Software Testing Technique Based on an Extended Pushdown Automaton for Undo/Redo Functions

Tomohiko Takagi Zengo Furukawa
Dept. of Reliability-based Information Systems Engineering,
Faculty of Engineering, Kagawa University
Takamatsu, Kagawa, Japan

Abstract—**We propose a new testing technique for undo/redo functions (UR functions) that play an important role in realizing the usability of software. In this technique, an extended pushdown automaton (ePDA) that is a state machine with two stacks for UR functions is used for defining software specifications and generating testcases. This paper shows the overview of UR functions, the definitions of an ePDA and coverage criteria, a simple example, and so on.**

Keywords - software testing; undo/redo; state machine; pushdown automaton; testcase; coverage

I. Introduction

Most software has undo/redo functions (UR functions). The undo function is for canceling a previous user operation and returning to a previous state of software, and the redo function is for reperforming a user operation canceled by an undo function. UR functions play an important role in realizing the usability of software, and their faults seriously affect the quality of software. However, a testing technique for UR functions has not been established.

Reference [1] showed the simple model-based testing technique using a pushdown automaton (PDA) for testing undo functions. A PDA is a state machine with a stack, and it can be used as a description language of software specifications. Recording the history of state transitions on the stack enables testcase generation in which undo functions are included. However, [1] didn't consider redo functions and the detailed mechanism of UR functions. So in this paper we propose a new testing technique for UR functions by extending the technique of [1].

II. Undo/Redo Functions

This section shows the overview of the mechanism of UR functions, and then illustrates its faults.

UR functions provide users the way to cancel a previous user operation or to reperform a canceled user operation. There are user operations that UR functions are not available to; for instance, UR functions can't be executed for a user operation that interacts with external systems, or that doesn't change the state and data of software. Regarding user operations that UR functions are available to, their execution history is stored in the memory for UR functions (stacks with limited capacity). The history includes the previous state of software, differences of data resulting from an executed user operation, and so on. Some user operations may trigger the initialization of the stacks; for instance, editor software would initialize the stacks when its user closes his manipulated file. When the size of a history exceeds the capacity of a stack, the oldest element in the history is automatically removed in order to store a new element.

The typical faults of UR functions are as follows:

- UR functions aren't available to a user operation that the UR functions should be available to.
- Performing UR functions can't reproduce the previous state and data of software correctly.
- Software has unpredictable behavior because the memory for UR functions was destroyed.

The difficulty of testing UR functions is in that a failure may not appear soon even if a faulty function was executed. It requires the systematic testing technique for UR functions.

III. Technique Overview

A. Definitions

This section shows the technique overview of testing UR functions using an extended PDA (ePDA). This technique consists of the following four definitions.

Definition 1: ePDA for describing software specifications

$$ePDA = (S, i, F, E, u, r, A, T, H_u, H_r),$$

where S is a set of states in $ePDA$; i is an initial pseudo state in $ePDA$; F is a set of final pseudo states in $ePDA$; E is a set of events in $ePDA$, and represents general user operations (UR functions are available to part of them, but they don't invoke UR functions); u and r are special events in $ePDA$, and represent user operations of invoking undo functions and of invoking redo functions respectively; A is a set of actions to manipulate H_u (a stack for undo functions) and H_r (a stack for redo functions) in $ePDA$, and it is expressed as $A = \{a_p, a_n, a_i\}$; a_p is the action to push a current state onto H_u, a_n is the action not to do anything for stacks, and a_i is the action to initialize both of the stacks; T is a set of transitions in $ePDA$; an element of T is expressed as 4-tuple (x, e, a, y) elements of which represent a from-state, an event, an action for stacks and a to-state respectively, and it satisfies $x \in (S \cup \{i\}) \land e \in E \land a \in A \land y \in (S \cup F) \land \neg (x=i \land y \in F)$.

The implicit behavior rule of $ePDA$ (i.e., software with UR functions) is as follows. When $ePDA$ accepts an element of E, it performs a specified action, and then goes to a next state based on a current state and the accepted element. Additionally, H_r is initialized if H_r isn't empty. When $ePDA$ accepts u, it pushes a current state onto H_r, and then returns to a previous state that is popped from H_u. *ePDA* shall not revisit i by accepting u, and shall not accept u in an element of F. When *ePDA* accepts r, it pushes a current state onto H_u, and then returns to a state that is popped from H_r.

When *ePDA* has the above rule, test engineers aren't required to explicitly define the transitions triggered by u and r. Since the rule can be implemented in a testcase generation tool, it reduces their work load.

Definition 2: A set of measuring objects for a state UR coverage criterion in *ePDA*

$$O_S = \{\, s \mid \exists t\, (t \in T \wedge t[3]=a_p \wedge (t[1]=s \wedge t[4] \notin F \vee t[4]=s \wedge t[1] \neq i)) \,\},$$

where $t[x]$ represents the xth element of 4-tuple t. A measuring object is an item that should be tested to increase coverage in a specified coverage criterion. In a state UR coverage criterion, testcases are designed so that all the states of having transitions with a_p are revisited by u or r at least once. Test engineers can confirm that each state is correctly reproduced by UR functions at least once. This criterion doesn't give any combinations among transitions, then it would be useful only as a preliminary step.

Definition 3: A set of measuring objects for an N-switch UR coverage criterion ($N \geq 0$) in *ePDA*

$$\begin{aligned} O_N = \{\, (t_1, t_2, \cdots, t_n) \mid\ & n=N+1 \wedge \\ & \forall j\, (1 \leq j \leq n \rightarrow t_j \in (T \cup T_u \cup T_r)) \wedge \\ & \exists k\, (1 \leq k \leq n \rightarrow t_k \in (T_u \cup T_r)) \wedge \\ & \forall l\, (1 \leq l \leq n-1 \rightarrow t_l[4]=t_{l+1}[1]) \,\}, \end{aligned}$$

where T_u and T_r are sets of implicit transitions triggered by u and r respectively; the to-states of these transitions are dynamically determined by the contents of stacks. In an N-switch UR coverage criterion, testcases are designed so that all the sequences of successive transitions of length N+1 (in each sequence there are always one or more elements of T_u and/or T_r) are performed at least once. This is the extension of Chow's N-switch criterion [2]. Test engineers can confirm that all the sequences of successive transitions of length N+1 are correctly performed by UR functions. This criterion includes the state UR coverage criterion, and would be effective against the faults described in section II.

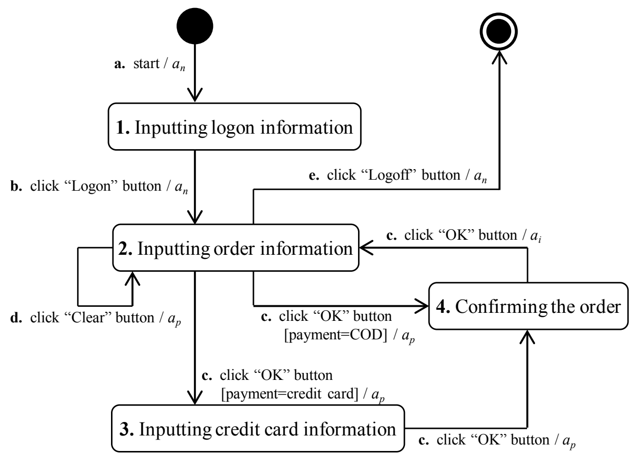
Definition 4: A general formula for coverage

$$C(O') = \frac{|O'|}{|O|},$$

where $|O|$ represents the number of elements of O; O is a set of measuring objects, and O' is a set of elements of O that have been tested. When $O = O_S$, this formula is for state UR coverage, and when $O = O_N$, this is for N-switch UR coverage.

B. Simple Example

We revised the PDA described as an example in [1] so as to make it fit for this technique, which is shown in Fig.1. The measuring objects of a state UR coverage criterion are states 2, 3 and 4; and the measuring objects of a 0-switch UR



a. start / a_n
1. Inputting logon information
b. click "Logon" button / a_n
e. click "Logoff" button / a_n
2. Inputting order information
c. click "OK" button / a_i
d. click "Clear" button / a_p
c. click "OK" button [payment=COD] / a_p
4. Confirming the order
c. click "OK" button [payment=credit card] / a_p
3. Inputting credit card information
c. click "OK" button / a_p

Figure 1. Example of an ePDA (extended pushdown automaton) of a simple internet shopping system.

coverage criterion are transition sequences 2u2, 2r2, 2r3, 2r4, 3u2, 3r4, 4u2 and 4u3. As described in definition 1, u and r represent an undo event and a redo event respectively. For example, the state UR coverage of a testcase a1b2c4u2r4c2e is about 67% (2/3) because states 2 and 4 are revisited by performing u or r. On the other hand, 0-switch UR coverage is 25% (2/8) because 4u2 and 2r4 are performed.

IV. Conclusion and Future Work

In section II, user operations were classified into three types; i.e., (i) ones UR functions are available to, (ii) ones UR functions aren't available to, and (iii) ones that trigger the initialization of stacks. The type of a user operation in an ePDA depends on the explicit action of a transition; (i) are transitions with a_p, (ii) are transitions with a_n, and (iii) are transitions with a_i. The definition of an ePDA in this paper is the fundamentals and can be extended; for instance, it may be required to introduce special events that perform undo/redo of multiple user operations. Additionally this paper showed the definitions of UR coverage criteria and examined their testcases. The UR coverage criteria are the extension of existing criteria, and can be extended further based on other existing criteria.

The advantage of this technique is in that the implicit behavior rule of an ePDA helps test engineers to use the ePDA easily. We plan to develop UR coverage criteria, and then evaluate the effectiveness through trial applications to actual software testing.

References

[1] T. Takagi and Z. Furukawa, "GB Coverage Criteria: The Measurement for Testing a "Go Back" Function Based on a Pushdown Automaton", Proceedings of 19th International Symposium on Software Reliability Engineering, CD-ROM, 2008.

[2] T.S. Chow, "Testing Software Design Modeled by Finite-State Machines", IEEE Transactions on Software Engineering, Vol.SE-4, No.3, pp.178-187, 1978.