PAPER Special Section on Software Reliability Engineering

Model-Based Mutation Testing Using Pushdown Automata

Fevzi BELLİ[†], **Mutlu BEYAZIT**[†], *Nonmembers*, **Tomohiko TAKAGI**[††a)], *and* **Zengo FURUKAWA**[††], *Members*

SUMMARY A model-based mutation testing (MBMT) approach enables to perform negative testing where test cases are generated using mutant models containing intentional faults. This paper introduces an alternative MBMT framework using pushdown automata (PDA) that relate to context-free (type-2) languages. There are two key ideas in this study. One is to gain stronger representational power to capture the features whose behavior depends on previous states of software under test (SUT). The other is to make use of a relatively small test set and concentrate on suspicious parts of the SUT by using MBMT approach. Thus, the proposed framework includes (1) a novel usage of PDA for modeling SUT, (2) novel mutation operators for generating PDA mutants, (3) a novel coverage criterion, and an algorithm to generate negative test cases from mutant PDA. A case study validates the approach, and discusses its characteristics and limitations.

key words: *model-based testing, mutation testing, pushdown automata, mutation operator, test coverage criteria*

1. Introduction and Related Work

One of the techniques for achieving the required level of software reliability is software testing. Test engineers attempt to effectively detect faults by applying systematic testing techniques to the *software under test* (SUT), and then correct the faults revealed before shipping the SUT. Test cases are constructed based on models, source codes, etc. Test techniques based on models are known as *model-based testing* (MBT) techniques, and the intensive research efforts in this area demonstrate their importance in the field of software engineering. Most of the MBT techniques operate on graph-based abstractions and use some coverage criteria for test generation [1].

MBT is broadly classed as *positive testing* and *negative testing* [2]. In positive testing, test engineers create test cases from correct models (that is, *positive test cases*), and confirm that the SUT runs as specified by the model. In negative testing, test engineers insert faults into the model, and create test cases containing faulty behavior from the faulty models (that is, *negative test cases*) to confirm that the SUT does not conform to the undesirable behavior. The latter is closely related to *model-based mutation testing* (MBMT) [3], [4] which is a relatively new research area on software engineering.

Manuscript received December 25, 2011.
Manuscript revised April 19, 2012.

[†]The authors are with the Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Paderborn, Germany.
[††]The authors are with the Faculty of Engineering, Kagawa University, Takamatsu-shi, 761–0396 Japan.
a) E-mail: takagi@eng.kagawa-u.ac.jp

DOI: 10.1587/transinf.E95.D.2211

In MBMT, a model is injected with intentional faults using mutation operators to produce mutant models (simply called *mutants*). Later, test cases are generated from these mutants to reveal actual faults in the SUT. Thus, based on the selection of mutation operators and test generation methods, MBMT is used to perform positive and/or negative testing.

MBMT differs radically from conventional mutation testing [5], [6]. Conventional mutation testing requires insertion of intentional faults by changing the source code. Its main purpose is to evaluate the adequacy of a given test set, that is, to check whether this set can reveal the injected faults or not. On the other hand, MBMT strives for immediate testing of the SUT and enables to perform negative testing. It is still possible to adapt and use it for adequacy evaluation by executing the given test cases on mutants, which leads to *model-based mutation analysis*.

Previous studies on MBMT generally make use of *regular models* such as finite state machines [7], event sequence graphs [3] and regular grammars [8], which represent regular (type-3) languages in Chomsky hierarchy. However, such models can hardly represent the features whose behavior depends on previous states of the software, as exemplified in the following.

- Some software features save interim results and invoke other features. After completing the invoked features, they determine their subsequent behavior based on not only the results returned from the invoked features but also the interim results.
- Most software includes the feature to cancel recent operations and subsequently go back to the previous state, which is generally known as *undo* [9], [10].

In real-life systems, such features are almost always realized by keeping the track of previous states in memory. Likewise, representing such features in models also requires memory, which is absent in regular models.

In this paper, we propose an alternative MBMT framework using pushdown automata (PDA) that relate to context-free (type-2) languages. There are two key ideas in this study: (1) To use the PDA stack as the memory in order to gain stronger representational power. (2) To make use of a relatively small test set and concentrate on suspicious parts of the SUT (because a PDA potentially has a large state space, and a large test set is required to cover it in positive testing). Thus, to perform negative testing, PDA mutants with faulty transitions and N-switch faulty transi-

tion coverage criterion are used, and, to perform positive testing, the original PDA and N-switch transition coverage criterion are used (See Sects. 3–4). Therefore, the proposed MBMT framework includes

- a novel usage of a PDA for modeling SUT,
- novel mutation operators for generating PDA mutants,
- novel coverage criteria and an algorithm to generate negative/positive test cases from mutant/original PDA.

The rest of this paper is organized as follows. Section 2 outlines background information on PDA models. Section 3 introduces PDA mutation operators, whereas Sect. 4 discusses coverage criteria and an algorithm for negative/positive test case generation. Before Sect. 6 concludes the paper, Sect. 5 performs a comprehensive nontrivial case study to demonstrate the approach and analyze its characteristics.

2. Pushdown Automata

This section introduces the PDA models used in the paper.
Definition 1: A *pushdown automata* (*PDA*) is a tuple $M = (S, E, G, T, S0, Z0, F)$ where

- S is a finite set of *states* (or *state alphabet*),
- E is a finite set of *events* (or *event alphabet*),
- G is a finite set of *stack symbols* (or *stack alphabet*),
- T: $S \times E \cup \{\varepsilon\} \times G \rightarrow U$ ($U \subseteq S \times G^*$ is finite) is the *transition function* (ε is the *empty string*),
- $S0 \in S$ is the *initial state*,
- $Z0 \in G$ is the *initial stack symbol*, and
- $F \subseteq S$ is the set of *final states*.

Transition function T receives as input a triple (p, a, X), where p is the current state, a is the event 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. Thus a *transition* can be represented by 5-tuple (p, a, X, q, w). A *read* operation occurs if $w = X$, a *pop* operation is performed if $w = \varepsilon$ and Y is *push*ed onto the stack if $w = YX$. Also, a PDA is *deterministic*, if it satisfies the following properties:

- $|T(p, a, X)| = 1$ for each $p \in S$, $a \in E \cup \{\varepsilon\}$ and $X \in G$.
- For each $p \in S$ and $X \in G$, if $T(p, \varepsilon, X) \neq \emptyset$ then $T(p, a, X) = \emptyset$ for every $a \in E$.

In this paper, we use PDA models which satisfy the following properties. (1) They are deterministic (with no ε-transitions). (2) $G - \{Z0\} \subseteq S$. (3) Every state is reachable from $S0$ and a final state is reachable from each state.

We use state transition tables to represent PDA. Table 1 illustrates an example PDA model with $S = \{1, 2, 3, 4\}$, $E = \{a, b, c, d\}$, ten transitions, $G = \{0, 2, 3\}$, $S0 = 1$, $Z0 = 0$ and $F = \{4\}$. In the table, rows and columns correspond to states and events, respectively, and each cell contains the transition labels upon occurrence of an event in a state. For example, when the PDA receives event b in state 3, it performs transition labeled by $2/3 : 2, 2$, that is, $(3, b, 2, 2, 3 : 2)$. This PDA can also be represented as in Fig. 1 [11].

Table 1 Example PDA model - Transition table.

		a	b	c	d
		Event a	Event b	Event c	Event d
1	Initial state	0/0, 2			
2	State 2		0/2:0, 3 2/2:2, 3 3/2:3, 3	2/2, 2 3/3, 4	2/ε, 2 3/ε, 3
3	State 3		2/3:2, 2	2/2, 2	
4	Final state				

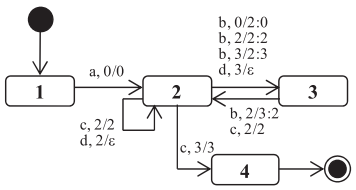

Fig. 1 Example PDA model.

From the modeling point of view, states of a PDA can be classified as *historic states* and *non-historic states*. A historic state relates to a main feature of SUT and affects the transitions in other states as a stack symbol. In Table 1, state 2 and 3 correspond to historic states and the others correspond to non-historic states. Likewise, events of a PDA can be classified into the following three types:

- A *read event* triggers only transitions that perform read operations, such as events a and c in Table 1.
- A *pop event* triggers only transitions that perform pop operations, such as event d in Table 1. This type relates to undo or cancel operations in SUT.
- A *push event* triggers only transitions that perform push operations, such as event b in Table 1. Events of this type are acceptable in historic states, and keep track of the historic states for future references.

A PDA has a stronger expressive power when compared to many other formal models. For example, a *regular model*, that is, a model which represents a regular (type-3) language in Chomsky hierarchy, results in an infinite state space while modeling a behavior represented by a PDA, unless the stack size is restricted (Since the states need to be defined as elements in $S \times G^*$ in order to represent the equivalent behavior). Even if the stack size is strictly restricted, the model may become too large to work with. Of course, models like UML state machine diagrams [12] and UML profiles with action languages also have stronger representational powers. However, they include informal representations or specific issues of programming languages, and thus need further formalizations and/or abstractions, respectively. More importantly, in their informal representation, they do not enable to use results of automata theory, which are very useful for test generation.

MBT aims to use simple models and increase the efficiency of the test process. Our modeling technique using PDA is indeed simple in structure. Nevertheless, it can be applied to complex, real-life systems. This is feasible, because we simplify some of the real-life features, that is, we abstract from irrelevant to focus on relevant, which is common in MBT [4] and explains the success of the widely accepted MBT techniques using simple, easy-to-understand approaches to be applied to practical systems. Here, we capture more intrinsic features using stronger, context-free PDA models, making the approach even more powerful than and preferable to the use of regular models of simpler structure which are already applied to actual software system [3], [7], [8].

3. Mutation Operators

Mutation operators are applied to the original model. This section includes *basic mutations* (whose combined and repeated applications can be used to transform a given PDA to any other PDA with the same event and stack alphabets), *transition corruptions* (which are applied to modify the existing transitions keeping the state, event and stack alphabets fixed), and important testing aspects.

3.1 Basic Mutations

As partially introduced in [3], basic mutation operators are *insertion* (I), *omission* (O), and *marking* (M) operators.

In general, insertion operators are used to generate mutants that have additional functionality when compared to the original model, omission operators are used to generate (correct) sub models and marking operators are used to change the type of certain model elements.

A PDA can simply be represented by a multi-directed graph. Nodes of this graph are the states of the PDA, and edges represent 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 2: *Transition mutation operators*. Given a PDA $M = (S, E, G, T, S0, Z0, F)$:

- *Transition insertion* (It) *operator* adds a new transition t to M. If $t = (q, a, X, p, w)$. It is assumed that $q, p \in S$, $a \in E \cup \{\varepsilon\}$, $X \in G$, $w \in G^*$ and $(p, w) \notin T(q, a, X)$. An insertion may also generate a non-deterministic PDA.
- *Transition omission* (Ot) *operator* deletes an existing transition t from M. If $t = (q, a, X, p, w)$, it is assumed that $q, p \in S$, $a \in E \cup \{\varepsilon\}$, $X \in G$, $w \in G^*$ and $(p, w) \in T(q, a, X)$. It is possible that an omission may leave some states with no incoming or outgoing transitions.

Definition 3: *State mutation operators*. Given a PDA $M = (S, E, G, T, S0, Z0, F)$:

- *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.
- *State omission* (Os) *operator* deletes an existing state q together with all the transition ingoing to and outgoing from q. After the deletion, some states in M may lose all their incoming or outgoing transitions.

Definition 4: *Marking operators*. Given a PDA $M = (S, E, G, T, S0, Z0, F)$:

- *Mark start* (Ms) *operator* marks an existing state in M as the start state and the old start as a non-start.
- *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.

3.2 Transition Corrupting Mutations

Sometimes basic mutations are too vague to use. To generate some specific faulty behaviors, one needs only to modify or corrupt the existing transitions. In this way, the use of higher order mutations, which results in a huge number of mutants, can also be avoided.

Here, more precise mutation operators that are relatively more suitable for PDA-based mutation testing or negative testing are defined. The operators can also be seen as the controlled combinations of basic mutations. They introduce specific faults into PDA models by corrupting the transitions without modifying the sets of states, events and stack symbols.

Definition 5: *Write replacement operators*. Given a PDA $M = (S, E, G, T, S0, Z0, F)$, *write replacement* ($Rw$) *operator* replaces the string to be put into the stack by the given string w', that is, for $t = (p, a, X, q, w)$, $Rw(t, w') = (p, a, X, q, w')$ where $w' \in G^* - \{w\}$. This operator can be performed in 4 different ways.

- *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 $w = X$, that is, the operation is already a read operation. Therefore, this operator should only be performed on transitions where a non-read operation occurs.
- *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 $w' \in G^* - \{\varepsilon, w\}$. Otherwise, *Rw-push*$(t, w') = (p, a, X, q, w'X)$ for some given $w' \in G^* - \{\varepsilon\}$.

- *Replace with pop* (*Rw-pop*) *operator* replaces the stack operation associated to transition t with a pop operation; that is, for $t = (p, a, X, q, w)$, $Rw\text{-}pop(t) = (p, a, X, q, \varepsilon)$. Note that this operator has no effect if $w = \varepsilon$, that is, the operation is already a pop operation.
- *Replace with pop-push (Rw-poppush) operator* replaces the stack operation associated to transition t with a pop followed by a push operation. More precisely, for $t = (p, a, X, q, w)$, $Rw\text{-}poppush(t, w') = (p, a, X, q, w')$ for some given $w' \in G^*$, where $w' \notin \{\varepsilon, w''X\}$ 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 a read or a push operation is performed.

Definition 6: *Read replacement operators*. Given a PDA $M = (S, E, G, T, S0, Z0, F)$, *read replacement* (*Rr*) *operator* replaces the symbol on the top of the stack by the given symbol X'; that is, for $t = (p, a, X, q, w)$, $Rr(t, X') = (p, a, X', q, w)$ where $X' \in G - \{X\}$. This operator can also be performed in different ways.

- *Replace with initial stack symbol* (*Rr-init*) *operator* replaces the symbol read from stack in transition t with the initial stack symbol $Z0$; that is, for $t = (p, a, X, q, w)$, $Rr\text{-}init(t) = (p, a, Z0, q, w)$. Note that the operator has no effect if $X = Z0$; that is, top symbol is already initial stack symbol.
- *Replace with new stack top* (*Rr-top*) *operator* replaces the symbol read from stack in transition t with the new stack top; that is, for $t = (p, a, X, q, w)$, where $w = Yw'$, $w' \in G^*$ and $Y \in G$, $Rr\text{-}top(t) = (p, a, Y, q, w)$. This operator converts the operation in transition t to a push operation. Therefore, it is not applicable when a pop operation occurs; that is, $w = \varepsilon$, and has no effect if a push operation is performed, that is, $Y = X$.
- *Replace with another stack symbol* (*Rr-another*) *operator* replaces the symbol read from stack in transition t with a stack symbol other than initial stack symbol or the new stack top. More precisely, let $t = (p, a, X, q, w)$: If $w = \varepsilon$, $Rr\text{-}another(t, X') = (p, a, X', q, \varepsilon)$ for some given $X' \in G - \{Z0\}$. Otherwise, $w = Yw'$ for some $w' \in G^*$ and $Y \in G$, $Rr\text{-}another(t, X') = (p, a, X', q, w'Y)$ for some given $X' \in G - (\{Z0\} \cup \{Y\})$.

Definition 7: *Event replacement operator*. Given a PDA $M = (S, E, G, T, S0, Z0, F)$, *event replacement* (*Re*) *operator* replaces the event in a transition by another event, that is, for $t = (p, a, X, q, w)$, $Re(t, b) = (p, b, X, q, w)$ where $b \in (E \cup \{\varepsilon\}) - \{a\}$.

Definition 8: *Source replacement operator*. Given a PDA $M = (S, E, G, T, S0, Z0, F)$, *source replacement* (*Rs*) *operator* replaces the source state in a transition by another state, that is, for $t = (p, a, X, q, w)$, $Rs(t, s) = (s, a, X, q, w)$ where $s \in S - \{p\}$.

Definition 9: *Destination replacement operator*. Given a PDA $M = (S, E, G, T, S0, Z0, F)$, *destination replacement* (*Rd*) *operator* replaces the destination state in a transition by another state, that is, for $t = (p, a, X, q, w)$, $Rd(t, s) = (p, a, X, s, w)$ where $s \in S - \{q\}$.

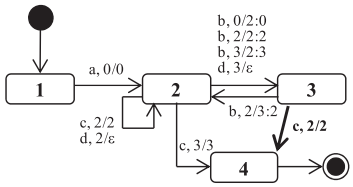

Fig. 2 Example PDA mutant.

Note that to induce a faulty behavior by performing a transition corrupting mutation, the mutated transition should not be already in the original model.

3.3 Testing Aspects

For (model-based) mutation testing of a system, generally small changes are inserted by using first (or small) order mutations. Although, the use of higher order mutants is sometimes quite beneficial [7], [13], [14].

Using (combinations of) the mutations, one can generate 3 types of mutants: (1) Mutants which do not contain any faulty behavior. (2) Mutants which contain faulty behavior that can be detected by some positive test case. (3) Mutants which contain faulty behavior that cannot be detected by any positive test case. For example, considering first order mutants, all omission mutants are type 1, transition corrupting mutants and insertion mutants are either type 2 or type 3.

All these different types have their uses in practice, and their performances depend on the test generation and execution strategies used. For example, type 1 mutants can be used to generate positive test cases by leaving out some parts of the system, whereas type 2 mutants can be used when covering the whole system is not preferred or only specific types of faults need to be tested, and type 3 mutants can be used to perform 'pure' negative testing.

Nevertheless, the use of some mutations is generally avoided since they are in general less useful (for example, omission mutants, which do not contain any faults, and marking mutants, which are mostly useful for validation or testing of the model) or quite hard to use (for example, state insertions, which contain relatively larger changes).

Also, in practice, it is usual to discard (mutant) models that do not satisfy some certain properties, or further process them so that they satisfy the intended properties. Examples of such properties are determinism and usefulness (or reachability) of model elements.

Figure 2 shows an example mutant of the PDA in Fig. 1. The mutant is a type 2 mutant and can be generated in different ways. One can insert $(3, c, 2, 4, 2)$ and then omit $(3, c, 2, 2, 2)$ to remove non-determinism, or one can replace the destination of $(3, c, 2, 2, 2)$ by 4.

4. Coverage Criterion

This section defines a coverage criterion for faulty PDA models, that is mutants, and an algorithm for generating

negative test cases based on this criterion. Related positive testing aspects are also briefly discussed.

4.1 N-switch Faulty Transition Coverage Criterion

Coverage rates the portion of the system that is covered by the given test set. This ratio is usually used as a decisive factor in determining the point in time at which to stop testing, that is, to release SUT, or to improve it and enhance the test set to continue testing [15]. 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. The definition of the measuring objects depends on the *coverage criterion* used.

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, it is said that the coverage criterion is *satisfied*.

In order to generate negative test cases from PDA mutants, we propose a novel coverage criterion called *N-switch faulty transition coverage* (*fixed* $N \geq 0$), which is developed based on the coverage for finite state machines [16]. Its measuring object is a sequence of $N + 1$ successive transitions containing stack top part (that is, some symbols on the top of the stack). More precisely:

A) The length of the transition sequences to be covered is $N + 1$ (or less if the transition sequences start from an initial state).
B) The length of the stack top part to be covered is N.
C) At least one faulty transition appears in each transition sequence.

When this criterion is satisfied, a test engineer can have the confidence that not only a suspicious operation is itself working correctly but also it is working correctly in the contexts induced by transition sequences where previous and following operations are also included. As the value of N gets larger, test engineers tend to have higher confidence in software quality, but then the size of the measuring objects (that is, the size of the test cases) also becomes larger.

In Fig. 2, measuring objects for 1-switch faulty transition coverage are three transition sequences of length 2: $(2, b, 3, 3, 2:3) \rightarrow (3, c, 2, 4, 2)$ with stack top 2, $(2, b, 0, 3, 2:0) \rightarrow (3, c, 2, 4, 2)$ with stack top 2, and $(2, d, 3, 3, \varepsilon) \rightarrow (3, c, 2, 4, 2)$ with stack top 2. When a test case $(1, a, 0, 2, 0) \rightarrow (2, b, 0, 3, 2:0) \rightarrow (3, c, 2, 4, 2)$ is executed, its coverage is about 33% (1/3) since it contains only transition sequence $(2, b, 0, 3, 2:0) \rightarrow (3, c, 2, 4, 2)$ with stack top 2.

4.2 Algorithm for Negative/Positive Test Case Generation

After completing the construction of a PDA model and generation of mutant PDAs containing faults, test process proceeds to test case generation. Test cases satisfying N-switch faulty transition coverage for a given mutant can systematically be generated by using Algorithm 1.

Algorithm 1: *PDA-based negative test case generation algorithm* consists of the following steps.

Step 1. Set the initial state as the current state, and begin to search the PDA.
Step 2. Select an executable outgoing transition in the current state.
- If the execution of the selected transition results in the execution of a new search object, it is executed and is added to the test case under construction. Here a search object is a sequence of successive transitions with the stack top part that satisfies A) and B) given in Sect. 4.1.
- If the selected transition does not result in a new search object, select another transition.

Step 3. Repeat Step 2 until no new search object can be found.
Step 4. If there is a transition that is not selected in Step 2 (that is, a transition that has a possibility of deriving a new search object), backtrack to a previous state that has such a transition.
Step 5. Repeat from Step 2 to Step 5 similarly.
Step 6. If there is a test case that includes no measuring object, or there is a test case in which all the measuring objects are included in another test case, eliminate such a test case in order to derive a final set of test cases.

An example of test case generation from Fig. 2 is shown in Fig. 3. The nodes of the search path are search objects. Also, (1), (2), (3), (4) and (5) correspond to Steps 1, 2, 3, 4 and 5 of Algorithm 1, respectively.

Algorithm 1 is developed based on depth-first search in directed graphs. It is obvious that a set of measuring objects is a subset of search objects. When a search object includes a faulty transition, it is identified as a measuring object. Covering all the search objects is indispensable for finding all the measuring objects since some measuring objects do not become executable unless specific search objects are previously executed.

Consequently, our implementation of Algorithm 1 runs

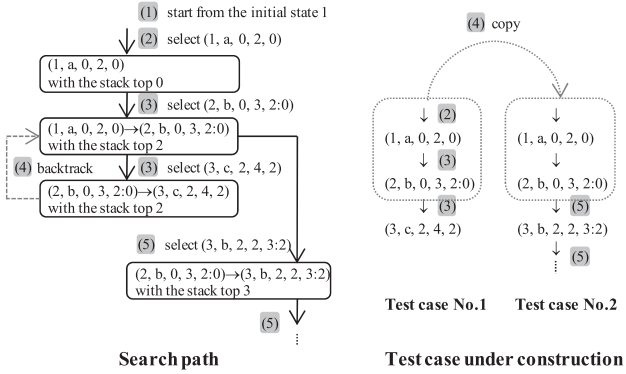

Fig. 3 Example of a test case generation from Fig. 2.

Table 2 Overview of test case generation.

N	Number of Search Objects / Measuring Objects		Number of Test Cases		Average Length of Test Cases	
	Positive	Negative	Positive	Negative	Positive	Negative
0	10 / 10	7 / 1	3	1	7.3	5.0
1	29 / 29	11 / 3	10	3	11.0	5.0
2	99 / 99	18 / 4	37	4	25.8	6.0
3	421 / 421	29 / 8	171	8	85.7	7.3
4	1708/1708	44 / 10	714	10	364.3	8.2

in $O(|S|(|T|/|S|)^{(N+1)})$ worst-case time, where $|S|$ is the number of states, $|T|$ is the number of transitions and N is from N-switch coverage. Also, it has a space complexity of $O(|S|(|T|/|S|)^{N})$.

Also, when Algorithm 1 is applied without Step 6 to an original PDA model, one can obtain positive test cases that satisfy a novel criterion, called *N-switch transition coverage* (*fixed* $N \geq 0$), where its measuring object is a sequence of $N + 1$ successive transitions with stack top part; more precisely, defined by A) and B) in Sect. 4.1.

A positive or negative test case reveals a fault when the observed behavior does not match the expected one. The expected behavior is derived by executing the test case on the original (correct) model and the observed behavior is obtained by executing it on the SUT. The only difference between a positive and a negative test case is that a negative test case contains a faulty transition.

Table 2 shows the overview of positive and negative test case generation from Fig. 1 and Fig. 2, respectively. As N becomes larger, the number of test cases increases significantly. For positive test cases, this increase is in general exponential. Therefore, in testing practice, N-switch based coverage criteria are used by selecting relatively small values for N (like $N \leq 1$) in order to keep the overall testing process efficient and scalable.

5. Case Study

This section includes a nontrivial case study which shows the testing process. Positive and negative testing approaches are compared using a realistic example and the results are used to discuss the effectiveness, threats to validity, and limitations.

5.1 System under Test

In the case study, new message creation function in typical e-mail software of a mobile phone is used as the SUT. Its behavior is as follows.

- When "Create a new message" is selected from the menu, the SUT is invoked. Initially, address, title and body fields are void, and edit menu is displayed. If "Cancel" is selected, the SUT is terminated.
- When either "Edit the address", "Edit the title" or "Edit the body" is selected, the SUT starts a corresponding edit mode. Upon selection of "Cancel", the edit mode is canceled. In edit mode, the user can fill out or delete the contents of the fields in arbitrary order.
- A recent edit operation is canceled by selecting "Undo".
- When the address field is not void, the user can select "Send" to send the message. If the user selects "Cancel" or a time-out occurs, sending the message is interrupted. After the message is successfully sent, the SUT is terminated.

Table 3 Simplified PDA model of a new message creation function.

		a	b	c	d	e	f	g
		Create a new message	Edit the address or title or body	Fill out or delete	Undo	Send	Successfully sent	Time-out
1	Initial state	0/0, 2						
2	Selecting an item from edit menu		0/2:0, 3 2/2:2, 3		2/ε, 2	2/2:2, 4		
3	Editing the address or title or body			2/2, 2				
4	Sending						2/2, 5	2/ε, 2
5	Final state							

The complete PDA model is quite large, consisting of 14 states, 12 events, 265 transitions and 9 stack symbols (See [17]). Due to lack of space, only a simplified model is given here in Table 3.

5.2 Experiment Details

The main purpose of this case study is to compare our approach against positive testing, and to confirm that it works well on such realistic systems. To do this, this case study makes use of model-based mutation testing, and the following steps are performed.

Step 1. The original PDA model is constructed based on the above specifications (It requires about four man-hours).
Step 2. To create a faulty SUT, 11 faults are injected using the mutation operators defined in Sect. 3. Later, to generate negative test cases mutants which contain similar faulty behaviors are generated.
Step 3. Positive test cases that satisfy the N-switch coverage and negative test cases that satisfy the N-switch faulty transition coverage ($N = 0, 1$) are generated from the original and mutants using Algorithm 1.
Step 4. Tests are executed on the faulty SUT to detect the injected faults and to collect data.
Step 5. Test generation and execution data are used to make realistic comparisons and analysis.

In Step 2, when a mutant does not satisfy the properties (1)–(3) discussed in Sect. 2, additional mutations are performed to convert it into a mutant that satisfies all the properties. Also, in the light of the discussion in Sect. 3.3, only *It*, *Rw-read*, *Rw-push*, *Rw-pop*, *Rw-poppush*, *Rr-init*,

Table 4 Overview of test generation and execution results.

Approach	N	Number of Test Cases	Average Length	Number of Events Executed	Number of Faults	Fault Detection Performance
Positive	0	117	56.6	7369	10	0.136%
	1	1403	432.6	609015	10	0.002%
Negative	0	11	63.8	1389	11	0.792%
	1	84	415.1	37366	11	0.029%

Rr-top, *Rr-another*, *Re*, *Rs* and *Rd* operators are used.

In Step 4, test cases are executed using the following strategy: Each test case is executed until its completion. When a fault is detected by a test case, first the fault is corrected, and then the test case is executed again. This process is repeated until all test cases are completed. To measure the performance, the number of revealed faults and number of executed events are counted (incrementally). Also, during test execution, states, events and stack top elements are assumed to be observable.

5.3 Test Generation and Execution Results

After test generation, test cases are executed on the faulty system. Various data are collected to give some insights on the respective performances of positive and negative testing approaches. Table 4 outlines the overall test generation and execution results, where fault detection performance is calculated by the ratio of number of faults revealed to number of events executed.

Furthermore, to observe the fault detection trend, the plots of "number of revealed faults vs. number of executed events" for 0-switch and 1-switch coverage are given in Fig. 4 and Fig. 5, respectively. Note that we select $N \leq 1$ to avoid from generating too many test cases and keep the process efficient/practical (See Sect. 4.2).

5.4 Discussion of the Results

Table 4 suggests that positive testing fails to detect one of the faults (in both cases). For both cases, this fault is the same one and it represents an additional unexpected behavior. Thus, positive test cases cannot detect such faults, whereas negative test cases are quite useful when correct mutants are used. These types of faulty behaviors are demonstrated by type 3 mutants.

Also, since fewer test cases are generated, fewer events are executed in total and 1 extra fault is detected using negative testing approach, there is a significant difference between fault detection performances. Thus, when correct mutants are selected, negative testing contributes to cost-efficiency positively and greatly.

Surprisingly, for both positive and negative testing, increasing switch value (N) decreases the fault detection performances, since the same faults are already revealed by 0-switch test cases by using fewer test cases.

As Fig. 4 and Fig. 5 demonstrate, in positive testing, there are long sequences of event executions which reveal

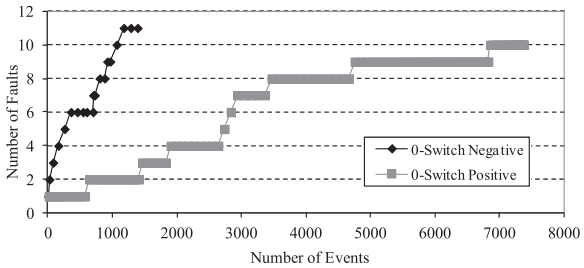

Fig. 4 Fault data for 0-switch criteria.

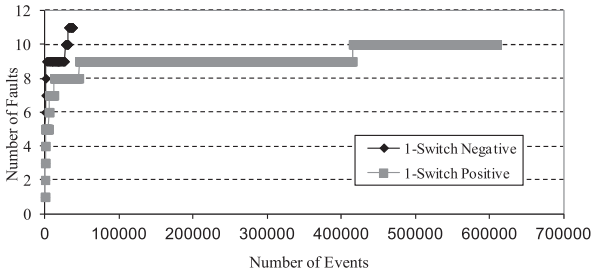

Fig. 5 Fault data for 1-switch criteria.

no additional faults, because positive tests cover the system in general and do not focus on specific faults. Thus, when one identifies the mutants properly and generate negative test cases, these long sequences of event executions can be shortened to a great extent.

5.5 Threats to the Validity

The case study demonstrates that negative testing approach is very capable of increasing the efficiency of testing process. However, the following issues remain.

The generated mutants are identified based on the injected faults. Therefore, a best case scenario is created for the faulty system. It would make sense to devise further experiments to identify the average and the worst cases, or introduce fault-prone analysis techniques and meta-heuristics for (approximate) mutant selection.

Also, a single application is used in the case study. While devising further experiments, including various different types of applications would increase the reliability of the results from a practical point of view.

Finally, all the obtained results are valid with respect to the discussion made in Sects. 3, 4 and 5.1, and most importantly, Sect. 5.2. In principle, changing one of these parameters may yield different outcomes.

6. Conclusion and Future Work

In this paper, a new model-based mutation testing framework is introduced. The use of pushdown automata in this framework increases the modeling power. Furthermore, with the definition of novel mutation operators, coverage criteria and related test generation, pushdown automata can be used for mutation testing.

A non-trivial case study is also performed to outline the advantages of the discussed framework by comparing positive and negative testing approaches. The results show that negative testing approach considerably contributes to the efficiency and, surprisingly, stronger coverage does not necessarily result in a higher fault detection performance.

Still, already mentioned in Sect. 5.5, there are additional issues to consider. For one thing, further experiments can be performed to investigate worst case and average case scenarios over multiple systems having different properties. The obtained results can be used to derive and evaluate feature analysis techniques or meta-heuristics to approximate best case scenarios.

In addition, different test execution strategies considering the properties of generated test cases and new test generations can be developed to further increase the efficiency of the process. The strategies can be evaluated using the test generation methods.

Also, further experimental results can be obtained using different test oracles (depending on different application areas) and effectiveness of mutation operators can be compared using different oracles to determine respective target application areas.

References

[1] B. Beizer, Software Testing Techniques, 2nd ed., Van Nostrand Reinhold, 1990.
[2] F. Belli, "Finite-state testing and analysis of graphical user interfaces," Proc. 12th International Symposium on Software Reliability Engineering (ISSRE2001), IEEE, pp.34–43, Nov. 2001.
[3] F. Belli, C.J. Budnik, and E. Wong, "Basic operations for generating behavioral mutants," Proc. 2nd Workshop on Mutation Analysis (MUTATION2006), IEEE, pp.9–18, Nov. 2006.
[4] M. Weiglhofer, B. Aichernig, and F. Wotawa, "Fault-based conformance testing in practice," International Journal of Software and Informatics, vol.3, no.2-3, pp.375–411, June/Sept. 2009.
[5] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on test data selection: Help for the practicing programmer," Computer, vol.11, no.4, pp.34–41, April 1978.
[6] R.G. Hamlet, "Testing programs with the aid of a compiler," IEEE Trans. Softw. Eng., vol.SE-3, no.4, pp.279–290, July 1977.
[7] F. Belli and M. Beyazit, "Event-based mutation testing vs. state-based mutation testing - an experimental comparison," Proc. 2011 IEEE 35th Annual Computer Software and Applications Conference (COMPSAC 2011), IEEE, pp.650–655, July 2011.
[8] F. Belli and M. Beyazit, "A formal framework for mutation testing," Proc. 2010 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010), IEEE, pp.121–130, June 2010, (Revised version: http://adt.et.upb.de/download/papers/BB2010_SSIRI2010corrected.pdf.)
[9] T. Takagi and Z. Furukawa, "GB coverage criteria: the measurement for testing a "go back" function based on a pushdown automaton," Proc. 19th International Symposium on Software Reliability Engineering (ISSRE 2008), IEEE, pp.293–294, Nov. 2008.
[10] F. Belli, M. Beyazit, T. Takagi, and Z. Furukawa, "Mutation testing of "go-back" functions based on pushdown automata," Proc. 4th International Conference on Software Testing, Verification and Validation (ICST 2011), IEEE, pp.249–258, March 2011.
[11] J.E. Hopcroft, R. Motwani, and J.D. Ullman, Introduction to Automata Theory, Languages and Computation, 2nd Edition, Addison-Wesley, 2000.
[12] Object Management Group, UNIFIED MODELING LANGUAGE, http://www.uml.org/.
[13] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants," Software Testing, Verification and Reliability, vol.19, no.2, pp.111–131, June 2009.
[14] Y. Jia and M. Harman, "Higher order mutation testing," Information and Software Technology, vol.51, no.10, pp.1379–1393, Oct. 2009.
[15] H. Zhu, P.A. Hall, and J.H. May, "Software unit test coverage and adequacy," ACM Comput. Surv., vol.29, no.4, pp.366–427, Dec. 1997.
[16] T.S. Chow, "Testing software design modeled by finite-state machines," IEEE Trans. Softw. Eng., vol.SE-4, no.3, pp.178–187, May 1978.
[17] F. Belli, M. Beyazit, T. Takagi, and Z. Furukawa, "PDA model of a new message creation function in e-mail software of a mobile phone," http://stwww.eng.kagawa-u.ac.jp/˜takagi/fullmodel.pdf or http://quebec.upb.de/ieice2012_fullmodel.pdf.

Fevzi Belli received the M.S. and Ph.D. degrees from Berlin Technical University in 1973 and 1978, respectively. He worked as a software engineer in Munich before he changed in 1983 to the University of Applied Sciences in Bremerhaven. Since 1989, he is a professor of Software Engineering at the University of Paderborn and works on software reliability/fault tolerance, model-based testing, and test automation.

Mutlu Beyazıt received the B.S. and M.S. degrees from İzmir Institute of Technology in 2005 and 2008, respectively. In 2009, he started Ph.D. in the Department of Computer Science, Electrical Engineering and Mathematics at the University of Paderborn. His current interests lie in the area of model-based system and software testing.

Tomohiko Takagi received the B.S., M.S. and Ph.D. degrees from Kagawa University in 2002, 2004 and 2007, respectively. Since 2008, he has been an assistant professor in the Faculty of Engineering at Kagawa University. His research interests are in software engineering, particularly software testing.

Zengo Furukawa received the B.S. and M.S. degrees from Kyushu University in 1975 and 1977 respectively, and joined the Hitachi Systems Development Laboratory. In 1986, he became a research associate at Kyushu University, a lecturer in 1990, and an associate professor at the Educational Center for Information Processing in 1992. Since 1998, he has been a professor in the Faculty of Engineering at Kagawa University. He holds a Ph.D. degree.