $M$ and $M^*$. For simplicity, we assume that $S = S^*$, in which the states are not required to be reachable. Hence the difference of $M$ and $M^*$ is determined by $R$ and $R^*$. If $R = R^*$, then $M = M^*$ and $M^*$ is an equivalent mutant.

Mutants can be classified as three different types: (1) I-mutant: $M^*$ is called an increscent mutant of $M$ if $R \subseteq R^*$, denoted by $M \leq M^*$. (2) D-mutant: $M^*$ is called a decrescent mutant if $R^* \subseteq R$, denoted by $M^* \preceq M$. (3) C-mutant: $M^*$ is called a cross mutant if $M^*$ is neither I-mutant nor D-mutant, denoted by $M \triangleleft M^*$. $M^*$ is an equivalent mutant of $M$, denoted by $M \equiv M^*$ if and only if $M^* \succeq M$ and $M \preceq M^*$. Otherwise, $M^*$ is an inequivalent mutant of $M$, denoted by $M \not\equiv M^*$.

In a Kripke model $M$, a path is an infinite sequence of states $\pi = s_0s_1 \cdots$, such that $(s_i, s_{i+1}) \in R$ for every $i \geq 0$. A prefix $s_0 \cdots s_k$ of path $\pi$ is denoted by $\pi_k$. Let $M^*$ be a mutant of $M$, a test case is a finite prefix $\pi_k$ in $M \cup M^*$. Test cases are generated based on counterexamples produced from a model checker. A counterexample is always interpreted as a trace that illustrates the difference between model $M$ and mutant $M^*$ that contains system behavior different from the original one.

In this paper, we assume that there are no hidden variables in model $M$. That is an inequivalent mutant can be observed by testing. A test case $\pi_k$ could also be considered as a set of transitions, i.e. $\{(s_i, s_{i+1}) | 0 \leq i \leq k - 1\}$. $\pi_k$ is said to kill (or detect) a mutant $M^*$ from $M$ if $\pi_k \subseteq R \cup R^*$ and $\pi_k \not\subseteq R \cap R^*$. Two specific types of killing test cases can be defined: (1) A test case $\pi_k$ is positive if $\pi_k \subseteq R$ and $\pi_k \not\subseteq R^*$. (2) A test case $\pi_k$ is negative if $\pi_k \not\subseteq R^*$ and $\pi_k \subseteq R$.

In addition, it is necessary to state the desired properties which the system must satisfy. In this paper, we use Linear Temporal Logic (LTL), which consists of atomic propositions, Boolean operators and temporal operators [9]. Two temporal operators $X$ and $G$ will be used to specify the desired properties in this paper. $X$ is the next-state operator, e.g., $Xp$ expresses that $p$ has to be true in the next state. $G$ is the always operator, e.g., $Gp$ expresses that $p$ holds at all states of a path.

If $M^*$ is an inequivalent mutant of $M$, then at least one of the following cases applies: (1) $M$ contains behavior that is not in $M^*$, i.e. $R - R^* \neq \emptyset$. (2) $M^*$ contains behavior that is not in $M$, i.e. $R^* - R \neq \emptyset$. Therefore, we can use the LTL formula $G\!(s & Xs')$ to generate the counterexample with the desired transition $(s, s') \in R - R^*$ or $(s, s') \in R^* - R$. A
challenge of testing with mutation-driven model checking is now to extract the desired transitions from the model specification.

## 3 Testing with Mutation-Driven Model Checking

In this paper, we use a popular model checker NuSMV [10] to demonstrate how mutation analysis could be implemented by model checking. Table I shows the NuSMV specification model of a simplified thermostat system [11]. The variables are defined in the VAR section. The transition system is defined in the ASSIGN section. The system’s initial conditions are declared using INIT() statements. The transition relation is specified using a TRANS expression, in which next statements assign the next values of variables. A similar method to specify the transition relation is using case statements. Please note that the conditions of TRANS expressions are ordering insensitive but the case statements are ordering sensitive. The original specification of thermostat system has been simplified. The case statements have been translated into the TRANS expression. This simplified model is not equivalent to the original one, nevertheless we only use the example to illustrate mutation-driven model checking.

### Table 1: NuSMV Model Specification of Simplified Thermostat System

<table>
<thead>
<tr>
<th>MODULE</th>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>INIT(Thermostat=Off &amp; SwitchIsOn)</td>
</tr>
<tr>
<td>TRANS</td>
<td>A_1 &amp; next(B_1)</td>
</tr>
<tr>
<td></td>
<td>deadlock &amp; next(Thermostat)=Thermostat &amp; next(Enuml)=Enuml &amp; next(SwitchIsOn)=SwitchIsOn</td>
</tr>
<tr>
<td>DEFINE</td>
<td>A_1 := Thermostat=Off &amp; !SwitchIsOn;</td>
</tr>
<tr>
<td></td>
<td>B_1 := SwitchIsOn &amp; Enuml=TempOk &amp; Thermostat=Inactive;</td>
</tr>
<tr>
<td></td>
<td>A_2 := Thermostat=Off &amp; !SwitchIsOn;</td>
</tr>
<tr>
<td></td>
<td>B_2 := SwitchIsOn &amp; Enuml=TooCold &amp; Thermostat=Heat;</td>
</tr>
<tr>
<td></td>
<td>A_3 := Thermostat=Inactive &amp; !SwitchIsOn;</td>
</tr>
<tr>
<td></td>
<td>B_3 := SwitchIsOn &amp; Thermostat=AC;</td>
</tr>
<tr>
<td></td>
<td>A_4 := SwitchIsOn &amp; Thermostat=Off;</td>
</tr>
<tr>
<td></td>
<td>B_4 := !SwitchIsOn &amp; Thermostat=AC;</td>
</tr>
<tr>
<td></td>
<td>A_5 := Thermostat=Active &amp; !SwitchIsOn;</td>
</tr>
<tr>
<td></td>
<td>B_5 := Enuml=TooCold &amp; Thermostat=Heat;</td>
</tr>
<tr>
<td></td>
<td>A_6 := Thermostat=Active &amp; !SwitchIsOn;</td>
</tr>
<tr>
<td></td>
<td>B_6 := Enuml=TooHot &amp; Thermostat=AC;</td>
</tr>
<tr>
<td></td>
<td>A_7 := Thermostat=Heat &amp; SwitchIsOn;</td>
</tr>
<tr>
<td></td>
<td>B_7 := !SwitchIsOn &amp; Thermostat=Off;</td>
</tr>
<tr>
<td></td>
<td>A_8 := Thermostat=Heat &amp; !SwitchIsOn;</td>
</tr>
<tr>
<td></td>
<td>B_8 := Enuml=TempOk &amp; Thermostat=Inactive;</td>
</tr>
<tr>
<td></td>
<td>A_9 := Thermostat=AC &amp; SwitchIsOn;</td>
</tr>
<tr>
<td></td>
<td>B_9 := !SwitchIsOn &amp; Thermostat=Off;</td>
</tr>
<tr>
<td></td>
<td>A_10 := Thermostat=AC &amp; !SwitchIsOn;</td>
</tr>
<tr>
<td></td>
<td>deadlock := !(A_1</td>
</tr>
</tbody>
</table>

Each condition of the TRANS statement is formalized as $A_i \& \text{next}(B_i)$. We define that $T_i := A_i \& X(B_i)$, $T := \bigvee_{i=1}^n T_i$ and $T/k := \bigvee_{i=1}^{n-1} T_i \lor \bigvee_{j=k+1}^n T_j$. Each $T_i$ interprets a set of transitions $(s, s')$ such that $s$ satisfies $A_i$ and $s'$ satisfies $B_i$. Formally, we use the following notation: $R_i := \{(s, s') | A_i(s) \& B_i(s')\}$. The transition relation could be
interpreted as $R = \bigcup_{i=1}^{n} R_i$. Let $R/k := \bigcup_{i=1}^{k-1} R_i \cup \bigcup_{i=k+1}^{n} R_i$. Please note that each pair of $T_i$ and $T_j$ is not required to be disjoint. That is, $R_i \cap R_j \neq \emptyset$. Hence, $R/k \neq R - R_k$ unless $R/k \cap R_k = \emptyset$.

As a model checker inputs a model specification, the Kripke model is not mutated directly. Instead, the textual model specification used as input to the model checker is modified [6]. The resulting mutants are checked against a given specification, and counterexamples illustrate specification violations. In this paper, we consider only single mutations in one of the transition conditions, denoted by $T_k^*$. That is $T_k \neq T_k^*$ and $T_i = T_i^*$ for each $i \neq k$.

The mutated transition conditions can be classified as three types: (1) I-transition: $T_k^*$ is called an increscent mutant of $T_k$ if $T_k \Rightarrow T_k^*$, i.e. $T_k$ implies $T_k^*$. (2) D-transition: $T_k^*$ is called a decrescent mutant of $T_k$ if $T_k^* \Rightarrow T_k$. (3) C-transition: $T_k^*$ is called a cross mutant if $T_k^*$ is neither I-mutant nor D-mutant.

Please note that an I-mutated transition will create an I-mutant and a D-mutated transition will create a D-mutant. However, a C-mutated transition also may create an I-mutant or a D-mutant, because mutated behavior may be masked by other transitions. Similarly, an equivalent mutated transition implies an equivalent mutant, but an inequivalent mutated transition does not imply an equivalent mutant.

The set of transitions is defined as $R^*_i := \{(s, s') | A_i^*(s) \& B_i^*(s')\}$, in which either $A_i = A_i^*$ or $B_i = B_i^*$, because of single mutation in $T_i^*$. $R_i^* = R_i$ for each $i \neq k$, because $T_i^* = T_i$. Hence $R^* = \bigcup_{i=1}^{n} R_i^* = R_k^* \cup R/k$. $R - R^* = (R_k - R_k^*) - R/k$, since $R = R_k \cup R/k$. Similarly $R^* - R = (R_k^* - R_k) - R/k = R_k^* - R$. Please note that $R - R^* \neq R_k - R_k^*$ unless $(R_k - R_k^*) \cap R/k \neq \emptyset$. It is similar to $R^* - R \neq R_k^* - R_k$.

**Definition 2 (Characteristic Property)** Let $M^*_k$ be the mutant of $M$ w.r.t. $T_k^*$. Then the positive property is defined as $f_k = G!(T_k \& T_k^* \& !T/k)$ and the negative property is defined as $f_k^* = G!(T_k^* \& !T_k \& !T/k)$.

Let $M_k^*$ be a mutant of $M$ w.r.t. $T_k^*$. The LTL formula $f_k$ is designed to find the expected behaviors of $M$, i.e. transitions in $R - R^*$. The LTL formula $f_k^*$ is designed to find the unexpected behaviors of $M_k^*$, i.e. transitions in $R^* - R$. Therefore $M^*$ is equivalent to $M$ if and only if $M \models f_k$ and $M^* \models f_k^*$, for mutant $M_k^*$ w.r.t. $T_k^*$.

If $T_k^*$ is an I-mutant of $T_k$, i.e. $T_k \Rightarrow T_k^*$, then $T_k, !T_k^* \equiv false$, thus $f_k \equiv true$, $M \models f_k$. Therefore, $M$ is equivalent to $M_k^*$ if and only if $M_k^* \models f_k^*$. Hence, a test case $t$ that detects an I-mutant must be negative. Similarly, if $T_k^*$ is a D-mutant of $T_k$, then $M$ is equivalent to $M_k^*$ if and only if $M \models f_k$. Hence, a test case $t$ that detects a D-mutant must be positive.

## 4 A Non-trivial Example

In this section, we use the thermostat system model to demonstrate and validate our approach of testing with model checking using the notions of I-, D- and C-mutant.

Example 1 shows negative testing for an I-mutant. It is clear that $A_4 \Rightarrow A_4^*$. Then $f_4 \equiv true$ and $M \models f_4$. Hence, it required a negative test case to detect the mutant w.r.t. $A_4^*$. 
Example 1 (Negative testing for an I-mutant) Mutation of $A^∗_4:=(\text{Thermostat}=\text{Inactive})$. $f^∗_4$ must be set to $\text{LTLSPEC } G(\neg A_4 \land X(B_4) \land A^*_4 \land \neg T/4)$. Model checking: $M^*_4 \not\models f^*_4$.

Test case:
- $s_1$: (Thermostat=Off, Enuml=TempOk, SwitchIsOn=0),
- $s_2$: (Thermostat=AC, Enuml=TooHot, SwitchIsOn=1),
- $s_3$: (Thermostat=Inactive, Enuml=TempOk, SwitchIsOn=0),
- $s_4$: (Thermostat=Off).

Example 2 shows positive testing for a D-mutant. It is clear that $A^*_5 \Rightarrow A_5$. Then $f^*_5 \equiv \text{true}$ and $M^*_5 \models f^*_5$. Hence, it required a positive test case to detect the mutant w.r.t. $A^*_5$.

Example 2 (Positive testing for D-mutant) Mutation of $A^*_5:=(\text{Thermostat} = \text{Inactive} \& \neg (\text{Enuml}=\text{TooCold}) \& \text{SwitchIsOn})$. $f^*_5$ must be set to $\text{LTLSPEC } G(\neg A_5 \land X(B_5) \land A^*_5 \land \neg T/5)$. Model checking: $M \models f_5$.

Test case:
- $s_1$: (Thermostat=Off, Enuml=TempOk, SwitchIsOn=0),
- $s_2$: (Thermostat=AC, Enuml=TooHot, SwitchIsOn=1),
- $s_3$: (Thermostat=Inactive, Enuml=TempOk, SwitchIsOn=0),
- $s_4$: (Thermostat=Heat, Enuml=TooCold).

Example 3 shows testing for a C-mutant. We use both negative testing and positive testing for C-mutants. A weaker property $f^*_9$ can produce a test case. However, this is a wrong test case. It passes in the original model $M$, because the transition $(s_2, s_3)$ can be generated by $T_7 = A_7 \land X(B_7)$, in which $A_7 = (\text{Thermostat} = \text{Heat} \& \text{SwitchIsOn})$ and $B_7 = (\neg \text{SwitchIsOn} \& \text{Thermostat} = \text{Off})$. Therefore, the condition $\neg T/i$ is very important to the characteristic property. On the other hand, $A^*_9$ is a C-mutated transition, but it creates a D-mutant, because its incremental mutated behavior is masked by other transitions.

Example 3 (Testing for a C-mutant) Mutation of $A^*_9:=(\text{Thermostat} = \text{Heat} \& \text{SwitchIsOn})$. $f^*_9$ could be set to $\text{LTLSPEC } G(\neg A_9 \land X(B_1) \land A^*_9)$. Model checking: $M \not\models f_9$.

Wrong test case:
- $s_1$: (Thermostat=Off, Enuml=TempOk, SwitchIsOn=0),
- $s_2$: (Thermostat=Heat, Enuml=TooCold, SwitchIsOn=1),
- $s_3$: (Thermostat=Off, Enuml=TempOk, SwitchIsOn=0).

Example 3 (Testing for a C-mutant) Mutation of $A^*_9:=(\text{Thermostat} = \text{Heat} \& \text{SwitchIsOn})$. $f^*_9$ could be set to $\text{LTLSPEC } G(\neg A_9 \land X(B_1) \land A^*_9 \land \neg T/9)$. Model checking: $M^*_9 \models f^*_9$.

Wrong test case:
- $s_1$: (Thermostat=Off, Enuml=TempOk, SwitchIsOn=0),
- $s_2$: (Thermostat=Heat, Enuml=TooCold, SwitchIsOn=1),
- $s_3$: (Thermostat=Off, SwitchIsOn=0).

5 Conclusion and Future Work

We presented a mutation-driven model checking approach based on existing work. As a novelty, positive and negative testing has been taken into account. Mutants have been
divided into three types and it has been presented how to generate test cases for them. A thermostat system was used to exemplify the approach. Future work aims to present our approach in more detail.

References


