A Method of Software Specification Mutation Testing Based on UML State Diagram for Consistency Checking

Lei Mia* Kerong Ben

*aDepartment of Computer Engineering, Naval University of Engineering, Wuhan, 430033, China

Abstract

Specification mutation testing can be used to check the correctness and consistency of the specification and the program. The paper proposed a method of specification mutation testing based on UML state diagram for consistency checking. We define a set of mutation operators based on the mutation location. Each operator is examined whether it would generate unreasonable mutant, in order to reduce the number of mutants. Then, the required condition of generating test case for the mutant is also analyzed. Based on the basis, there is an integration of the operators according to the inclusion relation among them, which can reduce the cost and improve the efficiency of the mutation testing. The experiment shows that our method of specification mutation testing is effective to detect the inconsistency in the specification and the program.

Keywords: Specification Mutation Testing, UML State Diagram, Mutation Operator, Consistency Checking

1. Introduction

Software mutation testing is a fault-based testing technique which can accelerate the exposure of defects and measure the effectiveness of a test set [1]. The principle of mutation testing is to inject faults into the original software and create a set of faulty version called mutants. These mutants are executed against the test cases. If the result of running a mutant is different from the original software for any test
cases, the mutant is killed and the corresponding injected fault is detected. The number of mutants killed by a set of test case indicates the quality of the set. Mutation testing was originally proposed in the program, and it has also been applied in the specification. Specification mutation testing can be used to check the correctness and consistency of the specification and the program [2].

UML state diagram is a widely used method to describe the dynamic behavior of an entity and plays an important role in the software analysis and design [3]. Specification mutation testing based on state diagram is a hot topic. Definition of mutation operator is one of the crucial issues in mutation testing [4]. Fabbi [2] defined a set of mutation operators to test state diagram at first. Li [5] designed eleven operators to simulate the errors in state, transition, input and output. Belli [6] took the state diagram as a directed graph regardless of the event and the guard condition. On this basis, four mutation operators were defined.

There is some limitation in current definition. For example, both of Li’s and Belli’s operators excluded the mutation in transition guard condition. Therefore, the mutation cannot simulate the corresponding faults and reduce the ability of detecting fault. Meanwhile, the specification mutation testing can check the consistency between specification and program. However, there is little study in existed works.

This paper investigates how to apply the specification mutation testing to check the inconsistency in the specification and the program. The rest of the paper is organized as follows. Section 2 gives an overview of UML State Diagram. Section 3 analyzes the mutation operators defined in this paper. Section 4 gives an experiment to evaluate the effect of our method to check consistency. Finally, the paper is concluded in Section 5.

2. UML State Diagram

UML state diagram consists of state and transition, and describes the dynamic behavior of an entity. The initial/final state indicates the start/end of the behavior. Transition indicates how the entity changes from one state to other one. Every transition has three optional elements: event, guard condition and action. Given that the entity is in a state, if the event is accepted, and the guard condition is true, then the corresponding transition is fired, and the action is performed. Meanwhile, the target state of the transition becomes the current state of the entity.

Our method is mainly used in analyzing non-nested state diagram. If there are nested states in the diagram, it can be flattened to be a general state diagram, and then apply our method for analyzing. Therefore, UML state diagram can be expressed as $D = <S, T, E, A, \alpha, \beta, \gamma, \delta, \rho>$, including:

- $S$ is the set of states in $D$, $s_0$ is the initial state, $s_e$ is the final state;
- $T$ is the set of transitions in $D$;
- $E$ is the set of event to fire the transition;
- $A$ is the set of action that occurs when the transition is fired;
- $\alpha$: $T \rightarrow S$, for each transition $t \in T$, $\alpha(t)$ is the source state of $t$;
- $\beta$: $T \rightarrow S$, for each transition $t \in T$, $\beta(t)$ is the target state of $t$;
- $\gamma$: $T \rightarrow E$, for each transition $t \in T$, $\gamma(t)$ is the sequence of events that can fire $t$;
- $\delta$: $T \rightarrow A$, for each transition $t \in T$, $\delta(t)$ is the action performed when $t$ is fired;
- $\rho$: for each transition $t \in T$, $\rho(t)$ is the condition $r$ needed to fire $t$.

Similarly, mutant $M$ is generated by changing the elements of $D$, so it can be expressed as $M = <S', T', E', A', \alpha', \beta', \gamma', \delta', \rho'>$.

3. Specification Mutation Testing for Consistency Checking

According to the principle in [7], five classes of mutation operators are defined in this section.
3.1. Mutation of Transition

Mutant $M$ is generated by adding, deleting or changing a transition in $D$. Corresponding operators is as follows:

1) Addition of Transition (AOT)
   Adding a transition $t'$ to generate mutant $M$. If $\gamma(t')$ and $\rho(t')$ is not satisfied in state $\alpha(t')$, a legitimate transition cannot be generated to check the consistency between the specification and the software program, so the mutation is unreasonable. Otherwise, a test case can be generated to make the software enter state $\alpha(t')$, and satisfy $\gamma(t') \land \rho(t')$. If the program continues to transition, it indicates the existence of implicit transition in the program, which needs to be fixed.

2) Deletion of Transition (DOT)
   Deleting a transition $t_i \in T$ to generate mutant $M$. That means if software is in state $\alpha(t_i)$ and satisfies $\gamma(t_i) \land \rho(t_i)$, it cannot continue the transition. If the test case set meet the transition coverage criteria, this error can be detected, so this operator is not recommended to be executed.

3) Change of Transition (COT)
   Changing the source state or target state of a transition $t_i \in T$ to generate mutant $M$. If the source state $\alpha(t_i)$ is changed, it is equivalent to adding a transition to state $\alpha'(t_i)$. If the target state $\beta(t_i)$ is changed, the corresponding error can be detected by the existed test cases. So, it need not to change $\beta(t_i)$. If $t_i$ is reversed by switching $\alpha(t_i)$ and $\beta(t_i)$, it is equivalent to adding a transition to state $\beta(t_i)$ and deleting a transition to $\alpha(t_i)$. At this time, it is needed to check whether there is a reachable trace from $s_0$ to $\beta(t_i)$. If so, we can use the trace to check the consistency. Otherwise, the mutation operation should be stopped.

3.2. Mutation of State

Mutant $M$ is generated by adding, deleting or changing a state in $D$. Corresponding operators is as follows:

1) Addition of State (AOS)
   Adding a state $s'$ to generate $M$. If $s'$ is the initial state $s_0$, it is equivalent to adding $t'$ to $s_0$. When the program is initialized correctly, the corresponding error can be detected, so this mutation is not recommended. If $s'$ is the final state $s_e$, it needs to add a transition $t'$ which satisfies $\beta(t') = s'$. It is also equivalent to adding $t'$ to $\alpha(t')$. If $s'$ is other states, it needs to add transition $t'$ and $t''$ that satisfies $\alpha(t') = s'$ and $\beta(t'') = s'$. It is also equivalent to adding $t''$ to $\alpha(t'')$. Therefore, the two mutation can be analyzed as same as AOT.

2) Deletion of State (DOS)
   Deleting a state $s_i \in S$ to generate mutant $M$. For state $s_i$, there exist state $s_j$ and transition $t_k$ which satisfy $\alpha(t_k) = s_j$ and $\beta(t_k) = s_i$. The deletion of $s_i$ is equivalent to deleting transition $t_k$, so this operator is not recommended.

3) Change of State (COS)
   Changing state $s_i \in S$ to generate mutant $M$. If the existed test case can cover $s_i$, the corresponding error can be detected, so this operator need not to be executed.

3.3. Mutation of Event

Mutant $M$ is generated by adding, deleting or changing a event in $D$. Corresponding operators is as follows:

1) Addition of Event (AOE)
Adding a event \( e' \) in transition \( t_i \in T \) to generate \( M \). If \( \gamma(t_i) = \gamma(t_i) \wedge e' \), the constraint of firing \( t_i \) is enhanced, so the mutation trace can be executed correctly, and cannot be used to check the consistency. If \( \gamma(t_i) = \gamma(t_i) \lor e' \) is satisfied, the constraint is weakened. We can make program \( P \) enter state \( \alpha(t_i) \) and accept \( e' \), if \( t_i \) is fired, there exists inconsistency between \( P \) and \( D \). If \( e' \) cannot occur or \( e' \) occurs together with \( \gamma(t_i) \) in \( \alpha(t_i) \), the mutation should be stopped.

2) Deletion of Event (DOE)

Deleting a event \( e_d \) in transition \( t_i \in T \) to generate \( M \). If \( \gamma(t_i) = \phi \) or \( \gamma(t_i) = \gamma(t_i) \square e_d \), the constraint of firing \( t_i \) is weakened. If \( e_d \) occurs always, the mutation should be stopped. Otherwise, we can make \( P \) enter state \( \alpha(t_i) \) and accept \( \gamma(t_i) \), if \( t_i \) is fired, \( P \) is inconsistent with \( D \). If \( \gamma(t_i) = \gamma(t_i) \square e_d \), the constraint is enhanced, so it is not recommended to apply the mutation.

3) Change of Event (COE)

Changing event in \( t_i \in T \) to generate \( M \). Firstly, if \( \gamma(t_i) = e_1 \triangle e_2 \), then \( \gamma(t_i) = e_1 \triangle e_2 \). If \( e_1 \) and \( e_2 \) occurs at the same time in state \( \alpha(t_i) \), the mutation should be stopped. Otherwise, make \( P \) enter \( \alpha(t_i) \) and only accept \( e_1 \) (or \( e_2 \)), if \( P \) continues the transition, then \( P \) and \( D \) are inconsistent. Secondly, replace \( \gamma(t_i) \) with \( e' \in \gamma(t_i) \). If \( e' \) cannot occur in \( \alpha(t_i) \), then terminate the mutation. Otherwise, make \( P \) enter \( \alpha(t_i) \) and accept \( e' \) to check the consistency.

3.4. Mutation of Guard Condition

Mutant \( M \) is generated by adding, deleting or changing a guard condition in \( D \). Corresponding operators is as follows:

1) Addition of Condition (AOC)

Adding a condition \( r' \) in transition \( t_i \in T \) to generate \( M \). Similar to AOE, \( r' \) should meet \( \rho(t_i) = \rho(t_i) \square r' \) and \( r' \) can be satisfied in \( \alpha(t_i) \), otherwise the operator should not be executed. Then make \( P \) enter \( \alpha(t_i) \) and only satisfy \( r' \) to detect the inconsistency.

2) Deletion of Condition (DOC)

Deleting a condition \( r_d \) in transition \( t_i \in T \) to generate \( M \). The equation \( \rho(t_i) = \phi \) or \( \rho(t_i) = \rho(t_i) \square r_d \) should be satisfied. If \( r_d \) is always satiable, then stop the mutation. Otherwise, let \( P \) enter \( \alpha(t_i) \) and only satisfy \( \rho(t_i) \), if \( P \) continues the transition, the inconsistency is detected.

3) Change of Condition (COC)

Changing a condition in transition \( t_i \in T \) to generate \( M \). First, if \( \rho(t_i) = r_1 \square r_2 \), then \( \rho(t_i) = r_1 \square r_2 \). If \( r_1 \) and \( r_2 \) are not satisfied in the meantime, make \( P \) enter \( \alpha(t_i) \) and only satisfy \( r_1 \) and \( \gamma(t_i) \) (or \( r_2 \) and \( \gamma(t_i) \)) to check the consistency. Second, replace \( \rho(t_i) \) with \( r' \). If \( t_i \) can be fired in \( \alpha(t_i) \) by satisfying \( r' \), there is inconsistency between \( P \) and \( D \).

3.5. Mutation of Action

By adding, deleting, or changing action in transition \( t_i \in T \) to generate \( M \). If the existed test cases have been covered the mutated transition, the corresponding error can be detected. Therefore, it is not recommended to use this kind of mutation operators.

4. Experiment

In the experiment, we choose a subsystem of the control system which we have tested in practice. There are 32 states and 45 transitions in the state diagram. Firstly, we generate 90 faulty version of program by injecting one error in the original program each time. These errors are injected in transitions, events or guard conditions. There are 30 faulty programs for each type of error. Secondly, we generate
test cases under state coverage and transition coverage criterion based on the corresponding state diagram of the under test system. Then run these test cases in the faulty programs to check whether these errors can be detected. The number of errors detected is shown in Table 1.

<table>
<thead>
<tr>
<th>Number of Transition Error</th>
<th>Number of Event Error</th>
<th>Number of Condition Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Coverage</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>Transition Coverage</td>
<td>23</td>
<td>10</td>
</tr>
</tbody>
</table>

The result shows neither criterion can generate a set of test cases to detect all the injected errors.

Then we applied our mutation operators, including AOT, COT, AOE, DOE, COE, AOC, DOC and COC, to generate mutants of the state diagram and the corresponding test cases, which are executed in the remained faulty programs. There are totally 240 test cases generated. The number of detected errors is recorded and is shown in Table 2.

<table>
<thead>
<tr>
<th>Number of Transition Error</th>
<th>Number of Event Error</th>
<th>Number of Condition Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Coverage</td>
<td>+14</td>
<td>+16</td>
</tr>
<tr>
<td>Transition Coverage</td>
<td>+5</td>
<td>+13</td>
</tr>
</tbody>
</table>

The experiment result shows that our method can detect most of remained errors, and can improve the ability of the test cases to detect errors in program. There are a few of errors which are not detected. The reason is that the test cases are generated based on the basic mutation operators, which cover all transitions in the diagram without considering the whole possible errors. If the application of operator is designed more carefully, we believe that all the injected errors can be detected.

5. Conclusion

This paper proposed a method of specification mutation testing based on UML state diagram for consistency checking. We define a set of mutation operators based on the mutation location, and examine each operator to avoid generating unreasonable mutant, which can reduce the number of mutants. Then, the required condition of generating test case for the mutant is analyzed. The experiment shows that our method is effective to detect the inconsistency in the specification and the program.

In future work, more complex experiments are used to validate the usefulness of our method. On this basis, better application strategy is studied to improve the efficiency of specification mutation testing.

References


