Autom Softw Eng (2010) 17: 5–31
DOI 10.1007/s10515-009-0056-x

Test input reduction for result inspection to facilitate fault localization

Dan Hao · Tao Xie · Lu Zhang · Xiaoyin Wang · Jiasu Sun · Hong Mei

Received: 18 July 2008 / Accepted: 31 July 2009 / Published online: 19 August 2009

Abstract Testing-based fault-localization (TBFL) approaches often require the availability of high-statement-coverage test suites that sufficiently exercise the areas around the faults. However, in practice, fault localization often starts with a test suite whose quality may not be sufficient to apply TBFL approaches. Recent capture/replay or traditional test-generation tools can be used to acquire a high-statement-coverage test collection (i.e., test inputs only) without expected outputs. But it is expensive or even infeasible for developers to manually inspect the results of so many test inputs. To enable practical application of TBFL approaches, we propose three strategies to reduce the test inputs in an existing test collection for result inspection. These three strategies are based on the execution traces of test runs using the test inputs. With the three strategies, developers can select only a representative subset of the test inputs for result inspection and fault localization. We implemented and applied the three test-input-reduction strategies to a series of benchmarks: the Siemens programs, DC, and

D. Hao · L. Zhang · X. Wang · J. Sun · H. Mei (✉)
Key Laboratory of High Confidence Software Technologies, Ministry of Education, Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, People's Republic of China
e-mail: meih@pku.edu.cn

D. Hao
e-mail: haod@sei.pku.edu.cn

L. Zhang
e-mail: zhanglu@sei.pku.edu.cn

X. Wang
e-mail: wangxy06@sei.pku.edu.cn

J. Sun
e-mail: sjs@sei.pku.edu.cn

T. Xie
Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
e-mail: xie@csc.ncsu.edu

TCC. The experimental results show that our approach can help developers inspect the results of a smaller subset (less than 10%) of test inputs, whose fault-localization effectiveness is close to that of the whole test collection.

Keywords Test suite reduction · Testing · Debugging · Fault localization

1 Introduction

Software debugging is usually an inevitable task in software development and maintenance as developers can hardly write faultless programs in the first place. Due to the tediousness of debugging, various approaches have been proposed to help developers locate faults. Among them, testing-based fault-localization (TBFL) approaches (Agrawal et al. 1995; Hao et al. 2008; Jones and Harrold 2005; Jones et al. 2002; Renieris and Reiss 2003) are quite promising. Typical TBFL approaches use the execution information of many test runs to calculate the suspicion of each statement and rank statements according to their suspicions. Intuitively, the suspicion of a statement mainly comes from its involvement in failed test runs.[1] Therefore, TBFL approaches need both the coverage information and the information whether each test run is passed or failed when calculating the suspicions.

Although TBFL approaches have shown to be effective in locating faults, these approaches assume the availability of a large number of test runs that sufficiently exercise the areas around the faults together with the information whether each test run is passed or failed. For example, TBFL approaches are popularly experimented using test runs (Do et al. 2005) achieving near 100% statement coverage. Typically, these approaches require a high-statement-coverage test suite composed of a large number of test cases, each of which includes both the test input and expected output. With such a test suite, it is convenient to acquire both the coverage information and the information whether each test run is passed or failed. However, this high-statement-coverage test suite with expected outputs is often not realistic in practice. Due to the test oracle problem (Baresi and Young 2001), a developer can usually acquire only a high-statement-coverage test collection[2] containing test inputs without expected outputs using capture/replay techniques (i.e., Saff et al. 2005) and/or test-generation techniques (i.e., Sen et al. 2005). In such a circumstance, the developer has to manually inspect[3] the results of test runs using test inputs from such a test collection to know whether the test runs are passed or failed. As a result, if the developer wants to apply a TBFL approach efficiently, he or she needs to manually inspect the results of many test runs. This requirement may become a burden for applying TBFL approaches in practice.

[1] When the result of one test run is expected, we refer to this test run as passed or successful; otherwise, we refer to this test run as failed. However, in the field of testing, some researchers prefer to use positive/negative test runs rather than passed/failed test runs.

[2] In this paper, a "test collection" represents a collection of test inputs without expected outputs, to be distinguished from a "test suite", which is a collection of test cases, including both test inputs and their expected outputs.

[3] In this paper, we use "check" and "inspect" interchangeably.

To enable practical application of TBFL approaches when sufficient test cases (i.e., test inputs together with their expected outputs) are not available, we are required to reduce the size of a test collection by selecting some test inputs (whose expected outputs are not available) to retain from the test collection so that the developer can apply a TBFL approach by inspecting only the results of these selected test inputs. This problem is a test-reduction problem. Test reduction and test selection are two popular problems in regression testing. Test reduction aims to *remove* the test cases that become redundant in retesting a modified program, whereas test selection aims to *select* test cases that are required in retesting a modified program by focusing on the modified code. As this paper aims to reduce the size of a test collection so that the reduced test collection can provide similar fault-localization effectiveness as the whole test collection, the problem in this paper is more test reduction than test selection. However, as the reduced test collection is constructed by selecting and retaining test inputs from the whole test collection, sometimes we use the word "select" in the rest of the paper although the word "select" in our approach does not refer to test selection in regression testing.

In this paper, we propose an approach to select test inputs (whose expected outputs are not available) from an existing test collection for developers to check or inspect their results before applying a TBFL approach. The major goal of our approach is to reduce the developers' effort on test-result checking. The basic idea is that for a specific fault previously revealed by a test run, our approach selects some test inputs for the developer to check their corresponding results. Thus, the developer can use only the information associated with the test runs for the selected test inputs and the failure-revealing test to locate the fault.

In particular, we propose three strategies to reduce the size of an existing test collection by selecting a subset of test inputs whose corresponding results are to be checked. In this way, less test-result checking can be performed to acquire information for applying TBFL to locate faults. As our approach relies on coverage information for this kind of selection, it is required that the execution traces of all the test inputs in the test collection should be obtained before we apply the reduction strategies. To investigate the effectiveness of our approach, we performed an experimental study on various benchmarks. In the experiment, we compared our approach with the test-suite reduction technique proposed by Harrold et al. (1993). The experimental results show that all the three strategies select a substantially smaller subset of test inputs from the original test collection. Among the three strategies, Strategy 1 and Strategy 2 select as few test inputs as Harrold et al.'s technique. Moreover, the test inputs selected by either of these two strategies are usually more effective in fault localization than the test inputs selected by Harrold et al.'s technique.

This paper makes the following main contributions. (1) We propose three strategies on reducing test inputs for result inspection to save effort of developers from tedious test-result checking when applying TBFL approaches. (2) We experimentally investigate the effectiveness of our three strategies compared with the test-suite reduction technique proposed by Harrold et al. (1993).

In the rest of the paper, for simplicity, we sometimes use a "test" to refer to a "test input".

The rest of the paper is organized as follows. Section 2 proposes our test-reduction approach in detail and illustrates the three strategies. Section 3 discusses some issues

in our work. Section 4 presents an experiment to investigate the effectiveness of our approach. Section 5 discusses related work. Section 6 concludes our paper.

2 Approach

This section presents the details of our approach. We give a brief introduction of our work (Sect. 2.1) and describe the details of three test-reduction strategies (Sect. 2.2).

2.1 Overview

As our approach aims at improving the test-result checking process for a specific fault, the starting point of our approach is one fault-revealing or failed test.[4] The subsequent test-result checking process aims at collecting sufficient information for locating the revealed fault by allowing the developer to check as few test results as possible. Typically, there are already existing (passed) test cases where expected outputs are already equipped or the developer may have checked some test results before encountering the fault-revealing test. That is, encountering a failed test when starting the test-result checking process would be a simplified situation. The actual starting point would be one failed test together with several passed tests. However, we can still assume that there is only one failed test in the beginning for convenience of presenting our approach, because the situation where there are several passed tests besides this failed test can be viewed as a situation occurring during the test-reduction process.

Figure 1 depicts an overview of our approach. The input of our approach is a failed test and a test collection, together with the corresponding execution traces of all these

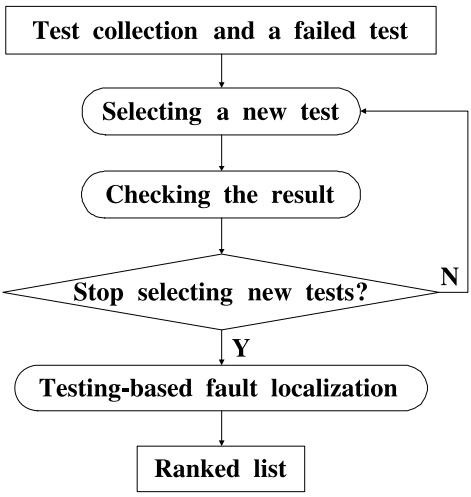

Fig. 1 Overview of our test-reduction approach

[4]The cases of multiple failing tests are also supported, for which any failed test can be used as the starting point of our approach.

tests. Here, an execution trace refers to which statements are executed by a test and the trace is gathered with some instrumentation tool such as "gcov" command of GNU C Compiler. Except for the failed test, it is unknown whether the result of any test in the test collection is passed or failed at this initial stage. To be distinguished from failed tests checked during the subsequent reduction process, the input failed test is referred to as the initial failed test in this paper. With the given test collection and the initial failed test, our approach selects tests from the test collection based on some test-reduction strategies. Specifically, we propose three test-reduction strategies in this paper, which are to be fully presented in the rest of this section. For any strategy, the test-reduction process is as follows. One unchecked test is selected according to the coverage information of all the tests and the information whether each checked test is passed or failed. After the result of the selected test is checked, our approach evaluates whether checking the result of another unchecked test is needed. If so, the test-reduction process continues. Otherwise, the test-reduction process stops. After the process stops, the checked tests and the initial failed test are fed to a TBFL approach, which can produce a ranked list of suspicious statements.

In this paper, we introduce our approach based on statement coverage. However, this approach can be easily generalized to other kinds of coverage criteria such as branch or definition-use pair coverage, which we plan to fully evaluate in our future work.

2.2 Test-reduction strategies

This section presents the three test-reduction strategies, each of which selects a representative set of tests for result inspection before applying fault-localization approaches.

2.2.1 Strategy 1

In TBFL, as the suspicion of each statement is calculated based on the information of how many passed and failed tests cover[5] the statement, statements executed by exactly the same tests are not distinguishable from each other and are assigned the same suspicion. Thus, a straightforward idea to reduce the number of tests fed to TBFL is to find a small subset of the test collection that has the same capability to distinguish the statements as the whole test collection. This idea motivates the development of our first strategy.

Given a target program P, which consists of statements $s_1, s_2, \ldots, s_n$, and a set of tests $T = \{t_1, t_2, \ldots, t_m\}$, two statements of program P are called "T-undistinguishable" if and only if each test in T either executes both statements or neither statement. According to the traditional set theory, this binary relation "T-undistinguishable" is an equivalence relation (Enderton 1977) because this relation is reflexive, symmetric, and transitive.

If R is an equivalence relation, variable x belongs to the domain and the range of R, then the equivalence class (Enderton 1977) of variable x is a set of variables y satisfying the following conditions:

[5] In this paper, we use "cover", "exercise", and "execute" interchangeably.

1. $\langle x, y \rangle \in R$;
2. y belongs to the domain and the range of R.

Note that the equivalence class mentioned in this paper refers to the concept defined in the traditional set theory. The equivalence relation refers to the equivalence relation "T-undistinguishable". Based on this equivalence relation, we have a partition of P. That is, the equivalence relation "T-undistinguishable" partitions the statements of the target program P into several equivalence classes. Each equivalence class consists of statements that are executed by the same tests.

For the given set of tests T, we use $Div(T)$ called division number to represent the number of equivalence classes that the tests in T have partitioned the statements of the target program into. Intuitively, the larger $Div(T)$ is, the more powerful T is to distinguish statements. In the simplest circumstance where T contains just one test, it partitions the statements of program P into two equivalence classes: (1) those covered by this test and (2) those not covered. Then $Div(T)$ is two.

Using the division number, we have our first strategy: when the set of currently selected tests is T_o, Strategy 1 always selects the next test t such that $Div(T_o \cup \{t\})$ can be maximized. If there is more than one test satisfying this condition, Strategy 1 selects any of these tests. When $Div(T_o \cup \{t\})$ is the same as $Div(T_o)$ for any unselected test t, no more tests are needed to be selected. This condition is the stopping condition of Strategy 1.

In the implementation of Strategy 1, we maintain the existing equivalence classes partitioned by currently selected tests (i.e., T_o). Note that T_o initially contains only the initial failed test. When determining which test to be selected next, we can just calculate how many existing equivalence classes can be further partitioned by each unselected test. The test t that partitions the most existing equivalence classes can be ensured to maximize $Div(T_o \cup \{t\})$. After t is selected, we need to update the existing equivalence classes through replacing each equivalence class that can be partitioned by t with its two sub-classes. When no unselected tests can further partition any existing equivalence classes, the reduction process stops.

The equivalence classes based on the definition of "T-undistinguishable" in Strategy 1 is similar to the concept of a "block" given by Baudry et al. (2006). Strategy 1 in this paper can be viewed as the mapping of Baudry et al.'s approach to the problem identified in this paper. However, Baudry et al.'s approach aims at enhancing existing test suites for fault localization, whereas our Strategy 1 aims at reducing test-result inspection efforts for fault localization. Strategy 1 serves as the starting point of our research. We further propose two more strategies considering some factors that possibly influence the number of reduced tests and the fault-localization effectiveness of the reduced tests.

Here we use an example from Jones and Harrold (2002) to illustrate Strategy 1. Table 1 shows the example program "Mid" and its execution traces. The functionality of "Mid" is to find the median of the three input values. There are two faults in Lines 7 and 12, respectively, where value m should be assigned with the value of x rather than that of y. The program is in Column 1 of the table. The initial failed test is t_1. The test collection has four tests, denoted as $T = \{t_2, t_3, t_4, t_5\}$. Execution traces are represented by •. Some statements are distributed in multiple lines. For example, Lines 4 and 5 in Table 1 compose an if-statement. We take them as two statements

Table 1 Program Mid and its execution traces

Mid( ) {	Test Collection				
int x, y, z, m;	t_1	t_2	t_3	t_4	t_5
	2, 3, 1	3, 3, 5	1, 2, 3	5, 5, 5	5, 5, 3
1. read(x, y, z);	•	•	•	•	•
2. $m = z$;	•	•	•	•	•
3. if($y < z$){	•	•	•	•	•
4. if($x < y$)		•	•		
5. $m = y$;			•		
6. else if($x < z$)		•			
7. $m = y$;}		•			
8. else {	•			•	•
9. if($x > y$)	•			•	•
10. $m = y$;					
11. else if($x > z$)	•			•	•
12. $m = y$;}	•				•
13. print(m); }	•	•	•	•	•

because either line is semantically complete and they are usually treated as two statements during debugging. The faulty program consists of statements $s_1, s_2, \ldots, s_{12}$, and s_{13}. The initial failed test is t_1, whose input is 2, 3, and 1, and covers statements s_1, s_2, s_3, s_8, s_9, s_{11}, s_{12}, and s_{13}. For simplicity, we abbreviate the execution of this test as t_1:{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{12}, s_{13}}. Similarly, we have t_2:{s_1, s_2, s_3, s_4, s_6, s_7, s_{13}}, t_3:{s_1, s_2, s_3, s_4, s_5, s_{13}}, t_4:{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{13}}, and t_5:{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{12}, s_{13}}.

The process of applying Strategy 1 to program "Mid" is summarized in Table 2. The first column shows the steps of the test-reduction process. The second column gives the existing T_o, which consists of selected tests. The third column lists the equivalence classes of program "Mid" partitioned by the existing T_o. The fourth column shows the value of $Div(T_o)$. The fifth column lists the value of $Div(T_o \cup \{t_i\})$, where $t_i \notin T_o$.

Initially T_o contains only one test (i.e., t_1). This test partitions the statements of program "Mid" into two equivalence classes $p_1 = \{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{12}, s_{13}\}$ and $p_2 = \{s_4, s_5, s_6, s_7, s_{10}\}$. $Div(T_o) = Div(\{t_1\}) = 2$. If we choose test t_2, t_3, t_4, or t_5, the corresponding division number is 4, 4, 3, or 2, respectively. That is to say, $Div(T_o \cup \{t_2\}) = 4$, $Div(T_o \cup \{t_3\}) = 4$, $Div(T_o \cup \{t_4\}) = 3$, and $Div(T_o \cup \{t_5\}) = 2$. We can choose either t_2 or t_3 in this step according to Strategy 1. To ease the description, we show only the process of choosing the test encountered earliest when more than one tests maximize $Div(T_o \cup \{t_i\})$, which is shown by Table 2. That is, in this step we select t_2. This process is to be repeated until T_o contains tests t_1, t_2, t_3, and t_4, with only t_5 unselected. If we select this last test, the division number $Div(T_o)$ will not increase at all (i.e., $Div(T_o \cup \{t_5\} = Div(T_o) = 7$). So the test-reduction process finishes without selecting t_5. Thus, the output of this test reduction is $\{t_1, t_2, t_3, t_4\}$. Moreover, if we select test t_3 after Step 1, not test t_2, the output of the test reduction is still $\{t_1, t_2, t_3, t_4\}$.

Table 2 Process of Strategy 1 for Program "Mid"

Steps	T_o	Existing Equivalence Classes	$Div(T_o)$	$Div(T_o \cup \{t_i\})$	
		$p_1 = \{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{12}, s_{13}\}$		t_2	4
Step 1	$\{t_1\}$		2	t_3	4
		$p_2 = \{s_4, s_5, s_6, s_7, s_{10}\}$		t_4	3
				t_5	2
		$p_1 = \{s_1, s_2, s_3, s_{13}\}$		t_3	5
Step 2	$\{t_1, t_2\}$	$p_2 = \{s_4, s_6, s_7\}$	4	t_4	5
		$p_3 = \{s_5, s_{10}\}$		t_5	4
		$p_4 = \{s_8, s_9, s_{11}, s_{12}\}$			
		$p_1 = \{s_1, s_2, s_3, s_{13}\}$			
Step 3	$\{t_1, t_2, t_3\}$	$p_2 = \{s_4\}$	5	t_4	7
		$p_3 = \{s_5, s_{10}\}$			
		$p_4 = \{s_6, s_7\}$		t_5	5
		$p_5 = \{s_8, s_9, s_{11}, s_{12}\}$			
		$p_1 = \{s_1, s_2, s_3, s_{13}\}$			
		$p_2 = \{s_4\}$			
		$p_3 = \{s_5\}$			
Step 4	$\{t_1, t_2, t_3, t_4\}$	$p_4 = \{s_6, s_7\}$	7	t_5	7
		$p_5 = \{s_8, s_9, s_{11}\}$			
		$p_6 = \{s_{10}\}$			
		$p_7 = \{s_{12}\}$			

2.2.2 Strategy 2

In Strategy 1, we select the test that can divide statements into most equivalence classes in each step without considering how the statements are divided. However, tests can partition statements in different ways. That is to say, a test can divide statements into several equivalence classes with similar or quite different sizes. Intuitively, if an equivalence class is divided quite unevenly by an unselected test, one of its sub-classes may be similar to the original equivalence class in size. Then more tests may be needed to divide the existing equivalence classes into more sub-classes. Therefore, this kind of division is often not desirable, as it could provide little help to distinguish the statements in the existing equivalence class.

In Strategy 2, we try to select a test that can not only divide more existing equivalence classes, but also divide them as evenly as possible. To quantify how evenly a test t divides an equivalence class, we propose the concept of the even division number. For an equivalence class p, which is induced by the existing selected tests, an unselected test t may partition this equivalence class into two sub-classes. The number of statements in the smaller sub-class is called the even division number of equivalence class p by test t, which is denoted as $Div_m(t, p)$. For convenience of presentation, if p is not divided by t, we define $Div_m(t, p)$ as 0.

Furthermore, as the aim of our approach is to locate the fault revealed by the initial failed test, we consider how to distinguish only the statements covered by the initial failed test in Strategy 2. That is, the division target of Strategy 2 is the statements covered by the initial failed test, not all the statements in the faulty program.

Based on the even division number, Strategy 2 is defined as follows: suppose that we have the set of currently selected tests T_o that divides the statements covered by the initial failed test f into several equivalence classes $p_1, p_2, \ldots, p_w$ at a certain stage of selection. Strategy 2 selects the test t that can maximize $\sum_{j=1}^{w} Div_m(t, p_j)$ among all the unselected tests. If more than one test satisfies this requirement, Strategy 2 selects any of these tests. The stopping condition of Strategy 2 is the same as Strategy 1.

When implementing Strategy 2, we can use the same mechanism for implementing Strategy 1. That is, we always maintain the existing equivalence classes and do some replacement to update them when a new test is selected. Moreover, we can use $Div_m(t, p)$ to describe the identical stopping condition of Strategy 2 presented earlier. That is, when $\sum_{j=1}^{w} Div_m(t, p_j) = 0$ for all unselected tests, no more tests are needed to be selected.

To illustrate how Strategy 2 works, also consider the example in Sect. 2.2.1, whose test-reduction process is shown by Table 3. Different from Table 2, the last column of Table 3 gives the value of division number $\sum_{j=1}^{w} Div_m(t_i, p_j)$, not $Div(T_o \cup \{t_i\})$.

The initial failed test t_1 divides the program into one equivalence class $p_1 = \{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{12}, s_{13}\}$. Then we calculate the sum of $Div_m(t, p_1)$ for each test in the test collection. The results are 4, 4, 1, and 0 for t_2, t_3, t_4, and t_5, respectively. $\sum_{j=1}^{w} Div_m(t_2, p_j) = \sum_{j=1}^{1} Div_m(t_2, p_j) = Div_m(t_2, p_1) = 4$, $\sum_{j=1}^{w} Div_m(t_3, p_j) = 4$, $\sum_{j=1}^{w} Div_m(t_4, p_j) = 1$, and $\sum_{j=1}^{w} Div_m(t_5, p_j) = 0$. Although tests t_2, t_3, and t_4 all divide the equivalence class p_1 into two equivalence classes, t_2 and t_3 both divide it into $\{s_1, s_2, s_3, s_{13}\}$ and $\{s_8, s_9, s_{11}, s_{12}\}$, whereas t_4 divides it into $\{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{13}\}$ and $\{s_{12}\}$. The former two divisions are more even than

Table 3 Process of Strategy 2 for Program "Mid"

Steps	T_o	Existing Equivalence Classes	$\sum_{j=1}^{w} Div_m(t_i, p_j)$	
			t_2	$\sum_{j=1}^{1} Div_m(t_2, p_j) = 4$
Step 1	$\{t_1\}$	$p_1 = \{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{12}, s_{13}\}$	t_3	$\sum_{j=1}^{1} Div_m(t_3, p_j) = 4$
			t_4	$\sum_{j=1}^{1} Div_m(t_4, p_j) = 1$
			t_5	$\sum_{j=1}^{1} Div_m(t_5, p_j) = 0$
Step 2	$\{t_1, t_2\}$	$p_1 = \{s_1, s_2, s_3, s_{13}\}$	t_3	$\sum_{j=1}^{2} Div_m(t_3, p_j) = 0$
		$p_2 = \{s_8, s_9, s_{11}, s_{12}\}$	t_4	$\sum_{j=1}^{2} Div_m(t_4, p_j) = 1$
			t_5	$\sum_{j=1}^{2} Div_m(t_5, p_j) = 0$
		$p_1 = \{s_1, s_2, s_3, s_{13}\}$	t_3	$\sum_{j=1}^{3} Div_m(t_3, p_j) = 0$
Step 3	$\{t_1, t_2, t_4\}$	$p_2 = \{s_8, s_9, s_{11}\}$	t_5	$\sum_{j=1}^{3} Div_m(t_5, p_j) = 0$
		$p_3 = \{s_{12}\}$		

the latter. So the former two sums of Div_m are bigger than the latter. As a result, we select t_2 or t_3 rather than t_4 and t_5 in this step. To ease the description, Table 3 gives only the following test-reduction process on choosing t_2 in this step. Following Strategy 2, we repeat the preceding process until T_o contains t_1, t_2, and t_4. At this stage, there is no need to select any more tests because the two remaining tests (i.e., t_3 and t_5) cannot divide any existing equivalence classes. That is to say, the process of test reduction finishes, and the output of this process is $\{t_1, t_2, t_4\}$. If we choose test t_3 rather than t_2 after Step 1, the output of test-reduction process will be $\{t_1, t_3, t_4\}$.

2.2.3 Strategy 3

In a typical TBFL approach such as TARANTULA (Jones and Harrold 2005; Jones et al. 2002), statements executed by different numbers of failed tests and different numbers of passed tests may be assigned with various suspicions, and thus are distinguishable for the developer. However, two statements covered by different sets of tests may still be undistinguishable from each other if these two statements are covered by the same numbers of passed tests and failed tests. To deal with this situation, Strategy 3 extends the definition of the binary relationship "T-undistinguishable". Given a target program P, whose statements are $s_1, s_2, \ldots, s_n$, and a set of tests $T = \{t_1, t_2, \ldots, t_m\}$, two statements of program P are called "extended T-undistinguishable" if and only if these two statements are covered by the same numbers of passed tests and failed tests.

Similar to "T-undistinguishable", this binary relationship is also an equivalence relationship. Based on the definition of "extended T-undistinguishable", the statements covered by the initial failed test are partitioned into several equivalence classes (Enderton 1977). Note that the equivalence classes in Strategy 3 are different from the equivalence classes in Strategy 1 and Strategy 2, because the former equivalence classes are based on the binary relation "extended T-undistinguishable" whereas the latter equivalence classes are based on the binary relation "T-undistinguishable".

If two statements are executed by the same number of passed tests and that of failed tests, these statements are "extended T-undistinguishable" in Strategy 3, even if they are executed by different sets of tests. That is, if two statements are "T-undistinguishable", then these statements are "extended T-undistinguishable". However, two "extended T-undistinguishable" statements cannot be guaranteed to be "T-undistinguishable".

In TBFL, the more failed tests one statement is involved in, the more suspicious this statement is. Therefore, when selecting a new test, statements covered by more failed tests should be paid more attention to. As all the statements in an equivalence class are covered by the same number of failed tests, an unselected test that could divide the existing equivalence classes whose statements are covered by many failed tests should be paid more attention to. That is to say, the priority for one existing equivalence class to be further divided could be different from the priority for another equivalence class: An equivalence class whose statements are covered by more failed tests should be assigned a higher priority. During the test-reduction process, Strategy 3 calculates the priority of an equivalence class p by $Pri(p) = T_f(s)/T_f$, where T_f denotes the number of failed tests that have already been selected, and $T_f(s)$ denotes the number of selected failed tests that cover a statement s in p.

Thus, the reduction process of Strategy 3 is as follows. Suppose that the set of currently selected tests T_o divides the statements covered by the initial failed test into the following equivalence classes $ep_1, ep_2, \ldots, ep_w$ at a certain stage of selection. Strategy 3 selects the test t that maximizes $\sum_{j=1}^{w} Div_m(t, ep_j) * Pri(ep_j)$ among all the unselected tests. When more than one test satisfies this requirement, Strategy 3 selects any of these tests. The stopping condition of Strategy 3 is similar to that of Strategy 1 or 2. The only difference is that the stopping condition in Strategy 3 is based on the concept of "extended T-undistinguishable".

Similar to the preceding two strategies, we can use the same mechanism to facilitate the implementation of Strategy 3. After t is selected, we need to replace each equivalence class that can be divided by the selected test. If t divides an equivalence class ep into two sets of statements denoted as st_1 and st_2, we replace ep with st_1 and st_2. After replacing all the divided equivalence classes, we also need to do some merging to ensure that we still have a series of equivalence classes based on "extended T-undistinguishable".

We next explain the process of Strategy 3 by applying it to "Mid", which is shown by Table 4. Initially, T_o has only one test t_1, which divides "Mid" into one equivalence class $ep_1 = \{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{12}, s_{13}\}$. As now T_o has only one failed test, and all the preceding statements are executed by this failed test. Thus, $Pri(ep_1)$ is 1 and $\sum_{j=1}^{1} Div_m(t_2, ep_j) * Pri(ep_j) = Div_m(t_2, ep_1) * Pri(ep_1) = 4$. Similarly, we have $\sum_{j=1}^{1} Div_m(t_3, ep_j) * Pri(ep_j) = 4$, $\sum_{j=1}^{1} Div_m(t_4, ep_j) * Pri(ep_j) = 1$, and $\sum_{j=1}^{1} Div_m(t_5, ep_j) * Pri(ep_j) = 0$. These results indicate that we should select t_2 or t_3. Table 4 lists the test-reduction process when t_2 is chosen at this stage. We repeat the preceding process until the set of existing selected tests T_o contains $\{t_1, t_2, t_4\}$. When we examine whether we need to set another test, we find that the stopping condition is satisfied. Thus, Strategy 3 stops and the output is $\{t_1, t_2, t_4\}$. If we select test t_3 rather than t_2 after Step 1, the output will be $\{t_1, t_3, t_4\}$.

Table 4 Process of Strategy 3 for Program "Mid"

Steps	T_o	Existing Equivalence Classes	$\sum_{j=1}^{w} Div_m(t, ep_j) * Pri(ep_j)$	
			t_2	$\sum_{j=1}^{1} Div_m(t_2, ep_j) * Pri(ep_j) = 4$
Step 1	$\{t_1\}$	$ep_1 =$	t_3	$\sum_{j=1}^{1} Div_m(t_3, ep_j) * Pri(ep_j) = 4$
		$\{s_1, s_2, s_3, s_8, s_9, s_{11}, s_{12}, s_{13}\}$	t_4	$\sum_{j=1}^{1} Div_m(t_4, ep_j) * Pri(ep_j) = 1$
			t_5	$\sum_{j=1}^{1} Div_m(t_5, ep_j) * Pri(ep_j) = 0$
			t_3	$\sum_{j=1}^{2} Div_m(t_3, ep_j) * Pri(ep_j) = 0$
Step 2	$\{t_1, t_2\}$	$ep_1 = \{s_1, s_2, s_3, s_{13}\}$	t_4	$\sum_{j=1}^{2} Div_m(t_4, ep_j) * Pri(ep_j) = 1$
		$ep_2 = \{s_8, s_9, s_{11}, s_{12}\}$	t_5	$\sum_{j=1}^{2} Div_m(t_5, ep_j) * Pri(ep_j) = 0$
		$ep_1 = \{s_1, s_2, s_3, s_{13}\}$	t_3	$\sum_{j=1}^{3} Div_m(t_3, ep_j) * Pri(ep_j) = 0$
Step 3	$\{t_1, t_2, t_4\}$	$ep_2 = \{s_8, s_9, s_{11}\}$	t_5	$\sum_{j=1}^{3} Div_m(t_5, ep_j) * Pri(ep_j) = 0$
		$ep_3 = \{s_{12}\}$		

Although the result produced by Strategy 3 is the same as that produced by Strategy 2 for this example, Strategy 2 and Strategy 3 do not always produce the identical results. In fact, Strategies 2 and 3 behave quite differently for general circumstances; such different behaviors are shown by the results of our experiment in Sect. 4.

3 Discussion

In this section, we first discuss the relation between the problem in this paper and test-suite reduction in Sect. 3.1. Then we discuss the characteristics of our approach (Sect. 3.2)and application of our approach (Sect. 3.3).

3.1 Test-suite reduction

In the literature, there are already many test-selection (Graves et al. 2001; Rothermel and Harrold 1997, 1998) and test-suite reduction (Jeffrey and Gupta 2005; Rothermel et al. 2002; Sprenkle et al. 2005) techniques, which deal with the situation where there are too many tests for the tester to use in a round of testing, especially regression testing. In fact, as the purposes of previously proposed test-reduction techniques are totally different from the purpose of our approach, the tests selected by the existing test-suite reduction techniques may not be as effective as the ones selected by our strategies for fault localization.

The existing test-suite reduction techniques may select tests that may be unlikely to improve fault-localization effectiveness. If we apply the technique for test-suite reduction proposed by Harrold et al. (1993) for example "Mid", the reduced test collection is $\{t_1, t_2, t_3\}$ or $\{t_2, t_3, t_5\}$. As t_1 is the specified initial failed test, it must be included in the selected test set. So the result of this test-reduction technique is $\{t_1, t_2, t_3\}$. These selected tests partition the suspicious statements, which are referred as to the statements executed by the initial failed test t_1, into two equivalence classes $\{s_1, s_2, s_3, s_{13}\}$ and $\{s_8, s_9, s_{11}, s_{12}\}$. Based on either the binary relation "T-undistinguishable" or "extended T-undistinguishable", we get the identical equivalence classes. However, either $\{t_1, t_2\}$ or $\{t_1, t_3\}$ partitions the suspicious statements into the same preceding equivalence classes. Therefore, this traditional test-suite reduction technique may induce redundant tests, which do not distinguish more statements and thus may be unlikely to improve fault-localization effectiveness.

The tests selected by the existing test-suite reduction techniques may not be as effective as the tests selected by our approach in fault localization if fed to some TBFL approaches. When applying TARANTULA (Jones and Harrold 2005; Jones et al. 2002) on the selected test sets of program "Mid", we can get the result described below. For any test sets selected by our strategies, the faulty statement (i.e., s_{12}) is uniquely ranked with the highest suspicion. For the test set selected by Harrold et al.'s technique for test-suite reduction, statements s_8, s_9, s_{11}, and s_{12} have the same highest suspicion. In this example, the tests selected by our approach are better than the tests selected by Harrold et al.'s technique in fault localization. In fact, test t_4 plays a key role in this example. If t_4 is not selected, the best result is just to rank statements s_8, s_9, s_{11}, and s_{12} as top suspects with the same highest suspicion. However, test t_4

would not be selected by Harrold et al.'s technique because this technique selects a test that could cover more statements and the statements executed by t_4 are already covered by test t_1. That is, this test-suite reduction technique selects tests that could cover more statements, but not necessarily distinguish more statements; such selected mechanism makes it less suitable to address the problem proposed in this paper. We will demonstrate this point later by the experimental results in Sect. 4.

3.2 Characteristics of our approach

In this section, we discuss some issues involved in the approach. First, the process of selecting tests for result inspection can be essentially adaptive. Each step of selection can be dependent on the result of the previous steps. The main advantage of using an adaptive process is that newly acquired information can be immediately used in the subsequent step. Strategy 3 is adaptive. The information acquired when checking whether a selected test is passed or failed is used in Strategy 3 when we calculate the priority of an equivalence class in the subsequent step. Of course, neither Strategies 1 nor 2 is adaptive as they always ignore the information newly acquired in previous steps.

Second, from the descriptions of the three strategies, we hypothesize that Strategy 1 might select more tests than Strategy 2, and Strategy 3 might select more tests than the other two strategies. Because Strategy 2 selects tests that tend to divide the existing equivalence classes more evenly, Strategy 2 may partition statements more quickly than the other two strategies. In Strategy 3, statements are partitioned based on the number of failed tests and that of passed tests. However, the numbers of failed and passed tests continuously change during the reduction process. Thus, two "extended T-undistinguishable" statements may be not "extended T-undistinguishable" any more after a new test is selected. For example, statement s_1 is covered by one selected passed test and one selected failed test, whereas statement s_2 is covered by two selected passed tests and one selected failed test. Then based on the existing selected tests, these two statements are not "extended T-undistinguishable", and they belong to different equivalence classes. If another test t is selected and checked to be passed, and it covers statement s_1 but does not cover statement s_2, then these two statements are "extended T-undistinguishable" because these statements are both covered by the same numbers of failed tests and passed tests. Thus, these two statements belong to an equivalence class after test t is selected. We validate this hypothesis with the experimental results in Sect. 4.

3.3 Application of our approach

In this section, we discuss some issues that potentially influence the application of our approach.

First, our approach requires to separate the process of executing tests and checking their results to make coverage information available for our reduction strategies. However, sometimes it is difficult or even impossible to separate this process. For example, when testing an interactive GUI-based application, developers may have to enter the corresponding data of a test and check the output interactively during the execution of the test. To address the issue, our approach can be carried out in two-phase

test execution: in the first phase, all tests are executed to collect coverage information; in the second phase, the selected tests are re-executed for result checking. Indeed, the re-execution of the selected tests may induce some extra cost for fault localization. Moreover, our approach in this paper cannot be used directly in a circumstance that the test execution process is not repeatable, such as non-deterministic parallel applications. We plan to investigate this problem in our future work.

Second, neither Strategy 1 nor strategy 2 is specific to any particular TBFL approaches, but Strategy 3 is specific to a category of TBFL approaches such as TARANTULA (Jones and Harrold 2005; Jones et al. 2002). In Strategy 3, statements covered by the same number of failed tests and the same number of passed tests are viewed as inseparable. This design decision aims at TBFL approaches where the suspicion of a statement is calculated based on only the number of failed tests and the number of passed tests that cover the statement. TARANTULA is a typical approach in this category. However, it is probably not applicable or effective for other approaches that do not use the number of passed tests and the number of failed tests, such as the nearest neighbor queries approach (Renieris and Reiss 2003). This approach calculates the suspicions of statements based on the distance measurement of passed tests and failed tests. That is to say, besides the strategies that are applicable to general TBFL approaches, there exist many specific strategies that are applicable to only particular TBFL approaches. Developing specific strategies for specific TBFL approaches may be a future direction of research.

Third, a faulty program may have more than one fault in practice. If a test collection contains only tests revealing fault A, then this test collection is effective in fault-localization of the fault A. However, if the test collection contains both tests revealing fault A and tests revealing a different fault B. Then this test collection may be uneffective in fault localization of either fault A or B because the tests revealing A and the tests revealing B impact each other. Our test-reduction strategies might exclude the impacts of one fault from the others because our strategies are targeted at selecting tests only for locating the fault revealed by the initial failed test, which might contain only one fault (either A or B).

4 Experiment

This section presents an experiment that we have conducted to evaluate the effectiveness of our proposed approach. In particular, we investigate two research questions (Sect. 4.1) by applying our approach to the subject programs (Sect. 4.2). Then we describe two typical TBFL approaches used by our experiment in Sect. 4.3. We describe the process of conducting the experiment in Sect. 4.4. We present the experimental results and analysis in Sect. 4.5. Finally we discuss the threats to validity of our experiment in Sect. 4.6.

4.1 Research questions

As our test-reduction approach is to reduce the number of tests whose results are examined by developers, our experiment investigates whether our test-reduction technique can reduce the size of the test collection for inspection by selecting a small

subset of tests from the test collection. Moreover, as the tests selected by our approach are used for fault localization, not fault revealing, our experiment also investigates whether the selected tests decrease the fault-localization effectiveness of TBFL approaches.

Therefore, in our experiment, we try to answer the following research questions: (1) Can our test-reduction strategies choose a small subset of tests from the test collection? (2) Can the tests selected by our approach decrease the fault-localization effectiveness of TBFL approaches compared with the whole test collection?

4.2 Subject programs

In our experiment, we use the Siemens programs (Hutchins et al. 1994; Rothermel and Harrold 1998), the desk calculator program (abbreviated as DC[6]) (Hao et al. 2005a), and the tiny C compiler (abbreviated as TCC)[7] as the subject programs to evaluate the effectiveness of our approach. All the programs are written in C. The details of these subject programs are in Table 5. In the table, the "Program" column gives the names of the seven Siemens programs and the other subject programs. The "Description" column gives the functionality of these programs. "LOC" presents lines of code for each program. "Version" presents the number of faulty versions of each program. The "Test" column presents the number of tests for each program.

The Siemens programs are a suite of seven small programs, which were first collected and used by Hutchins et al. (1994) and later enhanced by Rothermel and Harrold (1998). Each program has several faulty versions, each of which contains a single fault that was gathered from real experience. Every program has exactly or more than 1,000 tests.

DC is an arbitrary precision calculator[8] of GNU, whereas TCC is an open source program, whose source code can be downloaded from http://bellard.org/tcc/. Because

Table 5 Experimental subject programs

Program	Description	Version	LOC	Test
print_tokens	lexical analyzer	7	565	4072
print_tokens2	lexical analyzer	10	510	4057
replace	pattern replacer	32	563	5542
schedule	priority scheduler	9	412	2627
schedule2	priority scheduler	10	307	2683
tcas	altitude separator	41	173	1592
tot_info	info measurer	23	406	1026
DC	desk calculator	17	2700	1000
TCC	C compiler	10	19000	1000

[6]Our experiment used bc-1.06.

[7]Our experiment used tcc-0.9.20.

[8]http://www.gnu.org/software.

neither DC nor TCC originally had faulty versions, a graduate student, the first author of this paper manually injected some faults[9] into the DC program and the TCC program. When the student injected the faults, she used different mutation operators for the C language to mutate the statements in DC and TCC. There are about 70 mutation operators for the C language reported in the literature (Barbosa et al. 2001). During the fault-seeding process, the student randomly selected a subset of these operators to generate 17 faulty versions for DC (9 versions of buggy programs with two faults, and the other 8 versions with three faults) and 10 faulty versions for TCC (5 versions of buggy programs with two faults, and the other 5 versions with three faults). As DC did not have a test collection, the same student produced 1,000 tests for DC. For TCC, we selected 1,000 C programs[10] distributed in GCC 3.3.2 and GCC 3.4.0 as the tests.

As a faulty version of the Siemens programs contains only one fault, we used the faulty versions of the Siemens programs to represent programs with single faults. However, each faulty version of DC or TCC contains two or three faults, we used the faulty versions of DC and TCC to represent programs with multiple faults.

4.3 Two typical TBFL approaches

To evaluate the fault-localization effectiveness of the selected tests, we apply the output of test reduction to two typical TBFL approaches, including TARANTULA (Jones and Harrold 2005) and our modified Dicing approach (Agrawal et al. 1995).

Originally, TARANTULA is proposed as a visualization tool (Jones et al. 2002) for fault localization, but it is naturally adapted to produce a ranked list of suspicious statements for the ease of evaluation (Jones and Harrold 2005). Moreover, it has been empirically investigated (Jones and Harrold 2005) to be better than other TBFL approaches. In our experiment, we reimplemented TARANTULA proposed by Jones and Harrold (2005).

The Dicing approach is first proposed by Agrawal et al. (1995) based on the concept of a dice. A dice is the set of statements that are in the execution slice of a failed test (denoted as $Slice_{failed}$) but not in the execution slice of a passed test (denoted as $Slice_{passed}$), i.e., dice $= Slice_{failed} - Slice_{passed}$. The original dicing approach locates the faults to the statements in a dice. However, the faults may lie in the common statements of $Slice_{passed}$ and $Slice_{failed}$. Moreover, the statements in the dice are equally suspicious. Thus, it may still be a burden for developers to check whether the statements in the dice are suspicious one by one. To address these two problems, we modify the original Dicing approach by computing the dices between any failed test and any passed test, and rank each statement according to the number of dices that contain this statement. For convenience, we use "Dicing" to refer to our modified Dicing approach in the rest of this paper.

[9] These faulty programs were generated (Hao et al. 2008) to evaluate the fault-localization effectiveness of some TBFL approaches.

[10] These programs are small because almost each of them occupies 1 KB storage.

4.4 Experimental process

Before applying our strategies, developers may have some checked tests, whose results are passed or failed. The number of these passed or failed tests can be arbitrary in practice. To provide a dependable comparison, here we assumed that the number of passed or failed tests selected before the initial failed test is zero.

In our experiment, our approach took a test collection and a failed test as the input, and the aim was to help locate the fault revealed by the initial failed test. The initial failed test would influence the effectiveness of the reduction strategies and also the results of fault localization. To reduce the influence of the selection of the initial failed test on the validity of our experiment, for each faulty version, we randomly selected 30 failed tests and used each of them as the input initial failed test. When a test was selected as the initial failed test, the other tests in the test collection were treated as the input test collection. The results of the tests in the input test collection were not checked initially and the information of whether each test is passed or failed became available only after it was selected and checked. The average experimental results of using these 30 input initial failed tests were then treated as the corresponding experimental results for that faulty version. The average experimental results of all the faulty versions of a program were treated as the final experimental results for that program.

Given a faulty version and an initial failed test for this faulty version, the experimental steps are as follows. First, we applied the proposed three reduction strategies to the faulty version with the initial failed test and the test collection that consists of the other tests. When no new test could be selected by a reduction strategy, the existing selected tests would be taken as the output of that strategy and fed to TARANTULA (Jones and Harrold 2005) and Dicing (Agrawal et al. 1995). The number of the existing selected tests was recorded. The number of statements that have to be scanned before finding the first fault if TARANTULA or Dicing is applied is recorded as the result of TARANTULA or Dicing. That is to say, for each strategy, we recorded the following data: (1) the number of selected tests, (2) the number of statements that have to be scanned if TARANTULA was applied to the selected tests, and (3) the number of statements that have to be scanned if Dicing was applied. For comparison, we also applied the test-reduction technique proposed by Harrold et al. (1993) to the test collection guaranteeing that the initial failed test is selected, and then fed the selected tests to TARANTULA and Dicing. We also recorded the results of applying TARANTULA and Dicing to the original test collection. The fault-localization results based on the original test collection are not specific to the initial failed test.

As the correct version of each subject program is available, the expected output of any test can be acquired by executing it on the correct version. In this regard, all the tests are actually test cases (i.e., test inputs with expected outputs). However, in our experiment, we assumed that the expected outputs are not available when applying our three strategies, and the information whether a test is passed or failed is available only after it is selected and checked. When applying TARANTULA and Dicing to the original test collection, we assumed that all the tests in the test collection have been checked before applying these two techniques.

For comparison, we applied a test-reduction technique named as the HGS algorithm proposed by Harrold et al. (1993) as well as the random test-reduction technique to the subject programs. Based on the preceding experiment, we recorded the number of tests selected by each of the three strategies, and randomly selected the same number of tests. Specifically, the random test-reduction technique based on the number of tests selected by Strategy 1 is denoted by R1, whereas the corresponding Strategy 1 is denoted by S1 in our experiment. Then, the random test-reduction technique based on the number of tests selected by Strategy 2 (denoted as S2) is denoted by R2, whereas the random test-reduction technique based on the number of tests selected by Strategy 3 (denoted as S3) is denoted by R3. Moreover, in our experiment we use HGS to represent the HGS algorithm proposed by Harrold et al. (1993).

4.5 Experimental results and analysis

Figure 2 summarizes the average results of our experiment and compares our three test-reduction strategies with the HGS algorithm (Harrold et al. 1993). This figure consists of three sub-figures. Figure 2(a) depicts the test-reduction results, which show the ratio between the number of selected tests and the total number of tests in the corresponding test collection. Figure 2(b) shows the fault-localization results by applying the selected tests to TARANTULA. Figure 2(c) shows the fault-localization results by applying the selected tests to Dicing. The horizontal axis of each sub-figure is the subject programs in our experiment, which are numbered in the horizontal axis following the order of their appearance in Table 5. Specifically, the number

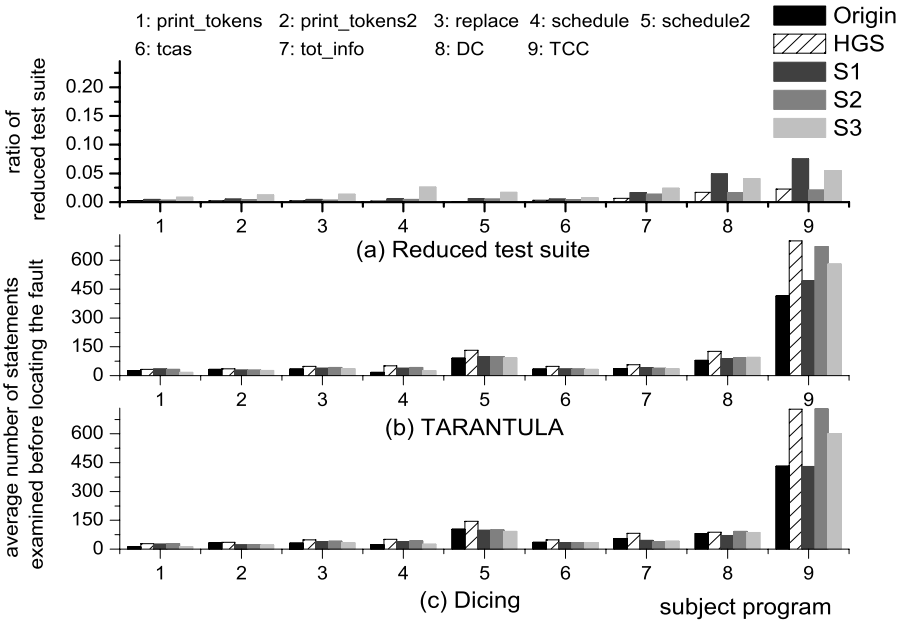

Fig. 2 Comparison with HGS

1, 2, . . . , 7, 8, and 9 in the horizontal axis denotes print_tokens, print_tokens2, replace, schedule, schedule2, tcas, tot_info, DC, and TCC, respectively. The vertical axis of Fig. 2(a) is the ratio between the average number of selected tests and the total number of tests in each test collection, whereas the vertical axis of Figs. 2(b) and (c) is the average number of statements examined before finding the faulty statement when applying the reduced test suite to the corresponding fault-localization approaches (i.e., TARANTULA and Dicing). The compared test-reduction approaches in this figure are the three proposed test-reduction strategies (S1, S2, and S3) and the HGS algorithm proposed by Harrold et al. (1993). For further comparison, we give the fault-localization results of applying the whole original test suite to TARANTULA or Dicing, which are abbreviated as "Origin" in the figure.

From Fig. 2(a), we observe that less than 10% of the tests in the original test collection are selected by our three strategies and Harrold et al.'s approach (Harrold et al. 1993). Moreover, except for the test-reduction results for DC and TCC (the eighth and ninth subject programs in this figure), our proposed test-reduction strategies reduce the scale of the test collection effectively as does the test-suite reduction technique proposed by Harrold et al. (1993). The results show that these techniques can effectively reduce effort for inspecting test results. Specifically, Strategy 3 is mostly less effective than Strategy 1, Strategy 2, and HGS in test-suite reduction, as it selects more tests than the other techniques in each of the ten subject programs except for the last two subject programs DC and TCC. Moreover, Strategy 2 is usually more effective than Strategy 1 as well as Strategy 3, and is usually as effective as HGS in test-suite reduction. From Fig. 2(b) and 2(c), we observe that the tests selected by any of our three strategies are more effective than HGS in fault localization in most subjects. Moreover, the fault-localization effectiveness of the tests selected by our strategy is close to whole test collection in most subjects except for the last subject program TCC.

Furthermore, we draw Fig. 3 to compare the three proposed test-reduction strategies (Si, i = 1, 2, and 3) with the corresponding random test-reduction strategies (Ri, i = 1, 2, and 3) on the fault-localization effectiveness of their selected tests since each pair (Si and Ri) selects the same number of tests from a test collection. Bars in various colors denote our three proposed test-reduction strategies, whereas curves with various symbols denote the corresponding three random test-reduction strategies. From this figure, we can observe that most of the bars are below the curves except for the last subject program (i.e., TCC). That is, mostly our test-reduction strategy is more effective than the corresponding random test-reduction strategy in fault localization since the fault-localization results of tests selected by our test-reduction strategy are mostly smaller than that by the corresponding random test-reduction strategies.

To further confirm the difference of these test-reduction strategies on reducing tests as well as fault localization, we performed a sign test (Joseph 1992) on the average results of these strategies. The results of the sign test is shown by Table 6, where "Fault Localization" represents that we compared the corresponding two test-reduction strategies on fault-localization effectiveness and "Test Reduction" represents that we compared the corresponding two test-reduction strategies on the test-reduction effectiveness. For each row in this table, "X vs. Y" denotes the comparison is between strategy X and strategy Y, where X and Y are one of the following

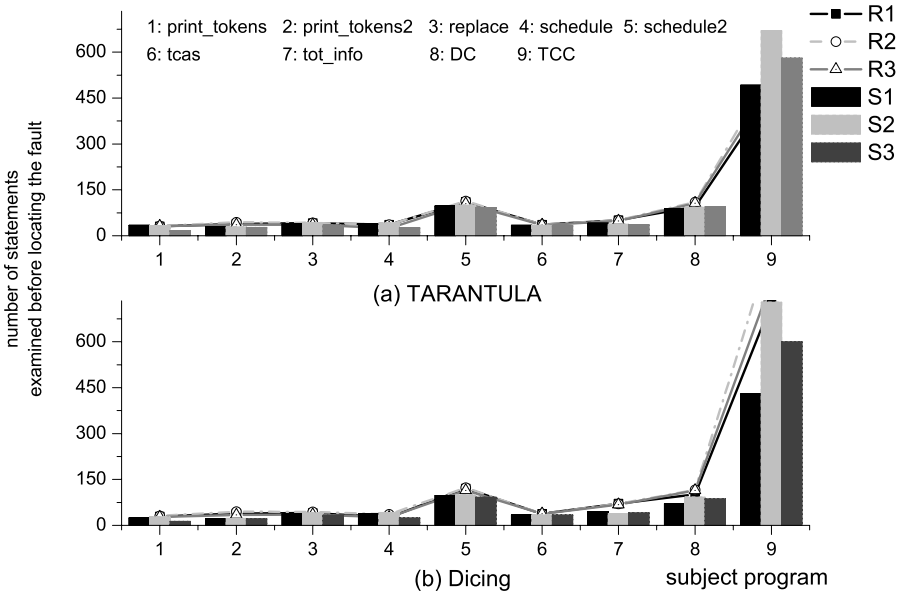

Fig. 3 Comparison with Random Test Selection

Table 6 Sign Test ($\alpha = 0.05$)

Comparison		n_-	n_+	p-value	Result
R1 vs. S1	Fault Localization	14	4	0.0154	Reject
R2 vs. S2	Fault Localization	13	5	0.0481	Reject
R3 vs. S3	Fault Localization	16	2	<0.0001	Reject
S1 vs. S2	Test Reduction	9	0	0.0020	Reject
S2 vs. S1	Fault Localization	12	3	0.0176	Reject
S3 vs. S1	Test Reduction	7	2	0.0898	Not reject
S1 vs. S3	Fault Localization	13	4	0.0245	Reject
S3 vs. S2	Test Reduction	9	0	0.0020	Reject
S2 vs. S3	Fault Localization	14	3	0.0064	Reject
S1 vs. HGS	Test Reduction	9	0	0.0020	Reject
HGS vs. S1	Fault Localization	17	1	<0.0001	Reject
S2 vs. HGS	Test Reduction	7	2	0.0898	Not reject
HGS vs. S2	Fault Localization	14	3	0.0245	Reject
S3 vs. HGS	Test Reduction	9	0	0.0020	Reject
HGS vs. S3	Fault Localization	18	0	<0.0001	Reject

strategies: our proposed three strategies S1, S2, S3, the corresponding random test-reduction strategies R1, R2, R3, and the HGS algorithm. The third and fourth columns n_- and n_+ denote the number of comparisons that Y wins X and the number of com-

parisons that X wins Y. The fifth column lists the computed p-value and the sixth column gives the result of the sign test. In this table, we use "< 0.0001" to represent that the corresponding p-value is much smaller. According to Table 6, we draw the following conclusions.

1. *In the effectiveness of test suite reduction*: The HGS algorithm is significantly more effective than S1 and S3, and more effective than S2 in most cases. S2 is significantly more effective than S1 and S3, but S1 is more effective than S3 in most cases.
2. *In the effectiveness of fault localization*: The three proposed test-reduction strategies are significantly more effective than the corresponding random test-reduction strategies and the HGS algorithm. S3 is significantly more effective than S1, and S1 is significantly more effective than S2.

Besides the preceding analysis on average results, we compute the standard deviations (shown in Table 7) on the experimental results. In Table 7, the first column shows whether the standard deviations are based on the number of reduced tests (abbreviated as Test), or the fault-localization results based on TARANTULA (abbreviated as TAR), or the fault-localization results based on Dicing (abbreviated as Dicing). We abbreviated the names of some subject programs in this table. For example,

Table 7 Standard Deviations

		print1	print2	replace	sche	sche2	tcas	tot	DC	TCC
	S1	1.71	1.26	1.10	0.87	1.37	0.56	1.09	2.94	84.53
	S2	1.81	2.67	4.41	0.59	1.14	1.05	3.93	4.02	26.04
	S3	14.27	22.03	42.29	16.45	11.69	7.00	9.78	18.98	104.30
Test	R1	1.71	1.26	1.10	0.87	1.37	0.56	1.09	2.94	84.53
	R2	1.81	2.67	4.41	0.59	1.14	1.05	3.93	4.02	26.04
	R3	14.27	22.03	42.29	16.45	11.69	7.00	9.78	18.98	104.30
	HGS	2.54	0.83	0.54	0.00	0.00	0.00	0.41	1.60	23.62
	S1	21.45	26.45	41.85	24.88	27.35	21.47	33	131.59	355.67
	S2	23.46	26.73	43.08	22.14	24.49	21.48	30.29	117.59	453.57
	S3	8.07	25.88	38.82	21.62	33.28	21.79	29.14	139.80	405.49
TAR	R1	22.19	42.49	41.92	31.28	38.14	21.16	35.57	143.39	335.31
	R2	21.56	51.34	43.18	30.86	38.33	21.25	34.45	134.63	373.04
	R3	21.90	39.03	38.84	22.60	38.97	21.57	33.65	155.67	354.07
	HGS	22.51	40.67	45.54	41.47	10.24	20.77	41.81	157.58	433.09
	S1	22.75	25.77	40.84	24.88	27.34	21.49	39.99	131.92	456.86
	S2	24.93	26.89	42.26	22.14	24.46	21.49	37.04	148.42	1112.04
	S3	7.55	29.65	38.71	22.10	33.79	21.88	38.09	163.96	904.16
Dicing	R1	22.18	48.54	42.93	31.89	42.00	23.49	47.41	160.15	1180.91
	R2	21.67	58.47	50.25	31.77	42.31	23.28	44.68	158.41	1336.91
	R3	21.77	42.24	42.58	23.61	43.17	23.70	45.89	180.19	1262.08
	HGS	23.74	40.82	45.49	41.47	0.86	20.77	53.46	175.77	1284.41

print1 denotes print_tokens1, print2 denotes print_tokens2, sche denotes schedule, sche2 denotes schedule2, and tot denotes tot_info. On test-reduction effectiveness, the standard deviations of S1, S2, and HGS are usually small, but the standard deviations of S3 are much bigger. However, on fault-localization effectiveness, the standard deviations of S1, S2, S3, R1, R2, R3, and HGS are close.

4.6 Threats to validity

The main threat to construct validity deals with whether the experiment is measured in a correct way. In our experiment, we measured only the effectiveness of test reduction by the number of selected tests, without the cost and effort spent in test reduction. Moreover, the fault-localization effectiveness of the selected tests is measured based on the evaluation framework proposed by Renieris and Reiss (2003). Although this measurement is widely used (Cleve and Zeller 2005; Hao et al. 2005a; Liu et al. 2005) in the literature of fault localization, such measurement ignores the difference of statements by counting the number of statements. In future work, we will design a novel measurement for our test-reduction approach by considering the preceding factors.

The main threats (Trochim 2007; Wohlin et al. 1999) to internal validity are as follows. The first threat is the TBFL approaches used in this experiment. To reduce this threat, we chose two typical approaches in our experiment. Strategy 3 is to some extent specific to TARANTULA and Dicing, and the results of Strategy 3 is to some extent specific to these two approaches. We can further reduce this threat by conducting experiments on other TBFL approaches.

The second threat is the assumption that the number of passed and failed tests selected before the initial failed test is zero. However, we do not think that this assumption can seriously impact the main conclusions of our experiment. With more initially selected tests, all the three strategies should select fewer tests during the reduction process. If these initially selected tests can harm fault localization, we can use the three strategies to exclude those harmful ones. Moreover, the initial failed test can also impact the conclusion of our experiment. We reduced this threat by randomly choosing the initial failed test 30 times. Generally speaking, the second internal threat can also be reduced by conducting more experiments. In these experiments, we can simulate the situation by using different sets of initially selected tests and record the average results. These experiments would be much more complicated, left as our future work.

The third threat comes from the order of tests in each test collection, because our experiment selects the first appearing test when more than one test satisfies the test-reduction requirement. To reduce this threat, we kept the original orders of these tests in the test collection, which are independent of our strategies. Moreover, tests for the Siemens programs were ordered by their providers, tests for TCC are ordered by their names in the dictionary order, and tests for DC are generated by a student unaware of our strategies. We plan to reduce this threat by experiments with random order of tests and reverse order of tests in future work.

The fourth threat comes from the manually produced test collection of DC and the faults of DC as well as TCC, because the experimental results of DC and TCC

may be affected by the performance of the student who conducted the manual work. This threat was reduced when the faults were seeded using standard procedures and the tests used in the experiment can be viewed as those generated by an ordinary software engineer. This threat can be further reduced by more experiments involving more different software engineers.

Finally, a threat lies in the compared test-suite reduction technique used in our experiment. In our experiment, we implemented the HGS algorithm and made a comparison between our three strategies with this algorithm, although in the literature of test-suite reduction, there are some other algorithms such as greedy algorithms (Ma et al. 2005). In future work, we plan to compare our three strategies with other algorithms to reduce this threat.

The main threats to external validity lie in the subject programs used in the experiment. The subject programs are not large, and the languages for these programs are all C. These threats can be reduced by applying our approach to other larger programs written in different languages.

5 Related work

Our approach is related to test selection, reduction, minimization and prioritization because our approach provides strategies on test reduction also based on statement-coverage information. However, the aim of our test reduction is to select tests as input to TBFL approaches, and then locate faults. So our work is also related to testing-based fault-localization approaches.

5.1 Test selection and prioritization

Test selection and prioritization, test-suite reduction and minimization are extensively discussed research topics in the literature. Test selection aims at selecting some tests from a given test suite for some specific purpose, whereas test prioritization schedules tests for execution in an order to achieve some specific goal. Test-suite reduction and minimization in the literature aim at eliminating redundant test cases from a test suite during software maintenance.

The random technique is a straightforward way to test selection and prioritization, but this technique could not guarantee the quality of selected tests. Besides this technique, most of the existing test selection and prioritization approaches are based on structural coverage such as coverage of statements and branches (Jones and Harrold 2003; Rothermel and Harrold 1998). Recently, researchers focus on empirical studies of test selection and prioritization. Graves et al. (2001) presented an empirical study of several test-selection techniques on their costs and benefits. Rothermel et al. (2001) summarized some criteria of test prioritization, which are based on statement coverage, branch coverage, and fault-exposing potential, respectively. Furthermore, Elbaum et al. (2002) investigated other research questions on test prioritization such as the effectiveness of coarse-granularity techniques and fine-granularity techniques by a family of empirical studies. Srivastava and Thiagarajan (2002) proposed a test-prioritization technique based on programs in binary forms. Moreover, their

technique prioritizes tests according to the test coverage on the affected program, but does not eliminate any tests. Walcott et al. (2006) proposed a genetic algorithm to prioritize tests based on a given time constraint. Jeffrey and Gupta proposed to select redundant test cases (Jeffrey and Gupta 2007) to increase the fault detection effectiveness of the reduced test suite.

Our work can be viewed as test reduction, as our strategies are to select tests from the given test collection and reduce the size of the test collection. However, traditional test selection and test-suite reduction aim at facilitating testing, such as regression testing whereas our approach aims at facilitating debugging. Our approach is similar to test prioritization as both adopt an adaptive process. However, our approach is an adaptive process that uses information such as failed and passed information about the tests, but test prioritization aims at optimizing the execution order of the selected tests for a specific goal, such as detecting a fault earlier.

Without test oracles for automatically generated tests, it is infeasible for developers to inspect the result of each generated test in order to detect failures beyond program crashes or uncaught exceptions. Xie and Notkin (2006) as well as Pacheco and Ernst (2005) developed approaches for selecting the most suspicious subset of generated tests for result checking. Different from their approaches, our reduction's purpose is not to detect unknown failures but to locate faults based on known failures.

Besides the preceding research on the traditional test selection and prioritization, we attempted to investigate how the test cases influence the fault-localization effectiveness of TBFL approaches. Our previous work on test reduction (Hao et al. 2005b) assumed that redundant test cases might harm the effectiveness of TBFL approaches, and proposed to use traditional test-reduction techniques before applying TBFL approaches. Recently, Yu et al. (2008) further investigated the relation between test cases and TBFL approaches with an empirical study.

5.2 Fault localization

The aim of our approach is to facilitate testing-based fault localization (TBFL), and we have applied our approach to two typical fault-localization approaches (i.e., TARANTULA, Jones and Harrold 2005; Jones et al. 2002 and Dicing, Agrawal et al. 1995) in our experiment. Besides these two approaches, there are also some other testing-based fault-localization approaches in the literature. Our previous work (Hao et al. 2005a) took into consideration the influence of the distribution of tests on TBFL, and proposed an approach based on the fuzzy set theory. The nearest neighbor query approach (Renieris and Reiss 2003) measures the distance between failed test cases and passed tests, and uses this distance to calculate the suspicions of statements. The main common characteristic of TBFL approaches is that they take the execution information of tests as input to calculate the suspicions, but they employ different strategies for the suspicion calculation. Different from the preceding research on TBFL approaches, our approach focuses on reducing efforts on test-result checking for TBFL.

Our previous work proposed an interactive approach for fault localization (Hao 2006; Hao et al. 2006, 2009), which aims at combining the benefits of TBFL approaches and manual debugging. Based on statements' suspicions, this approach se-

lects checking points (program execution points) for developers to inspect, and modifies suspicions according to the developers' estimation on the correctness of internal program states at checking points. Our new approach focuses on selecting tests for manual test-result checking rather than manual internal-program-state checking required by the previous interactive approach. Compared to manual test-result checking, manual internal-program-state checking is more challenging and less practical because the developers need to have deep knowledge of the internal program implementation to estimate the correctness of internal program states.

The Delta Debugging approach (Cleve and Zeller 2005) focuses on identifying the portion of one failed test that eventually causes the failure, and this approach has been demonstrated to be effective for finding the failure-causing state and helpful for fault localization. Besides these dynamic approaches (Agrawal et al. 1995; Cleve and Zeller 2005; Hao et al. 2005a; Jones and Harrold 2005; Jones et al. 2002; Liblit et al. 2003; Liu et al. 2005; Renieris and Reiss 2003), there are also static fault-localization approaches (Xie and Engler 2003), which usually focus on identifying anomalies in source code, which can often be used to infer faults.

Some other researchers focus on generating tests to locate faults. Wang and Roychoudhury (2005) proposed a technique to construct a "passed" execution for a faulty program based on a "failed" one and generate a bug report by the comparison of these two executions. Later, Guo et al. (2006) proposed to select a "passed" execution from a "passed" execution pool by their defined measurement and use this passed execution with the original failed one to generate a bug report. These two approaches only construct or choose a passed execution, whereas our approach selects tests without knowing whether they are passed or failed. Furthermore, when selecting new tests, our approach is not concerned about whether they are passed or failed.

The research most related to ours is done by Baudry et al. (2006), who proposed some diagnosis criteria on generating new tests to improve test suites for efficient fault localization. Their approach aims at using the criteria to guide the process of generating new tests to enhance the existing test suite, whereas our approach aims at obtaining a subset of the existing test collection. Specifically, our Strategy 1, which is also the starting point of our research, can be viewed as mapping their Test-for-Diagnosis criterion to our target problem. Besides Strategy 1, our approach includes two other strategies that consider more factors than Strategy 1 in selecting tests for result checking. According to our experimental results reported in Sect. 4, both Strategies 2 and 3 often win over Strategy 1 in fault-localization effectiveness, and Strategy 2 also wins over Strategy 1 in selecting fewer tests.

6 Conclusion and future work

In this paper we proposed a test-reduction approach (including three strategies), which selects some tests from the given test collection based on the execution traces of the test collection. Thus, developers check the results of only the selected tests and feed these selected tests to TBFL approaches. The experimental results show that the three strategies can help developers select a small subset of tests, which can still achieve effective fault-localization results.

In future work, we plan to investigate other test-reduction strategies based on other structural coverage such as branch coverage and condition coverage.

Acknowledgements This effort is sponsored by the National Basic Research Program of China (973) under Grant No. 2009CB320703, the Science Fund for Creative Research Groups of China under Grant No. 60821003, the National Natural Science Foundation of China under Grant No. 60803012, and the National Natural Science Foundation of China under Grant No. 90718016. Tao Xie's work is supported in part by NSF grants CCF-0725190, CCF-0845272, and ARO grants W911NF-07-1-0431 and W911NF-08-1-0105.

References

Agrawal, H., Horgan, J., London, S., Wong, W.: Fault localization using execution slices and dataflow. In: Proc. 6th International Symposium on Software Reliability Engineering, pp. 143–151, October 1995

Barbosa, E., Maldonado, J., Vincenzi, A.: Toward the determination of sufficient mutant operators for C. Softw. Test. Verif. Reliab. **11**(2), 113–136 (2001)

Baresi, L., Young, M.: Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, USA, August 2001

Baudry, B., Fleurey, F., Traon, Y.L.: Improving test suites for efficient fault localization. In: Proc. 28th International Conference on Software Engineering, pp. 82–91, May 2006

Cleve, H., Zeller, A.: Locating causes of program failure. In: Proc. 27th International Conference on Software Engineering, pp. 342–351, May 2005

Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Emp. Softw. Eng. **10**(4), 405–435 (2005)

Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: A family of empirical studies. IEEE Trans. Softw. Eng. **28**(2), 159–182 (2002)

Enderton, H.B.: Elements of Set Theory. Academic Press, San Diego (1977)

Graves, T.L., Harrold, M.J., Kim, J.-M., Porter, A., Rothermel, G.: An empirical study of regression test selection techniques. ACM Trans. Softw. Eng. Methodol. **10**(2), 184–208 (2001)

Guo, L., Roychoudhury, A., Wang, T.: Accurately choosing execution runs for software fault localization. In: Proc. International Conference on Compiler Construction, pp. 80–95, Mar. 2006

Hao, D.: Testing-based interactive fault localization. In: Proc. 28th International Conference on Software Engineering, Doctoral Symposium Track, pp. 957–960, May 2006

Hao, D., Pan, Y., Zhang, L., Zhao, W., Mei, H., Sun, J.: A similarity-aware approach to testing based fault localization. In: Proc. 20th IEEE International Conference on Automated Software Engineering, pp. 291–294 (2005a)

Hao, D., Zhang, L., Zhong, H., Mei, H., Sun, J.: Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study. In: Proc. 21st IEEE International Conference on Software Maintenance, pp. 683–686 (2005b)

Hao, D., Zhang, L., Mei, H., Sun, J.: Towards interactive fault localization using test information. In: Proc. 13th Asia Pacific Software Engineering Conference, pp. 277–284 (2006)

Hao, D., Zhang, L., Pan, Y., Mei, H., Sun, J.: On similarity-awareness in testing-based fault localization. Autom. Softw. Eng. J. **15**(2), 207–249 (2008)

Hao, D., Zhang, L., Zhang, L., Sun, J., Mei, H.: VIDA: Visual interactive debugging. In: Proc. 31st International Conference on Software Engineering, Formal Research Demonstrations, pp. 583–586, May 2009

Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. **2**(3), 270–285 (1993)

Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Proc. 16th International Conference on Software Engineering, pp. 191–200, May 1994

Jeffrey, D., Gupta, N.: Test suite reduction with selective redundancy. In: Proc. 21st IEEE International Conference on Software Maintenance, pp. 549–558, Sept. 2005

Jeffrey, D., Gupta, N.: Improving fault detection capability by selectively retaining test cases during test suite reduction. IEEE Trans. Softw. Eng. **33**(2), 108–123 (2007)

Jones, J.A., Harrold, M.J.: Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Softw. Eng. **29**(3), 195–209 (2003)
Jones, J.A., Harrold, M.J.: Empirical evaluation of tarantula automatic fault-localization technique. In: Proc. 20th International Conference on Automated Software Engineering, pp. 273–282 (2005)
Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proc. 24th International Conference on Software Engineering, pp. 467–477, May 2002
Joseph, N.: Statistics and Probability in Modern Life. Saunders, Philadelphia (1992)
Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Proc. ACM SIGPLAN 2003 Conference on Programming Languages Design and Implementation, pp. 141–154 (2003)
Liu, C., Yuan, X., Fei, L., Han, J., Midkiff, S.P.: SOBER: Statistical model-based bug localization. In: Proc. 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 286–295, September 2005
Ma, X.y., He, Z.f., Sheng, B.k., Ye, C.q.: A genetic algorithm for test-suite reduction. In: Proc. the International Conference on Systems, Man and Cybernetics, pp. 133–139, October 2005
Pacheco, C., Ernst, M.D.: Eclat: Automatic generation and classification of test inputs. In: Proc. 19th European Conference on Object-Oriented Programming, pp. 504–527 (2005)
Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Proc. 18th International Conference on Automated Software Engineering, pp. 30–39 (2003)
Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. **6**(2), 173–210 (1997)
Rothermel, G., Harrold, M.J.: Experimental studies of a safe regression test selection technique. IEEE Trans. Softw. Eng. **24**(6), 401–419 (1998)
Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. **27**(10), 929–948 (2001)
Rothermel, G., Harrold, M.J., von Ronne, J., Hong, C.: Empirical studies of test-suite reduction. Softw. Test. Verif. Reliab. **12**(4), 219–249 (2002)
Saff, D., Artzi, S., Perkins, J.H., Ernst, M.D.: Automatic test factoring for Java. In: Proc. 20th International Conference on Automated Software Engineering, pp. 114–123, November 2005
Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proc. 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 263–272 (2005)
Sprenkle, S., Sampath, S., Gibson, E., Pollock, L., Souter, A.: An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In: Proc. 21st IEEE International Conference on Software Maintenance, pp. 587–596, Sept. 2005
Srivastava, A., Thiagarajan, J.: Effectively prioritizing tests in development environment. In: Proc. 2002 International Symposium on Software Testing and Analysis, pp. 97–106, July 2002
Trochim, W.M.K.: Research Methods Knowledge Base. Thomson Custom Pub., New York (2007)
Walcott, K.R., Kapfhammer, G.M., Soffa, M.L., Roos, R.S.: Time-aware test suite prioritization. In: Proc. International Symposium on Software Testing and Analysis, pp. 1–12 (2006)
Wang, T., Roychoudhury, A.: Automated path generation for software fault localization. In: Proc. 20th International Conference on Automated Software Engineering, pp. 347–351 (2005)
Wohlin, C., Runeson, P., Host, M.: Experimentation in Software Engineering: An Introduction. Springer, Berlin (1999)
Xie, T., Notkin, D.: Tool-assisted unit-test generation and selection based on operational abstractions. Autom. Softw. Eng. J. **13**(3), 345–371 (2006)
Xie, Y., Engler, D.: Using redundancies to find errors. IEEE Trans. Softw. Eng. **29**(10), 915–928 (2003)
Yu, Y., Jones, J.A., Harrold, M.J.: An empirical study of the effects of test-suite reduction on fault localization. In: Proc. 30th International Conference on Software Engineering, pp. 201–210, May 2008