A Method and Tool Support for Automated Data Flow Testing of Java Programs

Van-Cuong Pham(✉) and Pham Ngoc-Hung

Faculty of Information Technology, VNU University of Engineering and Technology, Hanoi, Vietnam
{cuongpv.mi11,hungpn}@vnu.edu.vn

Abstract. This paper proposes a method and a tool support for automated data flow testing of Java programs. The key purpose of this method is to detect improper uses of data values due to coding errors. Given source code of a Java program, the proposed method analyzes and visualizes the program as a data flow graph. All test paths for covering all definition-use pairs of all variables are then generated. A test case corresponding to each generated test path is produced by identifying values to the input parameters so that the test path is executable. The expected outputs of these test cases are identified automatically. An implemented tool supporting the improved method and experimental results are also presented. This tool is promising to be applied in practice.

Keywords: Software testing · Data flow testing · White-box testing · Data flow anomaly · Data flow coverage.

1 Introduction

Software testing has been considered as the major solution in improving quality of software systems. Currently, software testing techniques can be divided into two kinds such as black-box testing and white-box testing. However, software companies focus only on the black box testing techniques in order to validate whether software products meet the customer requirements. By this approach, they only detect the errors/mistakes which can be observed by users. As a result, all potential errors of program code can be not detected. Moreover, detecting such errors has been recognized as a key difficult and expensive task in practice. In addition, the testers in charge this task are required high level knowledge and skills for analyzing source code. These issues are still open problems in software companies, especially in Vietnam.

Data flow testing has been known as a key white box testing technique that can be used to detect improper uses of data values due to coding errors [4]. These errors are inadvertently introduced in a program by programmers. For instance, a software programmer might use a variable without defining it, or he/she may define a variable, but not initialize it and then uses that variable in a predicate (e.g. int x; if(x==100);) [4]. The problem of errors in variables

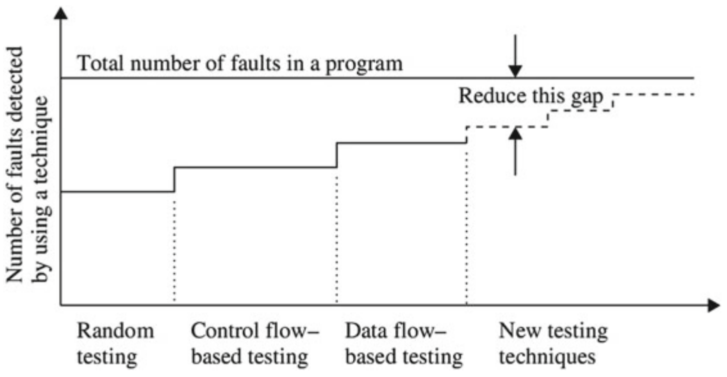


Fig. 1. Limitation of different fault detection techniques.

is common problems of programmers. In addition, Ntafos [10] has reported on the results of an experiment comparing with the effectiveness of three test selection techniques. The data flow testing, control flow testing, and random testing detected 90 %, 85.5 %, and 79.5 % respectively, of the known defects. Furthermore, Fig. 1 shows the limitation of different fault detection techniques [10]. These facts imply that data flow testing is one of the most effective methods for examining structure of programs. Although there are some methods and tools supporting data flow testing such as BPAS - ATCGS (Basic Program Analyzer System - Automatic Test Case Generation System) [8], JaBUTi [9], DFC (Data Flow Coverage) [3], etc, these methods only generate all paths for covering given source code. In fact, we need a tool that assists the tester in creating test data [5] that include expected output. Some free versions only allow testing the programs that are fixed in these tools and they are difficult to be extended in order to satisfy the specific data flow testing purposes of a certain software company.

This paper presents a method for data flow testing of Java programs. Given source code of a program, this method analyzes and visualizes the program as a data flow graph. All test paths corresponding to all paths of the data flow graph for covering all definition-use pairs of all variables in the program are then generated. All test cases of generated test paths are produced by giving values to the input parameters. The set of the values to the input parameters and expected outputs of the produced test cases are also generated automatically. In order to show the practical usefulness of the proposed method, a tool supports the method is implemented. The obtained experimental results by applying this tool for some typical programs are completely reliable in detecting all errors about using data variables. In addition, this tool is a free version, open source, and promising to be applied in practice.

The paper is organized as follows. We first review some background in Sect. 2. Section 3 describes a method for data flow testing of Java programs. Section 4 shows the implemented tool and experimental results. Finally, we conclude the paper in Sect. 5.

2 Background

This section presents some basic concepts which are used in our work as follows.

A definition of a variable x (denoted def) when the variable x is assigned a new value. When a variable x is used to compute in a statement, it is called a computation use (denoted *c-use*) of the variable x. Similarly, if the variable x is used in the predicate of conditional statement, it is called a predicate use (denoted *p-use*) of the variable x. A variable is defined in a statement and is used in another statement which may occur immediately or several statements after the definition called a definition-use (denoted *def-u*) pair of that variable.

Data flow graph(DFG) is a directed graph $G = \{N, E\}$, where N is a finite set of nodes and each node represents a *c-use* or def; E is a finite set of directed edges and each edge represents a *p-use*; $n_0, n_f \in N$ are entry node and exit node respectively. A path is a finite sequence of nodes connected by edges. A complete path is a path whose first node is the start node and whose last node is an exit node. A path is definition clear path (denoted *def-clear path*) w.r.t a variable if it contains no new definition of that variable. A test path is a path for covering a *def-u* pair of a variable in a program. A complete path is executable if there exists an assignment of values to input variables and global variables such that all the predicates evaluate to true, thereby making the path executable. Executable paths are also known as feasible paths [1]. For example, source code of a program and its data flow graph are shown in Fig. 2(a) and Fig. 2(b) respectively.

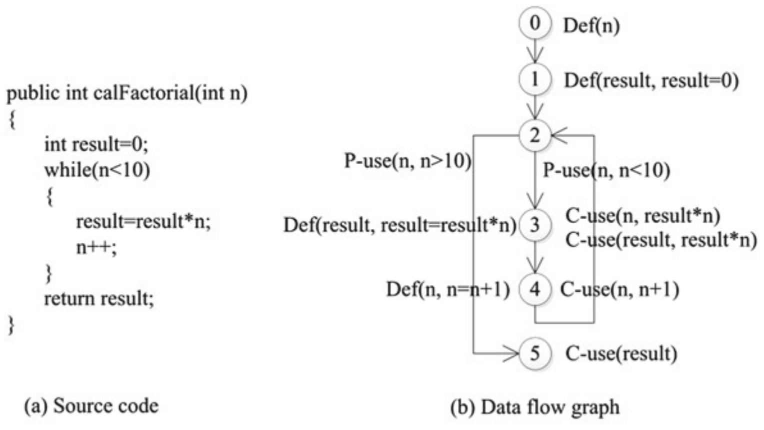

```
public int calFactorial(int n)
{
    int result=0;
    while(n<10)
    {
        result=result*n;
        n++;
    }
    return result;
}
```

Fig. 2. Source code and its data flow data flow graph.

Definition 1. *(DEF). A definition of a variable $v \in V$ at node $n \in N$, denoted $DEF(v, n)$, such that $DEF(v, n) = true$ if variable v is defined at node n and $DEF(v,n) = false$ otherwise.*

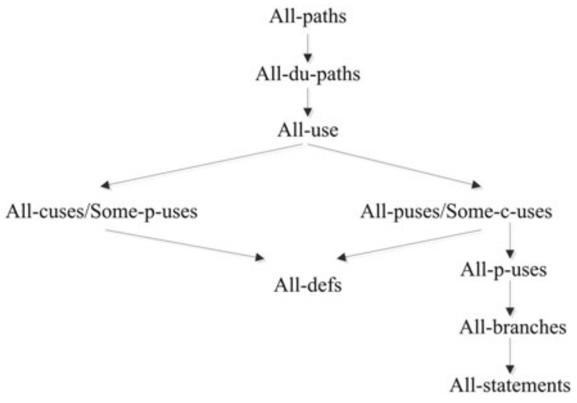


Fig. 3. Relationship among data flow testing criteria.

Definition 2. *(C-USE). A computation of a variable $v \in V$ at node $n \in N$, denoted C-USE(v,n), such that C-USE(v,n) = true if variable v is used to compute at node n and C-USE(v,n) = false otherwise.*

Definition 3. *(P-USE). A predicate of variable $v \in V$ at edge $e \in E$, denoted P-USE(v,e), such that P-USE(v,e) = true if the variable v is used at edge e and P-USE(v,e) = false otherwise.*

Definition 4. *Definition use path (du-path): A path $(n_1, ..., n_j, n_k)$ is a definition use path (denoted du-path) w.r.t. x if node n_1 has a def of x and either node n_k has a c-use of x and $(n_1, .., n_j, n_k)$ is def-c simple path w.r.t. x or edge (n_j, n_k) has a p-use of x and $(n_1, ..., n_j, n_k)$ is a def-c loop-free path w.r.t. x [1].*

Depending on the criterion selected, Frankl, Weyuker [1], and Parrish [2] have defined seven types of data flow testing criteria and relationship among them is shown in Fig. 3.

3 A Method for Data Flow Testing

This section presents a method for data flow testing of Java programs. Given source code of a program, this method analyzes and visualizes the program as a data flow graph. Next, finding all paths in the generated data flow graph for covering all *def-u* pairs of all variables in the program. Finally, all test cases corresponding to the generated test paths are created by giving values to the input parameters. The expected outputs of these test cases are also computed automatically.

Let U be a program, $V = \{v_1, v_2, .., v_n\}$ be a set of variables of U, $G = \{N, E\}$ be a data flow graph of U, and P be a set of du-path paths.

Algorithm 1. Finding all path for each variable in program

```
1: Initially, $P = empty$, $V$, $G$ {$P$ is a set du-paths of all variables, V is a set variables,
   G is a data flow graph}
2: for each variable $v$ in $V$ do
3:    for each node $m$ in $N$ do
4:       for each node $n$ in $N$ do
5:          if du-c(v,m,n)=true then
6:             $P_{def-c} = P_c(v, m, n)$
7:          end if
8:       end for
9:       for each edge $e$ in $E$ do
10:         if du-p(v,m,e)=true then
11:            $P_{def-c} += P_p(v, m, e)$
12:         end if
13:      end for
14:      for each path $p$ in $P_{def-c}$ do
15:         if $p$ is def-c path and $p$ is complete path then
16:            $P.add(p)$
17:         end if
18:      end for
19:   end for
20: end for
21: return
```

3.1 Test Path Generation

In this part, we propose a method for finding these paths. We are interested in finding all paths for covering all *def-u* pairs of all variables in a program U.

Definition 5. *(du-c). A variable $v \in V$ and node m and $n \in N$. du-c(v,m,n) = true if $DEF(v, m)$ = true and C-USE(v,n) = true else du-p(v,m,e) = false otherwise.*

Definition 6. *(du-p). A variable $v \in V$, $m \in N$, and $e \in E$. du-p(v,m,e) = true if $DEF(v, m)$ = true and P-USE(v,e) else du-p(v,m,e) = false otherwise.*

Definition 7. *(P_c). For each $v \in V$, $\forall m, n \in N$, if du-c(v,m,n) = true, existing a set of paths, denoted P_c, where $\forall p \in P_c$ has first node is m and last node is n.*

Definition 8. *(P_p). For each $v \in V$, $\forall m \in N$ and $\forall e \in E$ if du-p(v,m,e) = true, existing a set of paths, denoted P_p, where $\forall p \in P_p$ has first node is m and last edge is e.*

First, we will identify all paths which cover all *def-u* pairs of all variables. The set of these paths denotes P_{def-u}, where $P_{def-u} = P_c \cup P_p$. After that, examining all paths in P_{def-u}, $\forall p \in P_{def-u}$, if p is *def-c path* and *complete path*, then the path p is added into P.

The following is more detailed presentation of the method to find *test paths*. This method is shown in Algorithm 1. In this algorithm, we use an array data structure which contains the paths. These paths are generated by driving the path from a definition to a use of a variable in a program. Initially, the algorithm sets the array P to the *empty* (line 1). For each variable $v \in V$ (line 2), we identify a variable is defined and used at (line 5) and (line 10). Next, a set of paths is generated from a *def-u* pair (line 6 & 11). After having the set of path P, for each p in P, if p is *def-c path* and *complete path* (line 15), then add p into P (line 16). The algorithm iterates the entire process by looping from line 2 to line 20 until all variables in V are visited. The algorithm terminates and exits the program (line 21).

Depending on the criterion selected [1], we find the appropriate path for each data flow testing criteria. For example, with regard to the *All-defs* criteria, if $DEF(v, m) = true$, selecting a path $p \in P$ and path p has first node is m. Similarly, other data flow testing criteria are also identified by removing some inappropriate paths in P.

3.2 Test Case Generation

Test cases are generated by identifying the set of values to the input parameters and the set of expected outputs for each test path.

Definition 9. *(f_e). $\forall e \in E$, $\forall v \in V$, a boolean function, denoted fe, where fe:* $2^{|V|} \rightarrow \{true, false\}$.

For each $p \in P$, select a complete path p_c so that p_c contains path p. The set of edges in p_c that contains a *p-use* is $E_p = \{e | \textit{P-USE(v,e)} = true, e \in p_c, v \in V\}$. Let $Fe = \{f_e | \forall e \in E_p, f_e = \textit{boolean}\}$ be a set of boolean functions and $V' \in 2^{|V|}$ be a set of variable parameter. In order to p_c is an executable path, then we have to determine the values to the input parameters so that $f_e = true$, $\forall f_e \in Fe$. By improving the exhaustive search algorithm [7], the problems about generating test data have been solved partly. First, the Fe is divided into three main groups as follows.

$$Fe_1 = \{f_e(v) | v \in V', f_e(v) \in Fe, f_e(v) = C\}, \tag{1}$$

where $\forall fe \in Fe_1$, fe only has a variable and the sign of f_e is the sign of equality(=) and C is a constant.

$$Fe_2 = \{f_e(v) | v \in V', f_e(v) \in Fe, f_e(v) \neq C\}, \tag{2}$$

where $\forall fe \in Fe_2$, fe only has a variable and the sign of fe is different with the sign of equality and C a constant.

$$Fe_3 = \{f_e(V') | Length(V') \geq 2, f_e(V') \in Fe, f_e(V') + A = C\}, \tag{3}$$

where $\forall fe \in Fe_3$, fe has more than one variable, C is a constant, and A is a quantity which is added so that $fe(V') + A = C$.

Next, finding the solution of Fe_1 and Fe_2 and research spaces of Fe_3 are also the solution of Fe_1 and Fe_2. Finally, we use exhaustive research method [7] to identify final solution for whole Fe. For example, $p \in P$ has five expressions as follows: $5x = 10$, $x > 1$, $y > 1$, $y < 13$, and $x^2 + y^2 > 100$ in which x, y are parameters and $x, y \in [MIN, MAX]$. By applying the above principles, we have

$$Fe_1 = \begin{cases} 5x = 10 \\ x \in [MAX, MIN] \end{cases} \Rightarrow \begin{cases} x = 2 \end{cases}$$

After that, Let $x = \{2\}$ be a condition instead of $x = [MIN, MAX]$ in Fe_2 as follows.

$$Fe_2 = \begin{cases} x > 1 \\ y > 1 \\ y < 13 \\ x \in [MIN, MAX] \\ y \in [MIN, MAX] \end{cases} \Rightarrow \begin{cases} x > 1 \\ y > 1 \\ y < 13 \\ x = \{2\} \\ y \in [MIN, MAX] \end{cases} \Rightarrow \begin{cases} x = 2 \\ 1 < y < 13 \end{cases}$$

Finally, we use exhausted research method for Fe_3 where domain value of x is $\{2\}$ and domain value of y is (1, 13) as follows.

$$Fe_3 = \begin{cases} x^2 + y^2 > 100 \\ x = \{2\} \\ y \in (1, 13) \end{cases} \Rightarrow \begin{cases} x = 2 \\ 10 \leq y \leq 12 \end{cases}$$

The solution of Fe_3 as well as Fe is (2, 10), (2, 11), and (2, 12).

After obtaining the values to the input parameter, we will identify the expected output base on these values. Expected results are generated by using a mechanism called test oracle. For this purpose, we have used an alternate program for generating expected results. This means that using two different functions for the same problem, each function is described in different way but they return the same result when they set the same input values. Assume that P' is a program which is built so that $U(2^{|V|}) = U'(2^{|V|})$. U' is used to identify expected output. Table 1 shows test results is created from the test path p.

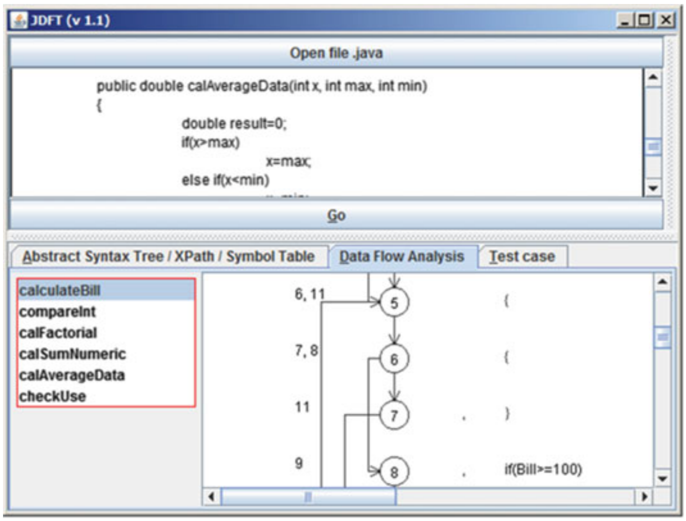
Table 1. A test case of U is created from the path p

No.	Path	Input	Expected output(P')	Actual output(P)	Pass
1	p=$(e_1, e_2, e_3, e_4, e_5)$	$x = 2$, $y = 10$	8	8	True
		$x = 2$, $y = 11$	9	9	True
		$x = 2$, $y = 12$	10	11	False

Similarly, we have to do the same for the other test paths and having a complete test suite for each data flow testing criteria.

4 Experiment and Discussion

This section presents our implemented tool for data flow testing and experimental results by applying this tool for some examples. We also discuss the advantages and disadvantages of the proposed method.



```
public double calAverageData(int x, int max, int min)
{
        double result=0;
        if(x>max)
                x=max;
        else if(x<min)
```

calculateBill
compareInt
calFactorial
calSumNumeric
calAverageData
checkUse

Fig. 4. Interface of the implemented tool.

Table 2. The test results of the examples for *All-du-paths* criteria

No.	Name	Number of paths	Predicate errors	Computation errors
1	calculateBill	26	7	3
2	calSumNumeric	14	3	0
3	calFactorial	11	4	2

4.1 Experiment

In order to show the effectiveness of the proposed method, we have implemented a tool, named JDFT. The tool is developed in Java and use a third party package name PMD (Programming Mistake Detector) [6], to analyze the given source code and display the generated data flow graph of the program. Figure 4 shows the user interface of JDFT. First, given source code of a Java program, this tool analyzes and visualizes the program as a data flow graph. Second, the JDFT finds all *def-u* pairs of all variables in the program. The test paths are then created based on finding the paths in data flow graph for covering *def-u* pairs. Third, this tool identifies the values to the input parameters which satisfy these test

Table 3. The test results of the examples for *All-du-paths* criteria

No.	Name	Number of paths	Predicate errors	Computation errors
1	calculateBill	28	0	0
2	calSumNumeric	16	0	0
3	calFactorial	13	0	0

paths and computing expected outputs of the test cases for the selected input. Finally, the test results are analyzed by comparing the actual output with the expected output of each test case.

We are also tested JDFT by using some typical examples in which the $calculateBill$ is used to calculate the bill of a cellular service, the $calSumNumeric$ is used to calculate the sum of integer divisible by 2 and the $calFactorial$ is used to calculate the factorial of n. There are some common errors related to improper uses of control structure statements and computed statements in the applied examples. All examples, JDFT tool, and help document are available at the site[1]. The results for *All-du-paths* of these examples are shown in Table 2. For the $calculateBill$, this tool detects 7 errors about using control structure statements and 3 errors about using computed statements. Similarly, for the $calSumNumeric$ is 3 and 0 and the $calFactorial$ is 4 and 2 respectively. After fixing errors in Table 2, we use the $JDFT$ to test these examples again and the obtained results of this purpose are shown in Table 3. There is not any error in the fixed programs detecting by the tool.

The above results are clear to show the practical usefulness of the implemented tool. In addition, this tool also can detect the unnecessary statements of the program under testing.

4.2 Discussion

The proposed method is a complete solution for automated data flow testing. It not only solves the key issues of white-box testing that generates test data, but also solves the problem about generating expected outputs.

Firstly, with regard to generate test data, dividing boolean functions into three main groups helps us to minimize the disadvantages of exhaustive method [7] that is running time. Generally, these functions will tend to fall into group 1 and group 2. Therefore, finding solution for these group is easy, whereas the boolean functions of group 3 is fewer and research space of this group is also solution of group 1 and group 2 hence calculation time is reduced significantly. In addition, although there are some methods to generate test data such as Genetic Algorithm [12], Ant Colony Algorithms [11], these methods are restricted about basic data types, while our method has solved all basic data types. However, it is still the restrictions as in some cases where the program includes complex boolean expressions, the calculation time is still high.

Secondly, for generating expected output, there are some automatic methods for generating expected output such as Statistical Oracle [13], Neural Network [14], and MT [15], but these methods are normally unavailable or too expensive to apply. By using an alternate program, this method is not only simple, but it is also easy to apply for software companies. Moreover, it also helps programmers have multiple perspectives on one problem, but this can also make workload of programmers as well as the volume of program code increase significantly.

[1] http://www.uet.vnu.edu.vn/uet/~hungpn/JDFT

Finally, comparing with other data flow testing tools has not been done, but the obtained experimental results are clear to show that the method is promising to be applied in industry.

5 Conclusion and Future Work

We have presented a method for data flow testing of Java programs. The key idea of this method is to generate all test cases such that they cover all *def-u* pairs of all variables used in the Java program under testing. The expected outputs of the generated test cases are also computed automatically by using an alternate program. A tool supports the proposed method is implemented in Java. Some typical examples are tested in order to show practical usefulness of the tool. The obtained experimental results are clear to show that the implemented tool can detect several common errors in coding.

Future work, we focus on solving the problem about test data, especially find out test paths which are not executable to remove unnecessary statements due to these statements is never executed. We are also investigating to apply some practical programs with their sizes are larger than the sizes of the applied systems in order to show the practical usefulness of the implemented tool. In addition, more experiments are needed in order to evaluate and compare the proposed method and the existing methods for data flow testing. Moreover, we also are going to apply this tool in some Vietnamese software companies.

Acknowledgments. This work is supported by the project no. QG.12.50 granted by Vietnam National University, Hanoi (VNU).

References

1. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Trans. Softw. Eng. **11**(4), 367–375 (1985)
2. Parrish, A.S., Zweben, S.H.: On the relationships among the all-uses, all-DU-paths, and all-edges testing criteria. IEEE Trans. Softw. Eng. **21**(12), 1006–1009 (1995)
3. Bluemke, I.: A coverage analysis tool for java programs. In: The 4th IFIP TC 2 Central and East European Conference on Advances in Software Engineering Techniques, pp. 215–228 (2009)
4. Copeland, L.: A Practitioner's Guide to Software Test Design. STQE Publishing, Massachusetts (2004)
5. Narmada, N., Mohapatra, D.P.: Automatic test data generation for data flow testing using particle swarm optimization. Commun. Comput. Inform. Sci. **95**(1), 1–12 (2010)
6. PDM Homepage, http://pmd.sourceforge.net/
7. Exhaustive research, http://en.wikipedia.org/wiki/Brute-force_search
8. BPAS-ATCGS Homepage, http://www.cs.ucy.ac.cy/~cs04pp2/WebHelp/index.htm
9. JaBUTi Homepage, http://jabuti.incubadora.fapesp.br (access, December 2007)

10. Ntafos, S.C.: On required element testing. IEEE Trans. Softw. Eng. **10**(6), 795–803 (1984)
11. Ghiduk, A.S.: A new software data-flow testing approach via ant colony algorithms. UniCSE (2010). ISSN: 2219–2158
12. Girgis, M.R.: Automatic test data generation for data flow testing using a genetic algorithm. J. UCS **11**(6), 898–915 (2005)
13. Mayer, J., Guderlei, R.: Test oracles using statistical methods. In: Proceedings of the First International Workshop on Software Quality, Lecture Notes in Informatics P-58, pp. 179–189 (2004)
14. Vanmali, M., Last, M., Kandel, A.: Using a neural network in the software testing process. Int. J. Intell. Syst. **17**(1), 45–62 (2002)
15. Hu, P., Zhang, Z., Chan, W.K., Tse, T.H.: An empirical comparison between direct and indirect test result checking approaches. In: Proceedings of the 3rd International Workshop on Software Quality ssurance (SOQUA06), pp. 6–13 (2006)