Mutation Testing of “Go-Back” Functions Based on Pushdown Automata

Fevzi Belli, Mutlu Beyazit
Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Paderborn, Germany
{belli, beyazit}@adt.upb.de

Tomohiko Takagi, Zengo Furukawa
Faculty of Engineering
Kagawa University
Takamatsu, Kagawa, Japan
{takagi, zengo}@eng.kagawa-u.ac.jp

Abstract—A go-back (GB) function for canceling recent user or system operations and going back to and resuming of previous state(s) is very often used regardless of the application domain. Therefore, faulty handling of them can cause severe damages in those applications. This paper proposes a mutation-based approach to testing GB functions modeled by pushdown automata. Novel mutation operators, recent coverage criteria, and a new algorithm for test case generation are introduced. A case study validates the approach and discusses its characteristics.

Keywords—go-back function; model-based testing; mutation testing; pushdown automata

I. INTRODUCTION

Go-back (GB) functions enable canceling of recent user actions and/or system operations and subsequently go back to the previous state(s) in a system or process. They are key factors for software quality, especially that of user friendliness. Thanks to GB functions, users can perform operations without much worry about operational errors because they can try to resume and conclude their work. Absence of GB functions can cause loss of hundreds of hours of valuable work that cannot be made up.

Today, several different programs like office programs, browsers or e-commerce applications make use of GB functions. Hence, GB functions are of significant importance for any kind of application, including proactive, reactive or interactive operation.

This paper introduces an approach to the systematic testing of a system under consideration (SUC). As realization of GB functions requires keeping track of operations (in a stack), they will conveniently be modeled by a pushdown automaton, or a collection of them, which can be organized hierarchically.

The approach introduced suggests using mutation analysis and testing technique for several reasons. First, it enables user-oriented fault modeling. Second, it allows determining the adequacy of test sets no matter how they are generated, that is, mutation analysis is independent of the technique used for test generation.

Mutation testing [1][2] was originally a white-box testing technique based on generating faulty versions (mutants) of software by introducing small but typical faults using mutation operators. It aims at evaluating adequacy of test sets by their ability to reveal these mutants (or detect faults).

Recent works have combined mutation testing with model-based testing [3][4][5], enabling also black-box testing. For this, the original model is mutated and test cases are generated from mutants to be executed on the SUC. Although, as in classical mutation testing, a mutation analysis of a given test set can also be performed in this way, during test execution, one mainly expects to observe failures that are not (only) related to injected faults.

Most of the model-based testing techniques operate on graph-based abstractions and use some coverage criteria for test generation [6]. The basic idea behind coverage entails generation of test cases and selection of an optimal number of them, called a “test suite”, in order to cost-effectively exercise a given set of structural or functional features of the SUC. Good test coverage increases user confidence in software artifacts, showing that the software is doing everything it is supposed to do (positive testing [7]). Some authors propose to cover not only the model, but also its complement, showing that the software is not doing anything it is not supposed to do (negative testing [7][8]). Consequently, they propose specific operators for model manipulation. Negative testing approach is strongly related to model-based mutation testing. Provided that appropriate mutation operators and test generation methods are available, model-based mutation testing can perform both positive and negative testing.

Problem Domain: One of the main problems in model-based testing of GB functions is that GB mechanism can not be fully captured without using memory feature. Therefore, finite state machines (FSMs) [9] or event sequence graphs (ESGs) [8][10] can not be used formally (We exclude UML diagrams [11] from further consideration, as we are primarily interested in a formal and precise handling of GB testing). FSMs represent regular (type-3) languages in Chomsky hierarchy. Thus, in order to capture GB functions, models of stronger representational power are necessary. This leads to pushdown automata (PDA) that relate to context-free (type-
2) languages. Figure 1 demonstrates why the use of FSM is not sufficient for modeling behavior of GB functions.

![Finite State Automaton](image1)

![Pushdown Automaton](image2)

Figure 1. FSA (Type 3 Languages) vs. PDA (Type 2 Languages)

**Solution Domain:** As a solution to the problem of testing GB functions, this paper concentrates on testing GB functions using the PDA model-based mutation testing approach. To do this, we

- propose novel mutation operators for PDA, and
- use recent GB function-based coverage criteria and introduce a new test generation method.

We claim to introduce for the first-time a model-based mutation testing framework for the testing of GB functions of SUC that is modeled by PDA.

The next section summarizes related work, while Section III outlines background information on model-based mutation testing, PDA models and GB functions. Section IV introduces basic mutation and novel stack mutation operators for PDA models, whereas Section V discusses GB coverage criteria for systematic test case generation and introduces a new test generation algorithm. Before Section VII concludes the paper and sketches research work planned, Section VI performs a comprehensive case study to demonstrate the application of the introduced notions in practice, which reveals also their characteristics and limitations.

II. RELATED WORK

Originally, mutation testing was proposed as a method of evaluating test cases by their ability to reveal faulty implementations. Using the classical approach, both computer programs [12],[13],[14] and system specifications [15],[16],[17] are used to generate mutants, and test cases are executed on these mutants for evaluation of adequacy. Alternatively, it is also possible to use the mutated system models to create test cases [3],[4],[5]. These test cases, then, can be executed on the actual implementations to reveal faults. Our primary interest lies in this latter approach, also called model-based mutation testing (see also Section III.A).

Kovács et al. [3] apply mutation testing to SDL. [18] specifications to automate the process of generating tests for telecommunications protocols. Mutation operators for SDL specifications are defined and an algorithm to generate test cases from mutant specifications is proposed.

Another approach [4] focuses on testing of event-based systems like GUIs. Mutation operators are defined for ESG models. Using coverage-based test generation in [19], minimized test sets are generated from mutants and testing of an event-based system is performed.


Belli et al. [24] transfer and extend the work in [4] to regular grammars (RGs). Using a special grammar form, regularity preserving mutations operators, RG-based coverage criteria, and, to avoid high memory requirements of minimization algorithms, a general test generation method is discussed.

Takagi et al. [25],[26] use particular PDA models to perform positive testing of undo/redo functions, and thus utilize models that relate to context-free languages, in contrast to common model-based approaches, which use models related to regular languages. The proposed PDA model is an extended UML state machine diagram that includes stack manipulations for undo/redo functions, and it is different from the formal PDA model used in this paper. The authors define fundamental GB function-based coverage criteria, but do not outline any test case generation methods.

This paper differs from the ones mentioned above because of the following.

- PDA models are introduced.
- Novel mutation operators are defined.
- Model-based mutation testing of GB functions is performed, including both negative and positive testing approaches, using a new coverage-based test generation algorithm.
- The methodology is demonstrated and validated in a case study, while analyzing its characteristics and threats to validity, and performing comparisons.

III. BACKGROUND

This section outlines the notions used in this paper.

A. Model-Based Mutation Testing

Differing from classical mutation testing, model-based mutation testing is a black-box testing approach and aims to test an SUC based on its correct model. The correct model of the implementation is mutated to generate incorrect or other correct (sub) models, that is, mutants. Then, these mutant models are used to generate test cases to check whether or not the implementation conforms to the behavior specified by the mutants. This is achieved in the following steps.

1. A model is created that is supposed to be correct.
2. Mutants are generated from this (correct) model.
3. For each mutant
   - a set of test cases is generated,
   - these test cases are executed on the implementation, and
   - the observed behavior is compared against the behavior outlined by the mutant.
4. The results are interpreted to reveal faults in the implementation, depending on the mutant’s type.
(such as correct or incorrect) and the original correct model.

The following elements are necessary to form a framework for model-based mutation testing:

- A formal model
- Mutation operators defined over the selected model
- Test case generation method(s) defined over the selected model.

B. Pushdown Automata

As justified in the Introduction, this paper suggests PDA for modeling SUC.

Definition 1: A pushdown automaton (PDA) is a tuple \( M = (S, E, G, T, S_0, Z_0, F) \) where

- \( S \) is a finite set of states,
- \( E \) is a finite set of input symbols or events, called the input alphabet,
- \( G \) is a finite set of stack symbols, called the stack alphabet,
- \( T : S \times E \times G \to U (U \subseteq S \times G^* \text{ is finite}) \) is the transition function (\( \epsilon \) is the empty string),
- \( S_0 \in S \) is the start state (or initial state),
- \( Z_0 \in G \) is the start symbol contained initially in the stack, and
- \( F \subseteq S \) is the set of accepting states (or final states).

In a PDA, a transition function \( T \) receives as input a triple \((p, a, X)\), where \( p \) is the current state, \( a \) is the input received in the current state, and \( X \) is the topmost stack symbol. The output of \( T \) is a finite set of pairs \((q, w)\), where \( q \) is the new state and \( w \) is the string of stack symbols which replaces \( X \) at the top of the stack.

A transition can be represented by \( S \)-tuple \((p, a, X, q, w)\). A read operation occurs if \( w = X \), a pop operation is performed if \( w = \epsilon \) and \( Y \) is pushed onto the stack if \( w = YX \). Thus, the stack is used to keep track of some specific information during computation.

Figure 2a demonstrates PDA model of a typical internet shopping system, where the system can be logged on and off by performing events "a" and "b," respectively. Event "c" saves order information and shows order confirmation step, whereas event "d" requests credit card information and then event "f" saves this information by bringing up the confirmation. In addition, event "e" can be used to clear the order and event "g" submits it. Finally, events labeled by "b" are designated as GB functions. For a further explanation of the system, refer to Section VLA.

For PDA in Figure 2a, we have the following:

- \( S = \{0, 1, 2, 3, 4, 5\} \)
- \( E = \{a, b, c, d, e, f, g, h\} \)
- a label "a, X\(w\)" on an arrow from state \( p \) to state \( q \) corresponds to transition \((p, a, X, q, w)\); for example the arrow from "1" to "2" labeled as "a, 0/10" is the transition \((1, a, 0, 2, 10)\)
- \( G = \{0, 1, 2, 3, 4\} \)
- \( S_0 = 0 \)
- \( Z_0 = 0 \)
- \( F = \{5\} \)

In order to demonstrate examples throughout this paper, we make use of this PDA.

Definition 2: A PDA is deterministic (DPDA), if it satisfies the following properties:

- \([T(p, a, X)] = 1 \) for each \( p \in S, a \in E \) and \( X \in G \).
- For each \( p \in S \) and \( X \in G \), if \( T(p, \epsilon, X) \neq \emptyset \) then \( T(p, a, X) = \emptyset \) for every \( a \in E \).

PDA in Figure 2a is deterministic because one can go to only one state from each state using the same stack top symbol and event pair, and there are no \( \epsilon \)-transitions. On the other hand, PDA in Figure 2b is nondeterministic. Although there are no \( \epsilon \)-transitions, from state 2, one can go to state 1 or, again, to state 2 when stack top is 1 and event is \( b \). Thus, \( |T(2, b, I)| = 2 \).

Note that deterministic FSA recognizes regular languages; deterministic PDA recognizes only a strict subset of context-free languages called deterministic context-free languages. Furthermore, equivalence problem is undecidable for non-deterministic PDA, whereas it is decidable for DPDA. This is the reason why we decided to use DPDA for modeling.

The common definition of a PDA does not include outputs, at least not explicitly or formally. The following definitions complete our modeling view.

Definition 3: Given a PDA \( M = (S, E, G, T, S_0, Z_0, F) \), and for some \( p \in S, a \in E \setminus \{\epsilon\}, X \in G \) and \( w \in G^* \),

- An incoming transition \( t \) to state \( q \) in \( M \) is a transition of the form \((p, a, X, q, w)\), that is, \((q, w) \in T(p, a, X)\).
- An outgoing transition \( t \) from state \( q \) in \( M \) is a transition of the form \((q, a, X, p, w)\), that is, \((p, w) \in T(q, a, X)\).
- A loop transition \( t \) in state \( q \) in \( M \) is a transition of the form \((q, a, X, q, w)\), that is, \((q, w) \in T(q, a, X)\).

Consider PDA in Figure 2a: \((1, a, 0, 2, 10)\) is an incoming transition to the state 2 and also an outgoing transition from the state 1, and \((2, e, 1, 2, 21)\) is a looping transition.

C. Go-Back Functions

GB functions are automatic logging mechanisms for improving user friendliness. Current state of software is stored in memory (a pushdown stack) just before execution of user operation. If a user requests to go back to a previous
Perform a transition similar to the ones defined for FSMs [16] or ESGs [4]. In back button on a console, and so on. Selecting the “Undo” item from a menu list, clicking the “Go to” button, or changing the perspective, since it affects runtime behavior of the SUC.

In the following discussion, the phrase “GB function” means the behavior of software based on the above-mentioned common knowledge, and “GB event” means a user operation to go back to a previous state, such as selecting the “Undo” item from a menu list, clicking the “Go back” button on a console, and so on.

As illustrated in Figure 1, the representational power of PDA is indispensable to capture GB functions. One of the key ideas is that PDA stack can be used to store previous states of SUC. To be more precise, a PDA model from the specifications can be produced as follows.

- When SUC receives a user operation that does not correspond to a GB event, it pushes its current state onto the top of a stack, and then deletes the received user operation to determine its next state. In other words, when receiving an event \( a \in E \) that is not a GB event in a state \( p \in S \), a PDA performs a transition \( (p, a, X, q, pX) \), where \( X, q \in S \).

- When SUC receives a user operation that corresponds to a GB event, it pops a stack symbol (a previous state) from the top of the stack, and then goes back to the popped previous state. In other words, when receiving an event \( a \in E \) that is a GB event in a state \( p \in S \), a PDA performs a transition \( (p, a, X, q, q) \), where \( X \in S \).

Once test engineers create PDA model, they can generate test cases from it. However, for systematic testing of GB functions, new test generation methods are required to cover crucial stack operations. The stack is unique from a testing perspective, since it affects runtime behavior of the SUC.

IV. MUTATION OPERATORS

This section discusses basic PDA mutation operators, similar to the ones defined for FSMs [16] or ESGs [4]. In addition, novel stack mutation operators are also introduced.

A. Basic Mutation Operators

The four fault categories introduced in Section III.C can be produced by means of the following operators: Insertion (I), omission (O), and marking (M).

In general, insertion operators are used to generate mutant models that have additional functionality when compared to the original model; that is, the original model is a correct (sub) model of the mutant model. In addition, application of an omission operator almost always yields a mutant model which is a correct (sub) model of the original one. Thus, omission operators are used to leave out some specific parts. Furthermore, marking operators are used to change the type of certain elements in the model.

The transition diagram of a PDA can simply be represented by a multi-directed graph. Nodes of this graph are the states of the PDA, and edges correspond to the transitions. Therefore, based on its multi-directed graph, a PDA can be mutated or transformed into another one using transition mutation, state mutation, and marking operators.

Definition 4: Transition mutation operators. Given a PDA \( M = (S, E, G, T, S0, Z0, F) \):

1. Transition insertion (It) operator adds a new transition \( t \) to \( M \). If \( t = (q, a, X, p, w) \), it is assumed that \( q, p \in S, a \in E \cup \{\epsilon\} \), \( X \in G \), \( w \in G^* \) and \( (p, w) \not\in T(q, a, X) \). Note that application of insertion operator may generate a non-deterministic PDA from a deterministic one.

2. Transition omission (Ot) operator deletes an existing transition \( t \) from \( M \). If \( t = (q, a, X, p, w) \), it is assumed that \( q, p \in S, a \in E \cup \{\epsilon\} \), \( X \in G \), \( w \in G^* \) and \( (p, w) \in T(q, a, X) \). It is possible that, after application of transition operator, some states have no incoming or outgoing transitions.

Definition 5: State mutation operators. Given a PDA \( M = (S, E, G, T, S0, Z0, F) \):

1. State insertion (Is) operator adds a new state \( q \) to \( M \) together with transitions \( t_1, \ldots, t_k \). State \( q \) is not reachable from another state in \( M \) if no incoming non-loop transition to \( q \) is inserted. Furthermore, no state in \( M \) is reachable from state \( q \) if no outgoing non-loop transition from \( q \) is inserted.

2. State omission (Os) operator deletes an existing state \( q \) together with all the transitions ingoing to and outgoing from \( q \). After the deletion, some states in \( M \) may lose all their incoming transitions and become unreachable from the start state. In addition, some states may be left without any outgoing transitions and turn into trap states.
Definition 6: Marking operators. Given a PDA $M = (S, E, G, T, S_0, Z_0, F)$:

- **Mark start (Ms)** operator marks an existing state in $M$ as the start state. The operator also marks the old start state as a non-start state because, by definition, a PDA can only have a single start state.

- **Mark final (Mf)** operator marks an existing state in $M$ as a final state.

- **Mark non-final (Mn)** operator marks an existing final state in $M$ as a non-final state.

- **Mark initial (Mi)** operator marks an existing stack symbol in $M$ as the initial stack symbol. Old initial symbol is marked as non-initial.

Figure 3 shows example mutants generated from PDA in Figure 2a using some of the operators in Definition 4, Definition 5, and Definition 6. In Figure 3a, transition $(4, d, 1, 3, \varepsilon)$ is inserted, whereas, in Figure 3b, transition $(4, b, 3, 3, \varepsilon)$ is omitted. Also, state $d$ is omitted and marked as a final state in Figure 3c and Figure 3d, respectively.

B. Stack Mutation Operators

Note that, using basic mutation operators mentioned above, one can generate all possible PDA (with common $E$ and $G$) by using higher-order operator applications. Therefore, any other mutation operator defined on PDA can be shown to be equivalent to some subsequent applications of these operators. Nevertheless, it is still beneficial to define additional operators, especially for stack operations, in order to avoid the use of higher-order operator applications, which results in a huge number of mutants, and generate some specific type of mutants, depending on the interest.

For this reason, we define additional operators to specifically mutate transitions of PDA. Since, in this paper, our main intention is PDA-based mutation testing, and stack element is an additional feature of a PDA that mainly differentiates it from a FSA, the defined operators focus on performing simple manipulations on stack operation of an existing transition. Simply, these manipulations can be performed either by replacing the value written onto the stack or read from top of the stack during a transition.

Definition 7: Write replacement operators. Given a PDA $M = (S, E, G, T, S_0, Z_0, F)$, write replacement (Rw) operator replaces the string to be put into the stack by the given string $w'$, that is, for $t = (p, a, X, q, w)$, $Rw(t, w') = (p, a, X, q, w'X)$ for some given $w' \in G^*$. This operator can be performed in 4 different ways.

1. **Replace with read (Rw-read)** operator replaces the stack operation associated to transition $t$ with a read operation; that is, for $t = (p, a, X, q, w)$, $Rw-read(t) = (p, a, X, q, X)$. Note that the operator has no effect if $w = \varepsilon$, that is, the operation is already a read operation. Therefore, this operator should only be performed on transitions where a non-read operation occurs.

2. **Replace with push (Rw-push)** operator replaces the stack operation associated to transition $t$ with a push operation. If the operation is already a push operation, a different string is pushed onto the stack. In other words, if $t = (p, a, X, q, wX)$ and $w \in G^*\{\varepsilon\}$, $Rw-push(t, w') = (p, a, X, q, w'X)$ for some given $w' \in G^*\{\varepsilon\}$.

3. **Replace with pop (Rw-pop)** operator replaces the stack operation associated to transition $t$ with a pop operation; that is, for $t = (p, a, X, q, w)$, $Rw-pop(t) = (p, a, X, q, \varepsilon)$. Note that this operator has no effect if $w = \varepsilon$, that is, the operation is already a pop operation.

4. **Replace with pop-push (Rw-poppush)** operator replaces the stack operation associated to transition $t$ with a push followed by a push operation. More precisely, for $t = (p, a, X, q, w)$, $Rw-poppush(t, w') = (p, a, X, q, w')$ for some given $w' \in G^*$, where $w'' \in G^*$ for some $w'' \in G^*$. Note that if $w' = \varepsilon$, only a pop operation is performed, and if $w' = w''X$ for some $w'' \in G^*$, either only a read or only a push operation is performed.

![Stack Mutations](image-url)
with a stack symbol other than initial stack symbol or the new stack top. More precisely, let \( t = (p, a, X, q, w) \): If \( w = ɛ \), \( Rr\text{-}another(t, X') = (p, a, X', q, ɛ) \) for some given \( X' \in G[\{Z\}] \). Otherwise, \( w = w'Y \) for some \( w' \in G^* \) and \( Y \in G \), \( Rr\text{-}another(t, X') = (p, a, X', q, w'Y) \) for some given \( X' \in G(\{Z\} \cup \{Y\}) \).

Figure 4 shows example mutants generated from PDA in Figure 2a using some of the operators in Definitions 7 and Definition 8. In Figure 4a, push operation in transition \((2, d, 1, 3, 21)\) is converted to read operation, and, in Figure 4b, pop operation in \((4, b, 2, 2, 3)\) is converted to push operation. In Figure 4c, symbol read from stack in transition \((3, f, 2, 4, 32)\) is replaced by new stack top symbol 3, whereas in Figure 4d, symbol read from stack in \((4, g, 2, 2, 42)\) is replaced by symbol 1, which is neither top nor initial symbol.

C. Higher Order Mutation Operators

It is possible to apply combined and/or multiple applications of the mutation operators introduced in Sections IV.A and IV.B to create higher order mutants. Although, in general, only first order mutants are used in mutation testing practice, recent works [27][28] show that the use of higher order mutants can also be beneficial. Sometimes, after application of a single mutation operator, the resulting model may turn out to be

- a nondeterministic PDA, or
- a PDA, where some states are not reachable from the start state.

Such first order mutant models can simply be discarded. However, a better way is to perform additional mutations necessary to convert the resulting model to a DPDA and establish reachability of all states (if required). For example, PDA in Figure 2a turns into nondeterministic PDA in Figure 2b after insertion of the transition \((2, b, 1, 2, 3)\). When the transition \((2, b, 1, 2, 3)\) is additionally removed, the resulting PDA becomes deterministic and a second order mutant.

In this paper, we make use of the higher, but generally small, order mutations and try to avoid discarding any generated mutants.

V. COVERAGE CRITERIA

This section defines GB coverage criteria and introduces an algorithm for generating test cases based on these criteria, both of which are designed for a PDA that represents the behavior of the SUC, including GB functions.

A. Go-Back Coverage Criteria

Coverage rates the portion of the system specification or implementation that is covered\(^1\) by the given test suite against the specification or implementation. This ratio is usually used as a decisive factor in determining the point in time at which to stop testing, that is, to release SUC, or to improve it and enhance the test set to continue testing [29].

To be more precise, the coverage \( C \) is defined as \( C = |O'| / |O| \), where

- \( O \) is a finite set of measuring objects,
- \( O' \) is a subset of \( O \) that has been tested, and
- \( |O| \) represents the number of elements of \( O \).

As software testing proceeds, the coverage increases and test engineers can have higher confidence in the software quality. When all the measuring objects of a specific coverage criterion have been executed (that is, when the coverage reaches 100%), it is said that the coverage criterion is satisfied. The coverage is also used as evidence that the software has “enough quality” to be released, and therefore it plays an important role in software development.

There are the following three different levels regarding the GB coverage criteria in our method:

- state GB coverage criterion,
- transition GB coverage criterion, and
- N-switch GB coverage criterion \((N>0)\).

Criterion 1: State GB coverage criterion. The measuring objects of the state GB coverage criterion are the states that have an incoming transition that is triggered by a GB event in a PDA.

When the state GB coverage criterion is satisfied, a test engineer can have confidence that the execution of GB operations is capable of correctly reproducing all reachable states. However, some GB functions may not always work correctly because it does not ensure that all transitions in PDA have been executed. This is the weakest one in GB coverage criteria.

In the example of the internet shopping system, measuring objects of this criterion are the states 1, 2, 3, and 4. When a test case "12a12a4b2h5" is executed, its state GB coverage ratio is 50% (2/4) since only the states 1 and 2 are visited by a GB event.

Criterion 2: Transition GB coverage criterion. The measuring objects of the transition GB coverage criterion are transitions that are triggered by a GB event in PDA.

When the transition GB coverage criterion is satisfied, a test engineer can be confident that GB operations in all states can correctly reproduce all recent reachable states. However, some GB functions may not always work correctly since stack contents are not fully considered in this criterion. Additionally, some GB functions may have side-effects on other state transitions. This criterion subsumes the state GB coverage criterion.

In the example of the internet shopping system, measuring objects of this criterion are the transitions "2b1," "2b2," "2b4," "3b2," "4b2," and "4b3." When the test case "12a12a4b2h5" is executed, its transition GB coverage ratio is about 33% (2/6) since only the transitions "2b1" and "4b2" are executed.

Criterion 3: N-switch GB coverage criterion \((N>0)\). The measuring object of the N-switch GB coverage criterion is a sequence of successive transitions of length \(N+1\) (and of length \(2N+1\) that starts from an initial state), in which a GB event occurs one or more times. When this criterion is satisfied, a test engineer can have confidence that not only a GB operation itself but also its previous and following
Algorithm I runs in $O(|S| (|T|/|S|)^N)$ worst-case time, where $|S|$ is the number of states, $|T|$ is the number of transitions and $N$ is from N-switch coverage. Furthermore, it has a space complexity of $O(|S| (|T|/|S|)^N)$. Note that $N=0$ for transition GB coverage criteria.

VI. CASE STUDY

Following, a case study demonstrates how the proposed concepts are applied to an SUC and how the method works on revealing faults in GB functions. The case study also compares different testing approaches and derives experimental results.

A. System Under Consideration

An abstracted model of a typical internet shopping system is used as SUC. Figure 5 shows its behavior in a state machine diagram, whose transitions are the executions of user operations. The system has five states and eight events, and they are identified by numerical and alphabetical letters respectively. "click Go back button" corresponds to a GB event, and is represented by "b." Transitions labeled by "b" cannot determine their to-states, since this diagram does not include any stack manipulations for GB functions. Thus, the system behavior seems simple, but it has potential complexity due to GB functions.

The behavior of the system is as follows.
- State 1 is the start state, where user enters logon information. After entering the input correctly, the "Logon" button takes the system to state 2.
- In state 2, order information is entered. If the "Credit Card" or "Confirm Order" button is clicked, the system goes to state 3 or 4, respectively. The system stays at state 2, if the user clicks "Clear." Finally, when the "Logoff" button is executed, the system goes to state 5.
- In state 3, the system requests credit card information. Upon correct input and clicking the "OK" button, the system goes to state 4.
- In state 4, the order is confirmed. If the user clicks the "Submit" button, the system confirms the order and goes to state 2.
- State 5 is the final state after logging off.
- In states 2, 3, and 4, the user can go back to the previous state by clicking the "Go back" button. For example, the user can go back to state 2 just after the execution of "1a2c4."
In order to apply our method to this system, Figure 5 needs to be extended into the form that explicitly shows all the stack manipulations of GB functions. Thus, PDA shown in Figure 2a is constructed using the specifications of GB functions described in Section III.C and definition of PDA in Section III.B or in [30]. Compared with the state diagram in Figure 5, the PDA in Figure 2a has about twice the number of transitions. When there are multiple transitions between two states, they are shown as a single arrow with multiple labels to keep the diagrams simpler.

### B. Test Process and Experimental Details

Note that this paper makes the following assumptions. First, it is assumed that all PDA models under consideration are deterministic and every state is reachable from the start state. Thus, when a first order mutant does not satisfy any of those conditions, additional mutations are performed to convert it into a DPDA. Second, it is assumed that, in each transition, a single element is either read from, or popped out of, or pushed onto the stack. This assumption is also preserved while creating the mutants.

To carry out model-based mutation testing of GB functions using PDA model, the following (GB function-related) faults are injected to the system.

1. The event "b" does not occur in the state 3 (the system must go to the state 2).
2. The event "b" occurs in the state 1, and then the system goes to the state 2 (the event "b" must not occur in the state 1).
3. When the event "b" occurs just after the occurrence of "1a2," the system stays at state 2 (the system must go to the state 1).
4. When the event "b" occurs just after the occurrence of "2e2," it performs a read operation instead of pop.
5. When the event "b" occurs just after the occurrence of "2c4," it returns to the state 1 (the system must go to the state 2).
6. Event "b" occurs after logging off in the state 5, and it restores the previous state correctly (the event "b" must not occur in the state 5).

Fault 1 above belongs to fault category 1 shown in Section III.C, faults 2 and 6 belong to fault category 2, and the others belong to fault category 3. The faults can be detected when a positive test case reveals a difference in behavior, or a negative test case reveals no difference.

Mutant PDA models are generated from the original model with respect to different coverage criteria using Algorithm I. Two different test sets for each mutant that satisfy transition GB coverage ($trans_{GB}$) and 1-switch GB coverage ($1switch$), respectively, are generated using the algorithm described in Section V.B.

While executing the generated test cases, the following test execution style is adapted. Each test case is executed until a failure is observed and no more (stop-at-first-failure test execution). This type of test execution is realistic since it prevents the propagation of the failure effects. However, it may also prevent observation of additional failures because some test case parts are not executed. Thus, it is important to stress that, no matter how strong the satisfied coverage is, under stop-at-first-failure test execution, the test set may fail to be as effective as expected. In this sense, experimental observations are quite important.

### C. Test Generation and Execution Results

Table I outlines the results of test generation and test execution obtained from original model and all mutant models with respect to different coverage criteria using Algorithm I.

![Table I](image)

![Table II](image)
Table II gives results of test generation and execution for each type of mutants. Mutation operator names (in Column 1) are used to refer to different type of mutant models.

Note in Table II that data related to Ot and Rr-init mutants are clustered together because Algorithm I outlined in Section V.B for test generation yielded the exact same set of test cases.

Table III gives detailed information on which of the faults (Section VI.B) are revealed by which of the generated test sets.

<table>
<thead>
<tr>
<th>Model</th>
<th>Coverage</th>
<th>Fault IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>transGB</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1switch</td>
<td>5</td>
</tr>
<tr>
<td>It</td>
<td>transGB</td>
<td>1, 3</td>
</tr>
<tr>
<td></td>
<td>1switch</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td>Ot &amp; Rr-init</td>
<td>transGB</td>
<td>1, 3, 4</td>
</tr>
<tr>
<td></td>
<td>1switch</td>
<td>1, 5</td>
</tr>
<tr>
<td>Os</td>
<td>transGB</td>
<td>1, 4</td>
</tr>
<tr>
<td></td>
<td>1switch</td>
<td>1, 5</td>
</tr>
<tr>
<td>Rw-read</td>
<td>transGB</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td></td>
<td>1switch</td>
<td>5</td>
</tr>
<tr>
<td>Rw-push</td>
<td>transGB</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td></td>
<td>1switch</td>
<td>1, 5</td>
</tr>
<tr>
<td>Rr-another</td>
<td>transGB</td>
<td>1, 3, 4</td>
</tr>
<tr>
<td></td>
<td>1switch</td>
<td>1, 3, 5</td>
</tr>
</tbody>
</table>

D. Discussion of the Results and Lessons Learned

Briefly, the results of the case study indicates that during PDA-based testing of GB functions one can increase the number of detected faults in using model-based mutation testing in a complementary (or even alternative) manner to model-based testing. As usual, the use of mutant models may introduce extra effort to the testing.

Test case numbers. Table I shows that the number of test cases increase significantly when mutants are used in addition to original model. For example, 149 new test cases (satisfying 1-switch coverage) are obtained from 35 mutants, in addition to the 12 test cases from the original model.

Identical test cases. As can be seen from Tables I and II, some of the test cases generated from different mutants may be identical. In our experiments, a distinguished case is noted: although the mutants are not equivalent, test cases generated from Ot and Rr-init mutants turn out to be identical. Such cases, if great in number, may result in wasted effort in the mutant generation process.

Revealed faults. Table III shows that the faults revealed using original model are also revealed using each type of mutants, regardless of transition or 1-switch GB coverage.

Coverage. A surprising observation is given in Table I. Test cases achieving transition GB coverage revealed more faults than test cases achieving 1-switch GB coverage, although 1-switch GB coverage is stronger than transition GB coverage. According to our observations, this stems from two main reasons: stop-at-first-failure test execution and the test generation method (which relies on depth-first search). To be more precise, it is observed that many test cases terminate by revealing the same fault since they have identical prefixes. Thus, although they contain different sequences that could reveal more faults, these sequences are not reachable during test execution.

E. Threats to Validity and Limitations

The case study demonstrates that the proposed approach is capable in terms of detecting the faults. However, the following issues still remain.

First, this paper makes use of the PDA model of SUC and focuses on the testing of GB functions. To test other types of special functions or to test the whole system, different coverage criteria and/or test generation methods may need to be selected.

Second, coverage-based test generation is used in the proposed framework. For relatively large systems, this may result in too many test sequences.

Also, the knowledge of the injected faults is used while creating the mutants. In practice, one generally does not possess such knowledge. Therefore, the situation induced here is some kind of best-case scenario. Furthermore, the order of the generated mutants is limited to 1 (or very close to 1). The systematic use of higher order mutants or definition of additional mutation operators may still be beneficial.

Finally, all the results obtained here are valid with respect to the elements defined in Sections IV and V, the information given in Section VI.B, and the SUC described in Section VI.A. In principle, changing one of these parameters, such as using difference test generation methods, etc., may yield different outcomes.

VII. CONCLUSION AND FUTURE WORK

In this paper, a new model-based mutation testing framework for testing of GB functions is proposed by using PDA models. The proposed approach is also validated over a case study by using realistic test case execution logic. The most significant aspects of the research work represented in this paper are as follows.

• In contrast to common model-based testing methods that make use of graph-based models mainly related to regular languages, the proposed model-based mutation testing framework is developed around more powerful PDA models that relate to context-free languages.

• Novel mutation operators are defined for PDA models.

• Recent coverage criteria and a new test generation method are used for mutation testing of GB functions.

On the other hand, the most significant results of the case study are as follows.
• Using PDA model-based mutation testing approach increases the number of detected faults and generated test cases, combining both positive and negative testing.

• Stronger coverage does not necessarily mean detecting more faults. While evaluating quality or adequacy of test cases and test generation methods, relative similarity of generated test cases and employed test execution logic should also be considered.

These results encourage us to consider additional issues to be addressed for further research. For one thing, a larger real-life application can be selected for case study, and properties of the proposed mutation operators can be compared in more detail while also using higher order mutants.

In addition, different test generation methods can be considered so that the number of identical test cases and/or the number of test cases with similar prefixes can be reduced. Using test purposes may prove to be useful, and different test generation methods can be compared using not only their coverage ratios but also different test execution styles.

Also, the proposed mutation testing framework can be extended to support testing of some more general system functions, like undo/redo, or, ultimately, the whole PDA itself.

REFERENCES


