

Optimized Test Case Generation Based on Operational Profiles with Fault-Proneness Information

Tomohiko Takagi and Mutlu Beyazıt

Abstract In this paper, a novel formal model called an OPFPI (operational profile with fault-proneness information) and a novel algorithm to generate optimized test cases from the OPFPI are proposed in order to effectively improve software reliability or gain a perspective of software reliability in a limited span of time for testing. The OPFPI includes the feasibility problem due to the use of guards (conditions to make specific state transitions feasible); therefore, ant colony optimization is employed in the algorithm to generate test cases that cover frequent and fault-prone state transitions as comprehensively as possible. A test tool that implements the OPFPI and executes the optimized test case generation has been developed, and it has been applied to a non-trivial system. The obtained results indicate that significant improvement of test cases can be achieved in a short time.

Keywords Model-based software testing · Operational profile · Test case generation · Ant colony optimization

1 Introduction

Testing is an important process to detect faults in software before shipping, and it can be performed systematically by using different software testing techniques. The testing techniques, however, cannot assure that there are no faults in the software [1], but software reliability [11] can be improved by correcting the detected faults. Software reliability is concerned with the operation of the software in expected

T. Takagi (✉)

Faculty of Engineering, Kagawa University, 2217-20 Hayashi-cho,
Takamatsu-shi, Kagawa 761-0396, Japan
e-mail: takagi@eng.kagawa-u.ac.jp

M. Beyazıt

Faculty of Engineering, Department of Computer Engineering,
Yaşar University, Üniversite Caddesi, No: 37–39, Ağacli Yol, Bornova, Izmir, Turkey
e-mail: mutlu.beyazit@yasar.edu.tr

environments without any faults; it is determined by *usage distributions* (i.e., how software is used by users or other systems) and *fault distributions* (i.e., how software includes latent faults).

OPBT (*operational profile-based testing*) [9, 14] is a testing technique focusing on usage distributions; it is sort of a combination of MBT (model-based testing), random testing, and state-transition-based testing. The operational profile is a probabilistic state machine created by a test engineer; it consists of a state machine and probability distributions. The former represents the expected behavior of SUT (software under test), and it is constructed based on software specifications. The latter represents the characteristics of the expected usage of the SUT, and it is derived based on the survey of operational environments. The operational profile is used to randomly generate test cases in the form of state-transition sequences starting from an initial state. The test cases statistically reflect the characteristics of the expected usage in operational environments, and, therefore, OPBT is effective in detecting faults that relate to frequent usage in operational environments and have serious impacts on software reliability. Thus, OPBT is generally not suitable for detecting the faults that relate to infrequent usage.

FABT (*fault analysis-based testing*) [5, 13], on the other hand, focuses on fault distributions. In FABT, the complexity of software specification or source code is evaluated by software metrics, or bug reports stored in a bug tracking system are analyzed by mathematical techniques. Then, test cases are created to test fault-prone parts that have high complexity or relate to a large number of bug reports. FABT is effective in devoting the test effort to the parts that have the likelihood of including a number of latent faults, but it does not ensure that test cases detect faults that have serious impacts on software reliability.

Test engineers require a testing technique that has the strength of both OPBT and FABT in order to effectively improve the software reliability or gain an insight into it in a limited span of time for testing. To our knowledge, such a testing technique has not been established except for [3], which, however, employs more of an analytical approach and uses simulations to support the analysis; it does not employ any models or propose any test case generation algorithms.

In this paper, we propose a new formal model called an *OPFPI* (*operational profile with fault-proneness information*), and a novel algorithm to generate optimized test cases from the OPFPI. The generated test cases have to satisfy the various constraints of the state machine and the test tactics to cover both the transitions that are frequently executed in operational environments and the fault-prone transitions that have high complexity or relate to a large number of bug reports. To solve this constraint problem, we construct the algorithm using ACO (ant colony optimization) [12], which is a metaheuristic suitable for applying to test case generation in MBT [4, 8].

The rest of this paper is organized as follows. Section 2 discusses related work. In Sects. 3 and 4, an OPFPI model and an optimized test case generation algorithm based on OPFPI are proposed, respectively. Section 5 evaluates the proposed technique by discussing its effectiveness and limitations over a non-trivial system. Finally, Section 6 concludes the paper and outlines future work.

2 Related Work

OPBT was proposed as a software testing technique mainly for the rapid improvement of software reliability. There are several studies reporting its effectiveness. For example, Hartmann et al. reduce the test efforts for medical systems by introducing OPBT [7]. One of the challenges in OPBT is to derive the usage distributions, and they show that the usage distributions can be systematically derived from software execution histories captured by a test tool (a capture/replay tool). Chruscielski et al. construct operational profiles for software of aircrafts, and show that it is effective for improving the test process and the software reliability [2]. However, Beizer points out that software functions of frequent use have fewer faults and, therefore, test efforts should be concentrated on the ones of infrequent use rather than of frequent use [1]. In general, several factors can affect the fault detection ability of a software testing technique. Also, relationships between usage and fault distributions are not always straightforward, which is a reason why we focus on not only usage distributions but also fault distributions.

In OPBT, there is a formal model called a test model [14], which is similar to the OPFPI model proposed in this paper. The test model is a probabilistic state machine that provides an overview of test results and the way to evaluate test stopping criteria. The state machine is constructed based on the actual behavior of the SUT and the probability distributions derived from the results of test case execution. If a fault is detected by test case execution, a special state transition that represents the occurrence of the fault and its probability are added to the test model. When a sufficient number of test cases are executed and a certain number of faults are revealed, the difference between the probability distributions of the operational profile and the test model becomes small. This means that test engineers can stop the OPBT, because the software reliability of the SUT has reached a sufficient level. Unlike OPFPI, the test model of OPBT does not operate on pure usage distributions and detailed fault distributions. Therefore, it is relatively less suitable for test case generation.

It is a well-known fact that there are relationships between software metrics and software quality, and a practical test tactic is to concentrate test efforts on the parts that possess low quality from the perspective of software metrics. For example, Eski et al. propose a technique to predict parts of low quality by combining several object-oriented software metrics [5]. Whittaker et al. show the way to predict fault-prone parts by using the frequencies and time of bug fixes [13]. However, software metrics and fault analysis have not been incorporated into MBT that is effective in systematically testing large and complex software.

Most techniques of MBT are aimed at covering the model as comprehensive as possible, and some recent researches introduce metaheuristics in order to generate optimized test cases, that is, test cases that satisfy the constraints included in the model and exercise all elements of the model by using a smaller set of test inputs. For example, it is difficult to deterministically generate optimized test cases from a state machine that includes guards (conditions to make specific transitions feasible), which leads to a feasibility problem. GAs (genetic algorithms) can be used to solve

the feasibility problem and generate optimized test cases that cover the transitions of the state machine [4, 8]. GAs are well-known metaheuristics that imitate the evolution of life to find approximate solutions. Srivastava et al. show that ACO works better than the GA on the feasibility problem [12]. ACO is a metaheuristic imitating the ants searching for food. Our technique is not aimed at covering the model, but it must address the feasibility problem since an OPFPI includes guards. Therefore, we employ ACO in our testing technique.

3 Operational Profile with Fault-Proneness Information

In this section, we propose an OPFPI, which is a novel formal model that consists of a conventional operational profile (a state machine and usage distributions) and fault distributions. An OPFPI is based on a state machine SM which is defined as follows.

Definition 1 $SM = \{S, E, s_0, F, \delta, G\}$, where:

- S is a finite set of states.
- E is a finite set of events corresponding to the stimuli from users and other systems, etc.
- s_0 is an initial state where $s_0 \in S$.
- F is a finite set of final states where $F \subseteq S$.
- δ is a state transition function where $\delta : S \times E \times \{G \cup \emptyset\} \rightarrow S$.
- G is a finite set of guards to express conditions that should be satisfied before specific state transitions can be performed.

An OPFPI is constructed by extending a state machine. The definition is given as follows:

Definition 2 $OPFPI = \{S, E, s_0, F, \delta, G, D_u, D_f\}$, where:

- D_u expresses usage distributions, that is, a finite set of occurrence probabilities of event e ($e \in E$) in state s ($s \in S$) in expected operational environments.
- D_f expresses fault distributions of the SUT, that is, a finite set of fault-proneness values of event e ($e \in E$) in state s ($s \in S$).

An example OPFPI is shown in Fig. 1. This OPFPI has three states and four events. In this paper, states and events are identified by numbers and letters, respectively. When the current state is state 1, events a , b , c and d can occur with probabilities of 25, 55, 10 and 10%, and bring the current state to states 2, 1, 3 and 1, respectively. The fault-proneness values of these transitions, expressed as $(1, a)$, $(1, b)$, $(1, c)$ and $(1, d)$, are 10, 60, 5 and 10, respectively. Note that all the fault-proneness values in Fig. 1 are normalized to range from 0 to 100. $(1, b)$ has a high occurrence probability and a high fault-proneness value, and, therefore, test efforts should be intensively devoted to it in state 1. On the other hand, when the current state is state 2, the

event state		a	b	c	d
		event a	event b	event c	event d
1	state 1 (initial state)	25% / 10 2	55% / 60 1	10% / 5 3	10% / 10 1
2	state 2	40% / 5 1[guard 1], 2[guard 2]	10% / 100 3	50% / 15 1[guard 2]	N/A
3	state 3	N/A	N/A	N/A	100% / 0 1

Fig. 1 Simple example of an OPFPI

evaluation of guards is required in order to determine feasible events. For example, when only guard 1 is satisfied in state 2, events *a* and *b* can occur with probabilities of 80 and 20%, and bring the current state to states 1 and 3, respectively. Note that event *c* cannot occur and becomes N/A, since guard 2 is not satisfied in this case. The fault-proneness values of (2, *a*) and (2, *b*) are 5 and 100, respectively. (2, *b*) has a low occurrence probability but an extremely high fault-proneness value, and, therefore, it should be focused in testing of state 2.

An OPFPI can be constructed by the following steps.

(Step 1) Test engineers construct a state machine that represents the expected behavior of SUT using the software specifications. The abstraction level of the state machine is determined based on the purpose of testing.

(Step 2) Test engineers can derive usage distributions in the expected operational environments of the SUT by using the following investigation methods:

- The execution histories are collected from other software that is similar to the SUT (such as a prior version or prototype of the SUT) by test/debug/security tools, and the state machine constructed in Step 1 is traced using the collected information in order to calculate the occurrence probabilities of events in states [7].
- The questionnaire or the expectation of engineers and users determine the occurrence probabilities [2].
- The usage distributions of other operational profiles are reused by mathematical programming [10].

(Step 3) Test engineers can derive fault distributions by using software metrics and fault prediction techniques, such as LOC (lines of code), cyclomatic complexity, Halstead's software science, CK metrics, MOOD metrics, and the fault prediction based on the frequencies and time of bug fixes [1, 5, 13]. The software metrics are evaluated by existing software development tools, and, also, the fault prediction can be automatically performed based on bug reports stored in existing bug tracking systems. One or more software metrics and fault prediction techniques are selected based on the domain of the SUT and the purpose of testing to obtain fault-proneness values of events in states. When multiple software metrics and

fault prediction techniques are selected, the results of them are normalized and combined [5]. For example, when metrics m_1 taking values in $[0, \infty)$ and m_2 taking values in $[0, 1]$ are selected, they can be normalized to take values in $[0, 100]$, and then combined by using the normalized values of m_1 and m_2 .

4 Test Case Generation Using Ant Colony Optimization

In this section, we propose a novel algorithm for generating optimized test cases from a given OPFPI. ACO is introduced to effectively generate optimized test cases that satisfy the various constraints of the state machine and the following test tactics.

- Cover the transitions that will be frequently executed in operational environments, which are revealed by usage distributions in the OPFPI.
- Cover the fault-prone transitions that have high complexity or relate to a large number of bug reports, which are revealed by fault distributions in the OPFPI.

The basic idea of the algorithm is that agents (i.e., ants) perform reiteration search of paths (i.e., test cases) on the OPFPI based on usage distributions, fault distributions, and pheromones. The agents release and add the pheromones onto traversed paths in order to improve their future search. The details of the algorithm are as follows.

(Step 1) A ($A \geq 1$) agents whose movable distance (i.e., the number of transitions that a test case consists of) is D are created, and then the initial pheromone levels of all transitions of the OPFPI are set at τ_0 ($\tau_0 \geq 0.0$).

(Step 2) All the agents perform the behavior specified by Step 2.1, 2.2, and 2.3.

(Step 2.1) An agent is placed on the initial state of the OPFPI.

(Step 2.2) The agent searches a path by repeating the random selection of a feasible event. The probability that an agent a traverse a transition by occurrence of an event e in a state s in time t (the number of execution of Step 2) is given as follows.

$$p_{se}^a(t) = \begin{cases} \frac{[\tau_{se}(t)] \cdot [\eta_{se}]^\alpha}{\sum_{x \in E_a(s)} [\tau_{sx}(t)] \cdot [\eta_{sx}]^\alpha}, & \text{if } e \in E_a(s) \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where $E_a(s)$ expresses a set of events that an agent a can select in a state s , that is, events that are not N/A and satisfy their guard conditions in a state s ; $\tau_{se}(t)$ expresses a pheromone level of an event e in a state s in time t ; η_{se} expresses a heuristic value of an event e in a state s ; and α is a parameter to control the weighting between the pheromone level and the heuristic value. Also, η_{se} is calculated as follows.

$$\eta_{se} = \beta \cdot \frac{u_{se}}{\sum_{x \in E(s)} u_{sx}} + (1 - \beta) \cdot \frac{f_{se}}{\sum_{x \in E(s)} f_{sx}} \quad (2)$$

where $E(s)$ expresses a set of events that can occur in a state s , that is, events that are not N/A in a state s ; u_{se} expresses occurrence probability of an event e in a state s in operational environments; f_{se} expresses a fault-proneness level of an event e in a state s ; and β ($0.0 \leq \beta \leq 1.0$) is a parameter to control the weighting between usage distributions and fault distributions.

(Step 2.3) The agent completes searching a path when it reaches a final state of the OPFPI or the length of the searched path reaches the movable distance D .

(Step 3) When $t = N$ ($N \geq 1$), the best path among all the paths that have been searched by the agents is selected as a solution (an approximate optimal solution), and the algorithm terminates. Evaluation of a path P is performed as

$$E_P = \sum_{(s,e) \in P} \eta_{se} \quad (3)$$

where η_{se} expresses a heuristic value of an event e in a state s . Note that the same (s, e) can appear in P repeatedly, but its heuristic value is not added repeatedly. P with a larger value of E_P is better for achieving the test tactics mentioned before.

(Step 4) For the next execution of Step 2, the pheromone levels of all the transitions are updated as follows.

$$\tau_{se}(t+1) = (1 - \rho) \cdot \tau_{se}(t) + \sum_{a=1}^A \Delta \tau_{se}^a \quad (4)$$

where ρ ($0.0 < \rho < 1.0$) expresses the evaporation rate of pheromones between t and $t+1$. Furthermore, $\Delta \tau_{se}^a$ expresses pheromones that an agent a newly adds to (s, e) , and it is calculated as

$$\Delta \tau_{se}^a = \begin{cases} E_{P_a}, & \text{if } (s, e) \in P_a \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

where P_a expresses a path that have been searched by an agent a . After the update, the algorithm goes back to Step 2.

The parameters used in the above algorithm are A (the number of agents), D (the movable distance of agents), N (the number of execution of Step 2), τ_0 (the initial pheromone level), α (the parameter to control the weighting between the pheromone level and the heuristic value), β (the parameter to control the weighting between usage distributions and fault distributions), and ρ (the evaporation rate of pheromones). In practice, the values for these parameters are determined by trial-and-error. Thus, test engineers can generate test cases suitable for the domain of SUT and the purpose of testing by using this algorithm with properly selected parameter values.

5 Evaluation

In order to evaluate our technique, we developed a test tool that implements the OPFPI model and performs the optimized test case generation proposed in Sects. 3 and 4. A non-trivial system is used to show an overview of the application, to obtain results and to evaluate the proposed technique based on these results.

The system is software that controls a vending machine, and its requirements are shown in Fig. 2. We created an OPFPI for the software based on the requirements. Also, usage and fault distributions are assumed and given in Fig. 3. The constructed OPFPI has 16 states, 8 events, 82 transitions (i.e., the cells in Fig. 3 that do not contain N/A), and several guards. The model is non-trivial; it is sufficiently large to experiment and evaluate the testing method proposed in the paper. Note that Fig. 3 can also be expressed as a state transition diagram.

Optimized test cases are generated from the constructed OPFPI automatically by using the test tool with parameter values of $A = 1$, $D = 300$, $N = 500$, $\tau_0 = 0.1$, $\alpha = 1.0$, $\beta = 0.5$ and $\rho = 0.05$, which are determined by trial-and-error. Fig. 4 outlines a graph, where the horizontal axis shows t (time, i.e., the number of executions of Step 2) and the vertical axis shows E_P (the evaluation of a test case P —See Eq. (3)).

In Fig. 4, P in the thick line corresponds to the best test case among all the generated test cases; that is, the thick line demonstrates the growth of the best test case. On the other hand, P in the thin line corresponds to a test case that is newly generated for each t . The thick line reveals that E_P for $t = 1$ (i.e., in the case that the ACO is not executed) and for $t = 500$ (i.e., in the case that the ACO is executed) are about 7.3 points and 12.2 points respectively. It means that this optimized test case generation has achieved an improvement of about 4.9 points (about 68%).

Our optimized test case generation is probabilistic; therefore, the generation with the above parameter settings was executed 100 times in order to obtain more reliable results. As a result, the average E_P of the best test case for $t = 500$ (i.e., final solution) is about 11.2 points, and its MAD (mean absolute deviation) is about 0.98. The MAD is small enough to conclude that the obtained results are stable; that is,

The VM (vending machine) sells coffees at 100 JPY (Japanese yen) apiece, and teas at 120 JPY apiece. It accepts only 100 JPY coins and 10 JPY coins. If the total of coins dropped into the VM reaches the price of a coffee or a tea, the button to buy a coffee or a tea becomes available respectively. If the available button is pushed, the VM provides a coffee or a tea, and the change (10 JPY coins). If a return lever is pressed down, the VM returns the dropped coins.

The VM indicates the lack of coffees, teas, and 10 JPY coins for change. If coffees or teas are sold out, the corresponding button does not become available even if sufficient money to buy is dropped into the VM. Also, if sufficient money requiring the change is dropped into the VM and the VM does not have sufficient 10 JPY coins for the change, the button to buy a tea does not become available. When no money is dropped into the VM, coffees, teas, and 10 JPY coins can be supplied by a serviceman.

Fig. 2 Requirements of software to control a vending machine

state	event	a	b	c	d	e	f	g	h
		drop a 10JPY coin	drop a 100JPY coin	return	buy a coffee	buy a tea	supply change	supply coffees	supply teas
1	b _{coffee} off, b _{tea} off s _{coffee} off, s _{tea} off changeoff	25% / 20 1 [drop<100JPY], 2 [drop=100JPY]	55% / 20 2 [drop<120JPY], 3 [drop=120JPY]	5% / 0 1 [drop<0JPY]	N/A	N/A	5% / 5 1 [drop<0JPY]	5% / 5 1 [drop<0JPY]	5% / 5 1 [drop<0JPY]
2	b _{coffee} on, b _{tea} off s _{coffee} off, s _{tea} off changeoff	30% / 20 2 [drop<120JPY], 3 [drop=120JPY]	30% / 10 3	5% / 5 1	35% / 50 1 [g _{coffee}], 4 [g _{coffee}]	N/A	N/A	N/A	N/A
3	b _{coffee} off, b _{tea} on s _{coffee} on, s _{tea} off changeoff	3% / 5 3	2% / 5 3	5% / 5 1	20% / 55 1 [g _{coffee}], 4 [g _{coffee}]	70% / 85 1 [g _{tea} & g _{change}], 6 [g _{tea} & g _{change}], 9 [g _{tea} & g _{change}], 14 [g _{tea} & g _{change}]	N/A	N/A	N/A
4	b _{coffee} off, b _{tea} off s _{coffee} on, s _{tea} off changeoff	30% / 20 4 [drop<120JPY], 5 [drop=120JPY]	45% / 20 4 [drop<120JPY], 5 [drop=120JPY]	5% / 0 4 [drop<0JPY]	N/A	N/A	5% / 5 4 [drop<0JPY]	10% / 10 1 [drop<0JPY]	5% / 5 4 [drop<0JPY]
5	b _{coffee} off, b _{tea} on s _{coffee} on, s _{tea} on changeoff	3% / 5 5	2% / 5 5	5% / 5 4	N/A	90% / 85 4 [g _{tea} & g _{change}], 8 [g _{tea} & g _{change}], 12 [g _{tea} & g _{change}], 16 [g _{tea} & g _{change}]	N/A	N/A	N/A
6	b _{coffee} off, b _{tea} off s _{coffee} on, s _{tea} on changeoff	20% / 20 6 [drop<100JPY], 7 [drop=100JPY]	55% / 15 7 [drop<100JPY]	5% / 0 6 [drop<0JPY]	N/A	N/A	5% / 5 6 [drop<0JPY]	5% / 5 6 [drop<0JPY]	10% / 10 1 [drop<0JPY]
7	b _{coffee} on, b _{tea} off s _{coffee} off, s _{tea} on changeoff	3% / 5 7	2% / 5 7	5% / 5 6	90% / 50 6 [g _{coffee}], 8 [g _{coffee}]	N/A	N/A	N/A	N/A
8	b _{coffee} off, b _{tea} off s _{coffee} on, s _{tea} on changeoff	4% / 5 8	6% / 5 8	10% / 0 8 [drop<0JPY]	N/A	N/A	10% / 5 8 [drop<0JPY]	35% / 10 6 [drop<0JPY]	35% / 10 4 [drop<0JPY]
9	b _{coffee} off, b _{tea} off s _{coffee} on, s _{tea} off changeon	40% / 20 9 [drop<100JPY], 10 [drop=100JPY]	20% / 30 10 [drop<120JPY & 100JPY coins=2], 11 [drop=120JPY & 100JPY coins=2]	10% / 0 9 [drop<0JPY]	N/A	N/A	20% / 10 1 [drop<0JPY]	5% / 5 9 [drop<0JPY]	5% / 5 9 [drop<0JPY]
10	b _{coffee} on, b _{tea} off s _{coffee} off, s _{tea} off changeon	30% / 30 10 [drop<120JPY & 100JPY coins=2], 11 [drop=120JPY & 100JPY coins=2]	5% / 20 10 [100JPY coins=2], 11 [100JPY coins=2]	10% / 5 9	55% / 80 1 [g _{coffee} & g _{change}], 4 [g _{coffee} & g _{change}], 9 [g _{coffee} & g _{change}], 12 [g _{coffee} & g _{change}]	N/A	N/A	N/A	N/A
11	b _{coffee} on, b _{tea} on s _{coffee} off, s _{tea} off changeon	3% / 5 11	2% / 5 11	10% / 5 9	18% / 85 1 [g _{coffee} & g _{change}], 4 [g _{coffee} & g _{change}], 9 [g _{coffee} & g _{change}], 12 [g _{coffee} & g _{change}]	67% / 100 1 [g _{tea} & g _{change}], 6 [g _{tea} & g _{change}], 9 [g _{tea} & g _{change}], 14 [g _{tea} & g _{change}]	N/A	N/A	N/A
12	b _{coffee} off, b _{tea} off s _{coffee} on, s _{tea} off changeon	40% / 30 12 [drop<120JPY & 100JPY coins=2], 13 [drop=120JPY & 100JPY coins=2]	25% / 30 12 [drop<120JPY & 100JPY coins=2], 13 [drop=120JPY & 100JPY coins=2]	10% / 0 12 [drop<0JPY]	N/A	N/A	10% / 10 4 [drop<0JPY]	10% / 10 9 [drop<0JPY]	5% / 5 12 [drop<0JPY]
13	b _{coffee} off, b _{tea} on s _{coffee} on, s _{tea} off changeon	3% / 5 13	2% / 5 13	10% / 5 12	N/A	85% / 85 4 [g _{tea} & g _{change}], 8 [g _{tea} & g _{change}], 12 [g _{tea} & g _{change}], 16 [g _{tea} & g _{change}]	N/A	N/A	N/A
14	b _{coffee} off, b _{tea} off s _{coffee} off, s _{tea} on changeon	20% / 20 14 [drop<100JPY], 15 [drop=100JPY]	45% / 10 15	10% / 0 14 [drop<0JPY]	N/A	N/A	10% / 10 6 [drop<0JPY]	5% / 5 14 [drop<0JPY]	10% / 10 9 [drop<0JPY]
15	b _{coffee} on, b _{tea} off s _{coffee} off, s _{tea} on changeon	3% / 5 15	2% / 5 15	10% / 5 14	85% / 80 6 [g _{coffee} & g _{change}], 8 [g _{coffee} & g _{change}], 15 [g _{coffee} & g _{change}], 16 [g _{coffee} & g _{change}]	N/A	N/A	N/A	N/A
16	b _{coffee} on, b _{tea} off s _{coffee} on, s _{tea} on changeon	2% / 5 16	3% / 5 16	10% / 0 16 [drop<0JPY]	N/A	N/A	25% / 10 8 [drop<0JPY]	30% / 10 14 [drop<0JPY]	30% / 10 12 [drop<0JPY]

- 1 In the names of states, the label *b_{coffee}* and *b_{tea}* express the on/off of the lights that indicate whether the buttons to buy a coffee and a tea are available or not, respectively. Also, the label *s_{coffee}* and *s_{tea}* express the on/off of the lights that indicate whether coffees and teas are sold out or not, respectively. The label *change* expresses the on/off of the light that indicates whether there are insufficient 10 JPY coins for change or not.
- 2 In guards, the variable *drop* expresses the current total of dropped coins. Also, the variable *10JPYcoins* expresses the current number of dropped 10 JPY coins. The variable *g_{coffee}* and *g_{tea}* indicate whether coffees and teas are sold out or not after the event occurrence, respectively. The variable *g_{change}* indicates whether there are insufficient 10 JPY coins for change or not after the event occurrence.

Fig. 3 OPFPI of software to control a vending machine

different runs yield similar values. Also, the average improvement is about 42 %, showing that the algorithm achieves significant improvements.

Depending on the selected parameters, evaluation value may converge in different fashions. Figure 4 shows that, in our case, it converges very fast. Therefore, there is a little change for $t > 10$ and no change for $t > 240$. This also means that sufficiently good test cases can be found without performing too many iterations of Step 2 (ACO).

The average time to complete the optimized test case generation is about 21 seconds on a laptop with 2.5GHz CPU and 2G Bytes RAM; it is acceptable for practical applications even if it needs to be executed many times by a trial-and-error

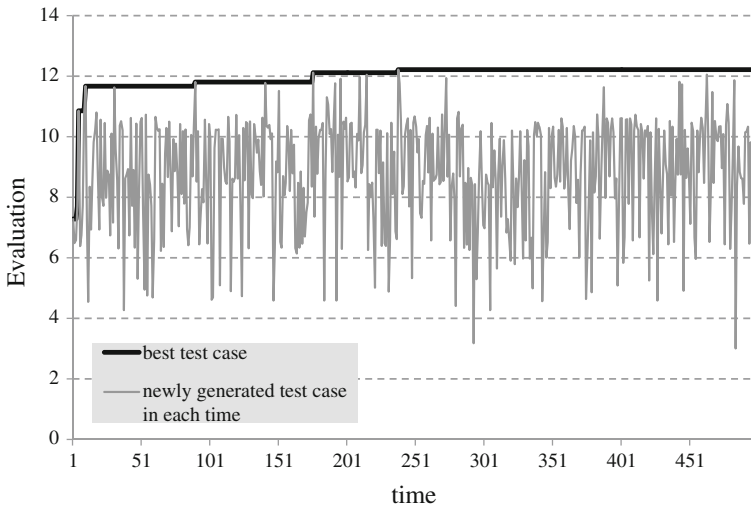


Fig. 4 Result of optimized test case generation

method. However, construction of an OPFPI model may be a relatively heavy task for test engineers. The quality of the OPFPI and the amount of time to construct it depend on test engineers' skill. When the SUT is developed by MDD (model-driven development) [6], the quality and time might improve due to the following reasons.

- Test engineers can get more familiar by gathering experiences while constructing models in requirements analysis and design processes. Also, they can use these models in order to efficiently construct a state machine for an OPFPI.
- Usage and fault distributions can be efficiently derived due to the traceability among models, source codes, and bug reports.

6 Conclusion

In this paper, a novel formal model called an OPFPI (operational profile with fault-proneness information) and a novel algorithm to generate optimized test cases from the OPFPI have been proposed in order to establish a testing technique that has a strength of both OPBT (operational profile-based testing) and FABT (fault analysis-based testing). The OPFPI has both usage distributions (i.e., information about how software will be used by users or other systems) and fault distributions (i.e., information about how software will include latent faults). It provides a good basis for generating test cases to effectively improve software reliability or gain a perspective of software reliability in a limited span of time for testing. The OPFPI includes the feasibility problem due to the use of guards (conditions to make specific transitions feasible); therefore, ACO (ant colony optimization) is employed in the algorithm to generate test cases that cover frequent and fault-prone transitions as comprehensively

as possible. A test tool that implements the OPFPI model and executes the optimized test case generation has been developed, and the technique has been applied to a non-trivial system. The obtained results indicate that significant improvement of test cases can be achieved in a short time. However, constructing an OPFPI of good quality with a small amount of effort is a challenge, and MDD might be a key to tackle this problem.

In future study, we intend to consider a technique that enables inexperienced test engineers to effectively construct an OPFPI by using MDD tools, and we plan to extend the OPFPI and the test case generation algorithm in order to achieve other test tactics and perform further evaluations. Furthermore, the approach can also be extended to support traits such as hierarchy, modularity and concurrency in an OPFPI so that the efficiency can be increased while working with larger real-world software.

Acknowledgments This work was supported by JSPS KAKENHI Grant Number 23700038.

References

1. Beizer, B.: *Software Testing Techniques*, 2nd edn. Van Nostrand Reinhold, New York (1990)
2. Chruscielski, K., Tian, J.: An operational profile for the cartridge support software. In: *Proceedings of 8th International Symposium on Software Reliability Engineering*, pp. 203–212 (1997)
3. Cotroneo, D., Pietrantuono, R., Russo, S.: Combining operational and debug testing for improving reliability. *IEEE Trans. Reliab.* **62**(2), 408–423 (2013)
4. Doungsa-ard, C., Dahal, K., Hossain, A., Suwannasart, T.: Test data generation from UML state machine diagrams using GAs. In: *Proceedings of International Conference on Software Engineering Advances*, p. 47 (2007)
5. Eski, S., Buzluca, F.: An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. In: *Proceedings of International Conference on Software Testing, Verification and Validation Workshops*, pp. 566–571 (2011)
6. Frankel, D.S.: *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, New York (2003)
7. Hartmann, H., Bokkerink, J., Ronteltap, V.: How to reduce your test process with 30 %—the application of operational profiles at Philips medical systems. In: *Supplementary Proceedings of 17th International Symposium on Software Reliability Engineering*. CD-ROM (2006)
8. Kalaji, A., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine (EFSM). In: *Proceedings of International Conference on Software Testing Verification and Validation*, pp. 230–239 (2009)
9. Musa, J.D.: The operational profile. *Reliab. Maintenance Complex Syst. NATO ASI Ser. F Comput. Syst. Sci.* **154**, 333–344 (1996)
10. Poore, J.H., Walton, G.H., Whittaker, J.A.: A constraint-based approach to the representation of software usage models. *Inf. Softw. Technol.* **42**(12), 825–833 (2000)
11. Rook, P.: *Software Reliability Handbook*. Elsevier Science, New York (1990)
12. Srivastava, P.R., Baby, K.: Automated software testing using metaheuristic technique based on an ant colony optimization. In: *Proceedings of International Symposium on Electronic System Design*, pp. 235–240 (2010)
13. Whittaker, J.A., Arbon, J., Carollo, J.: *How Google Tests Software*. Addison-Wesley Professional, Westford (2012)
14. Whittaker, J.A., Thomason, M.G.: A Markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.* **20**(10), 812–824 (1994)

Software Engineering Research, Management and
Applications

Lee, R. (Ed.)

2015, XIII, 234 p. 105 illus., Hardcover

ISBN: 978-3-319-11264-0