e-Informatica Software Engineering Journal, Volume 9, Issue 1, 2015, pages: 9–19, DOI 10.5277/e-Inf150101

Dataflow approach to testing Java programs supported with DFC

Ilona Bluemke[*], Artur Rembiszewski[*]
[*]*Institute of Computer Science, Warsaw University of Technology*
I.Bluemke@ii.pw.edu.pl, a.rembiszewski@gmail.com

Abstract

Code based ("white box") approach to testing can be divided into two main types: control flow coverage and data flow coverage methods. Dataflow testing was introduced for structural programming languages and later adopted for object languages. Among many tools supporting code based testing of object programs, only JaBUTi and DFC (Data Flow Coverage) support dataflow testing of Java programs. DFC is a tool implemented at the Institute of Computer Science Warsaw University of Technology as an Eclipse plug-in. The objective of this paper is to present dataflow coverage testing of Java programs supported by DFC. DFC finds all definition-uses pairs in tested unit and provides also the definition-uses graph for methods. After the execution of test information which def-uses pairs were covered is shown. An example of data flow testing of Java program is also presented.

1. Introduction

One of the key issues in developing software systems is effective testing. Popular approaches to testing include "black box" and "white box". Black-box and white-box testing are complementary to each other in the sense that they are likely to uncover different classes of faults. Black-box testing focuses on the functional requirements of the software. It aims at faults related to incorrect or missing functions, interface errors, behavior or performance errors and initialization and termination errors. White-box testing focuses on the internal structure of the program, to guarantee that all independent paths within a code have been executed at least once, exercise all logical decisions on their true and false sides, execute all loops at their boundaries and within their operational bounds and exercise internal data structures. White box approach can be divided into two main types: data flow coverage methods and control flow coverage. Control flow coverage methods were studied e.g. by Woodward and Hennell (2006) [1], Malevris and Yates (2006) [2].

The idea of data flow testing has been proposed in the seventies by Herman (1976) [3]. In this testing relationships between data are used to select the test cases.

Although experiments conducted in 1999 by Mei-Hwa Chen, Kao H.M. [4] show, that dataflow testing applied to object programs can be very effective this approach is not widely used for object programs. Among many tools supporting code based testing of object programs, only JaBUTi [5] supports dataflow testing of Java programs. At the Institute of Computer Science, Warsaw University of Technology, a tool, called DFC (Data Flow Coverage), for dataflow testing of Java program was implemented. DFC is implemented as an Eclipse plug-in so can be used with other testing tools available in eclipse environment.

The objective of this paper is to present dataflow coverage testing of Java programs supported by DFC. Introduction to dataflow approach and related work is given in section 2. DFC, presented in section 3, finds all *definition-uses* pairs in tested unit (section 2) and provides also the *definition-uses* graph for methods. After the execution of the test, the tester is provided with information which *def-uses* pairs were covered so she/he can add new tests for not covered pairs. The tester decides which methods are changing the state of an object. Such approach is novel and not available in other testing tools. In section 4 an example of Java program is used to explain the data flow coverage testing. Advantages and disadvantages of data flow testing of Java programs are also discussed. Section 5 contains some remarks.

2. Dataflow testing

In structural testing, also called "white box" testing, the tests are derived from the source code. Structural testing methods can be divided into two categories: code coverage and data flow coverage. Each of these techniques includes several criteria that define specific requirements that should be satisfied by the test set. Requirements determined by a testing criterion may be used either for the test set evaluation or the test set generation. In code coverage methods the test data may be chosen e.g. to execute all statements in the code (statement coverage), or to traverse all branches (branch coverage). Both control-flow and data-flow based testing criteria were originally defined to test procedural programs, they were also extended to object oriented programs. The adequacy of the testing activity is related to the determination of the effectiveness of a test criterion. Test effectiveness is related to the task of creating the smallest test set for which the output indicates the largest set of failures. The idea of data flow testing has been introduced by Herman (1976) [3]. Later, it was also studied by Laski and Korel (1983) [6]. Rapps and Weyuker (1985) [7] showed how to select test data using data flow information. Ostrand and Weyuker (1991) [8] were analyzing the test adequacy in data flow testing. Further research in data flow testing was also made by e.g. Harrold and Rothermel (1994) [9], Harold and Soffa (1989) [10], Vincenzi, Maldonado, Wong and Delamaro (2005) [11], Laski and Stanley (2009) [12]. Quite recently new methods were proposed by Chaim and de Araujo (2013, 2014) [13,14] and Vivanti (2014) [15].

In data flow testing the relations between data are the basis to design test cases. Different sub-paths from *definition* of a variable (assignment) into its *use* are tested.

A definition-use pair (*def-u*) is an ordered pair (d, u), where d is a statement containing definition of a variable v, and u a statement containing the use of v or some memory location bound to v that can be reached from d over some program path.

Test criteria are used to select particular definition-use pairs. A test satisfies a *def-u* pair, if executing the program with this test causes traversal of a sub-path from the definition to the use of this variable v without any v redefinition. A *def-u* pair is feasible if there exists some program input being able to exercise the pair.

Data-flow testing criteria proposed by Rapps and Weyuker [7] use the *def-use* graph (DUG), which is an extension of the control flow graph (CFG) with information about the set of variables defined – *def()* and used – *use()* in each node/edge of the CFG. An example of DUG for code in Listing 1 is presented in Figure 1. A program can be uniquely decomposed into a set of disjoint blocks having the property that if the first statement of the block is executed, the other statements are executed in the given order. The first statement of the block is the only statement which may be executed directly after the execution of a statement in another block. The last statement is the only one which may have a successor in the execution outside the block. Every conditional transfer must be the last statement of a block. The program graph G representing a program consists of one node i corresponding to each block b_i of the program and an edge from node j to node k, denoted *(j, k)*, if and only if either the last statement of b_i is not an unconditional transfer and it physically precedes the first statement of b_k, or the last statement of b_i is a transfer whose target

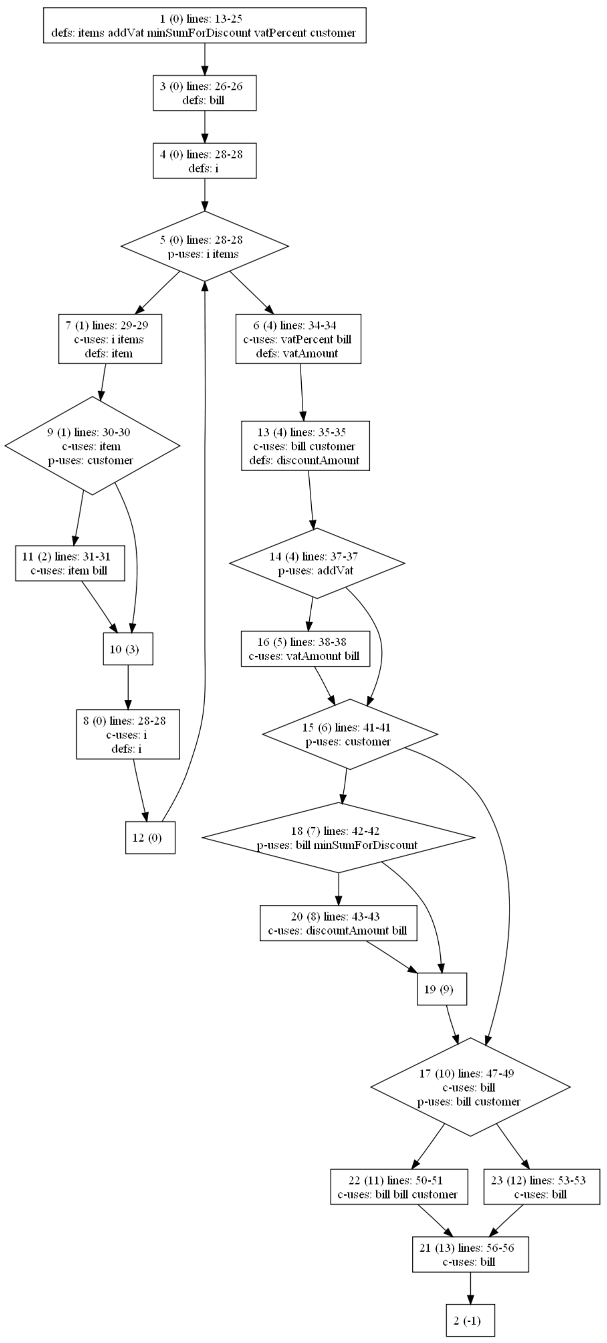


Figure 1. DUG for code given in Listing 1

is the first statement of b_k. The node correspondingto the block whose first statement is the start statement of the program is the start node and has no predecessors. A node corresponding to a block whose final statement is a halt statement is known as an exit node and has no successors. In addition, a node has two successors if and only if, the final statement of its corresponding block is a conditional transfer. All transfer statements should be effective, it means that two successors are different nodes. For every pair of nodes i and j, there is at most one edge from node i to node j. A path can be represented as a finite sequence of nodes.

In data flow testing the path selection criteria are based on an investigation of the ways in which values are associated with variables, and how these associations can affect the execution of the program. This analysis focuses on the occurrences of variables within the program. Each variable occurrence is classified as being a definitional (*def*), computation-use (*c-use*), or predicate-use (*p-use*) occurrence.

The *def/use* graph (DUG) is constructed from a program graph by associating with each node i the set of variables for which this node i contains a definition *def(i)* and the *c-use(i)* (the set of variables for which node i contains a *c-use*). The edge *(i, j)* of DUG is associated with *p-use(i, j)* (the set of variables for which edge *(i, j)* contains a *p-use).*

Many *def-u* criteria have been proposed and compared. First criteria introduced by Rapps and Weyuker contain e.g.: *all-nodes, all-edges, all-defs, all-du-paths, all-p-uses, all-c-uses/some-p-use.*

The criterion, called *all-defs* states, that for each DUG node i and all variables v, $v \in def(i)$ (defined in this node) at least one path *(i, j)* is covered. In node j this variable is used $v \in use(j)$ and on this path the variable v is not redefined.

The decision of which criterion to use as a basis for test data selection depends on several factors, including the size of the program, time and cost requirements and consequence of failure. The "stronger" the selected criterion, the more closely the program is scrutinized in an attempt to locate program faults but a "weaker" criterion can be fulfilled, using fewer test cases. The criteria *all-nodes* (statement coverage) and *all-edges* (branch coverage) are often used in program testing despite the fact that it is well known that they are weak criteria. Certainly they represent necessary conditions, for if some portion of the program has never been executed, one would not in general feel confident about its behavior. A similar intuition motivated Rapps and Weyuker in the definition of *all-defs* criterion. Even if every statement and branch had been executed, if the result of some computation had never been used, one would have little evidence that the intended computation had been performed.

The *all-uses* criterion requires that test data force some path to be traversed between every definition and each of its uses. Stronger requirement is that test data cause every path between a definition and its uses to be traversed. If the program contains loops, there may be infinitely many such paths. Rapps and Weyuker "strongest" criterion, *all-du-paths*, requires that test data cause the traversal of every *du-path* between a definition and each of its uses, thus avoiding this problem. Rapps and Weyuker also proved inclusion between the test criteria shown in Figure 2.

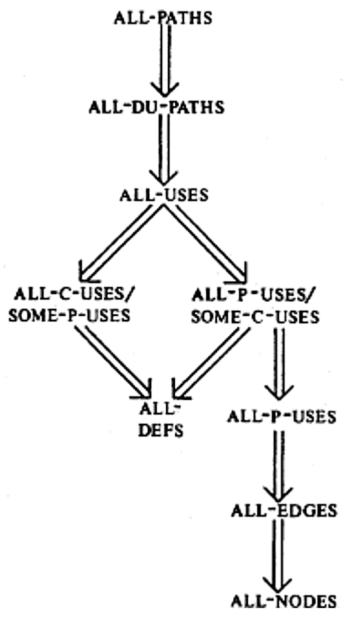


Figure 2. Inclusion of data flow test criteria (Rapps and Weyuker 1985 [7])

The dataflow technique described above was dedicated to structural programming languages and does not consider dataflow interactions that arise when methods are invoked in an arbitrary order. Harold and Soffa elaborated in 1989 [10] the inter–procedural data flow testing. They proposed an algorithm, called PLR, to find *def-u* pairs if the variable definition is introduced in one procedure, and the variable usage is in called or calling procedures. The algorithm works on inter–procedural control flow graph built from control flow graphs of dependent procedures. A call site is replaced by a call and a return node. The control flow graphs are connected by added edges from the call node to the entry nodes and from exit nodes to the return nodes to represent procedure calls in the program. A special entry node represents the entry to the "main" procedure of the program. The PLR algorithm first computes the definition and alias information for each procedure. Then, using the dataflow framework, propagates the local information to obtain inter–procedural reaching definitions from which inter–procedural *def-use* pairs can be calculated. This method can be adapted to global variables, class attributes and referenced method arguments in testing object programs. The *def-use* pairs can be used to test the possible interactions between methods. Data flow approach to test classes gives opportunities to find errors in classes that may not be uncovered by functional testing.

For object programs Harrold and Rothermel proposed in 1994 [9] three levels of dataflow testing:

- *Intra-method* level is based on the basic Rapps and Weyuker algorithm, is performed on each method individually; class attributes and methods interactions can not be taken into account. This level of testing is equivalent to unit testing in procedural language programs.
- *Inter-method* tests are applied to public method together with other methods in its class that it calls directly or indirectly. *def-u* pairs for class attributes can be found in this approach. This level of testing is equivalent to integration testing of procedures in procedural language programs.
- *Intra-class* – interactions of public methods are tested, when they are called in various sequences. The set of possible public methods calls sequences is infinite so only a subset of it is tested. Since users of a class may invoke sequences of methods in indeterminate order the *intra-class* testing can increase the confidence that sequences of calls interact properly.

For each of the above described testing levels appropriate *def-u* pairs were defined by Harrold and Rothermel (1994) [9] i.e. *intra-method, inter-method* and *intra-class*.

3. DFC – a tool for data flow testing

The process of testing software is extremely expensive in terms of labour, time and cost so many tools supporting this process have been developed but we found only one – JaBUTi [5], dedicated to the dataflow testing of Java programs (when we started the research in 2008). Dataflow testing of object programs can reveal many errors. An experiment described by Mei-Hwa Chen and Kao in 1999 [4], shows, that in the dataflow testing of C++ programs, the number of detected errors was four times greater, than in other code coverage methods i.e. instructions and conditions coverage. The results of this experiment motivated us to build a tool for the dataflow testing of Java programs.

Dataflow testing can't be applied in isolation so we decided to implement a tool supporting this approach, DFC – Data Flow Coverage (Figure 3), as an Eclipse plug-in. In Eclipse Java programming environment and testing tools e.g. JUnit are available. DFC finds all *def-u* pairs in testing Java code and after the test provides the tester information which *def-u* pairs were covered. Based on this information tester can decide which coverage criteria should be used and add appropriate test cases (shown in section 4). In preparing the test cases the tester can also use *def-use* graph (DUG) for a method provided by DFC.

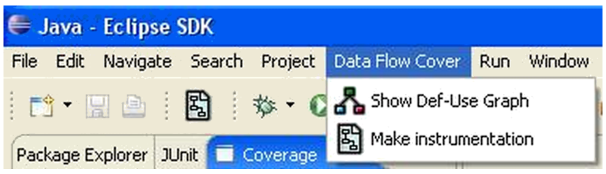

Java - Eclipse SDK
File Edit Navigate Search Project Data Flow Cover Run Window
Show Def-Use Graph
Make instrumentation
Package Explorer JUnit Coverage

Figure 3. DFC menu

In object languages the dataflow testing ideas proposed for structural languages must be modified. One of the main problems which must be solved is the identification which method is able to modify the object state and which one is using it only. In DFC *def-u* pairs are *intra-method*. Definitions of class attributes are located in the first node of DUG graph of tested method. The first node of DUG also contains definitions of arguments of the tested method.

Definitions of variable x are e.g.:

1. `int x; Object x; x = 5; x = y; x = new Object(); x=get_object(param);`
2. **x** is an object and a state modifying method is called in its context:
   `x.method1();`
3. **x** is an object and one of its attributes is modified:
   `x.a = 5;`

An instruction **uses a variable x** e.g.:

1. its value is assigned:
   `w = 2*x; x++;`
2. **x** is an object and a reference is used in an instruction:
   `w=x; method1(x); if (x == null)`
3. **x** is an object and a method using state of this object is called in its context:
   `x.method1();`
4. **x** is an object and one of its attributes is used in an instruction:
   `w = 2*x.a;`

In DFC the tester can decide which method is defining and which one is using the object state.

In Figure 4 the main parts of DFC and its collaboration with Eclipse environment are presented. The modules of DFC are denoted by bold lines.

The input for DFC is the Java source code (*SRC* in Figure 4). DFC user has to identify which file will be tested and indicate it to DFC. Module *Knowledge base* analyses the source code and generates the list of classes and methods. On this list tester may mark methods as modifying or using object state. The module *Instrumentation* instruments source code i.e. adds extra instructions needed for finding dataflow coverage and builds *def-use* graph (DUG). DUG (example for source code from Listing 1 is shown in Figure 1) contains information concerning the control flow, variable definitions and usage in its nodes. DUG is the input for module *Visualization*, drawing the graph, and *Requirements* – finding all *def-u*

Table 1. Test cases for method `doShopping`

Method	Test Cases						
Name	1	2	3	4	5	6	7
`addVat`	false	true	false	false	true	false	false
`minSumForDiscount`	200	200	20	200	20	200	200
`vatPercent`	20	20	20	20	20	20	20
`customer.needItems`	item1, item5	item1, item5	item1, item5	item1, item5	item1, item5	item1, item5	
`customer.getDiscountPercent ()`	10	10	10	10	10	10	10
`customer.getMoneyAmount()`	100	100	100	100	100	50	100
`customer.isSpecial()`	false	false	true	true	true	false	false

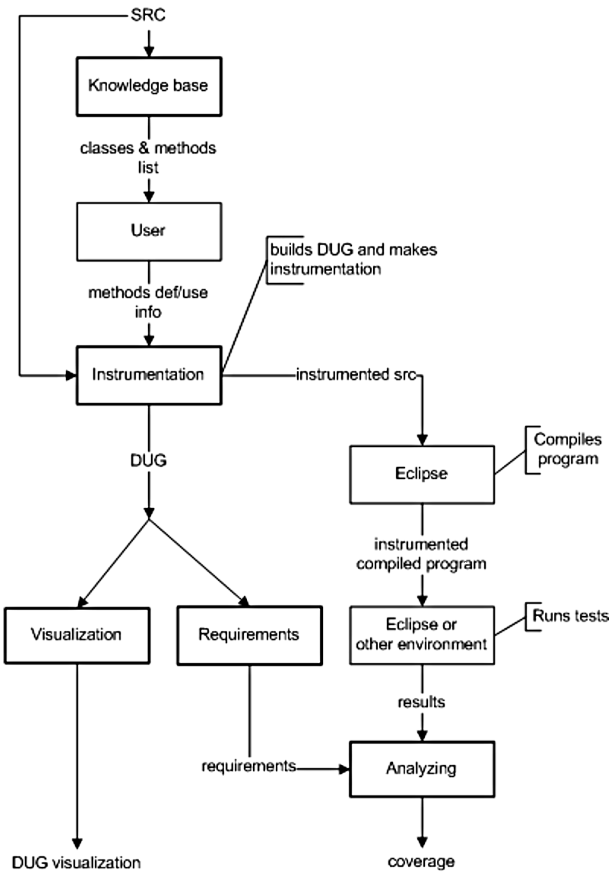


Figure 4. The idea of testing with DFC

pairs. The constructed DUG graph is presented after pressing `Show DUG` button in the DFC menu (Figure 3). Instrumented code should be compiled and run in the Eclipse environment and for these activities the programmer is responsible. During the test execution, extra code added by *Instrumentation* module sends data concerning the coverage to DFC. Module *Analyzing* is locating covered and not covered *def-u* pairs. More details on DFC implementation and its usage were described by Rembiszewski (2009) [16] and by Bluemke, Rembiszewski (2012) [17].

4. Example

In this section the dataflow testing of Java code is presented in a small example. In Listing 1 the source code of the method `doShopping` from class `Shop` is shown. This method was tested with the DFC tool. The DUG for this method is shown in Figure 1. The *all-defs* coverage criterion was used. In Table 1 the test cases are listed. Column "Name" contains the name of variable or method returning the private attribute. Following columns contain the values used in test cases. In objects of class `Item` the method `getPrice` returns `10` for variable `item1` and `50` for `item5`. Testing of code, shown in Listing 1, was performed in two phases. In the first phase the *def-use* chains were covered. In DFC all methods were initially identified as not modifying the state of object and using it (similarly as in JaBUTi [5]).

Listing 1 Method `doShopping` from class `Shop`

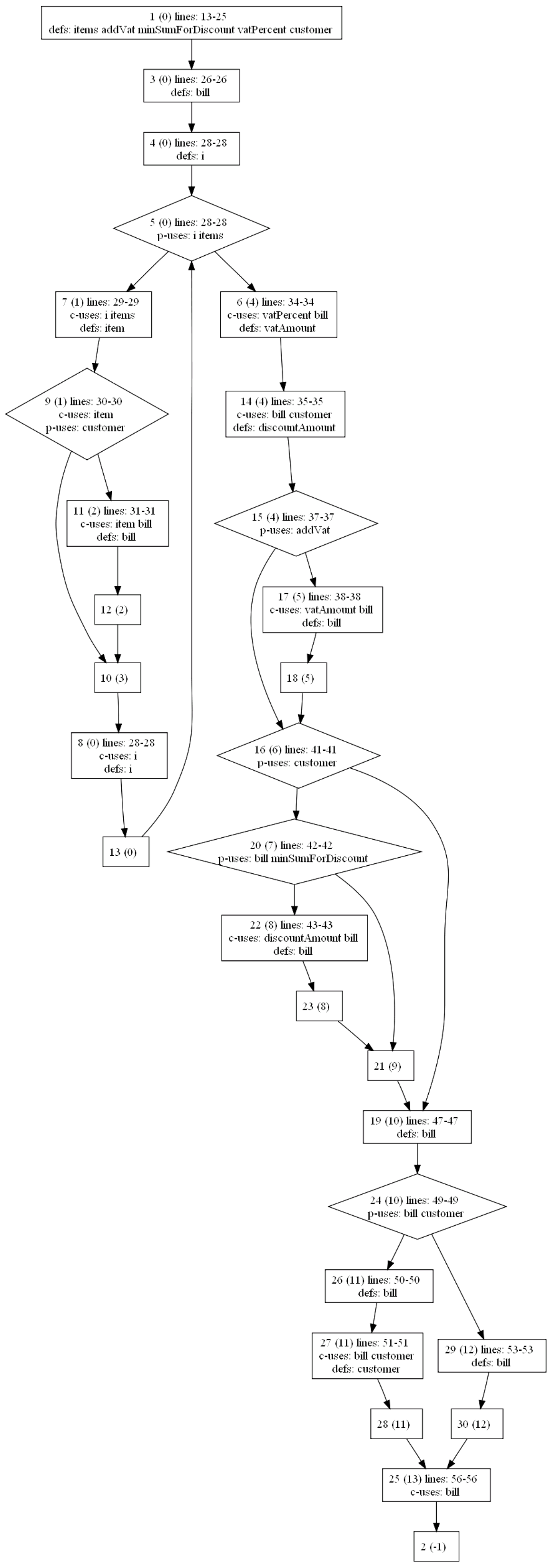
```
25) public Bill doShopping(Customer customer) {
26)     Bill bill = new Bill();
27)
28)     for (int i=0; i<items.size(); i++) {
29)     Item item = items.get(i);
30)     if (customer.need(item))
```

```
31)         bill.add(item.getPrice());
32)     }
33)
34)     double vatAmount = (vatPercent/100) *
                            bill.getTotalSum();
35)     double discountAmount = bill.getTotalSum() *
          (customer.getDiscountPercent()/100);
36)
37)     if (addVat) {
38)         bill.add(vatAmount);
39)     }
40)
41)     if (customer.isSpecial()) {
42)         if (bill.getTotalSum() > minSumForDiscount) {
43)             bill.subtract(discountAmount);
44)         }
45)     }
46)
47)     bill.close();
48)
49)     if (bill.getTotalSum() <= customer.getMoneyAmount()) {
50)         bill.pay();
51)         customer.getFromAcount(bill.getTotalSum());
52)         } else {
53)         bill.cancel();
54)     }
55)
56)     return bill;
57) }
```

When the full coverage of *all-defs* pairs was achieved, DFC was reconfigured; the methods modifying state of object and using it were marked as shown in Figure 5 (such functionality is not available in JaBUTi [5]). The DUG graph after this modification is presented in Figure 6.

After the re–execution of tests new test cases, for not covered *def-use* pairs were added. The test results are given in Table 2.

Column "Definition" shows the name and line number containing the definition. If a line contains two definitions (e.g. line 28) of the same variable, after comma the column of definition is given. Definitions 1–11 were found in the first phase, while definitions 12–18 in the second one. It can be noticed, that definitions found in the first phase are the subset of definitions found in the second phase. Column "Test cases" *n*, contain

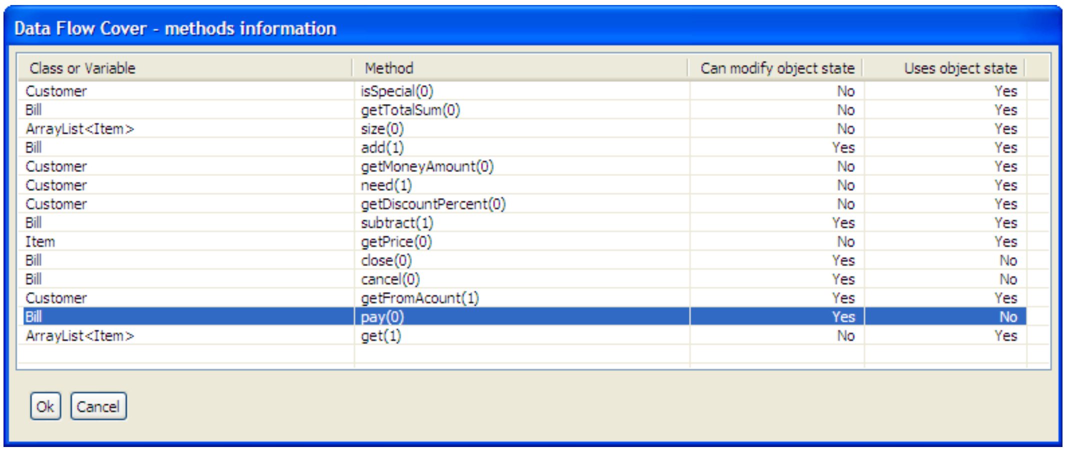


1 (0) lines: 13-25 defs: items addVat minSumForDiscount vatPercent customer
3 (0) lines: 26-26 defs: bill
4 (0) lines: 28-28 defs: i
5 (0) lines: 28-28 p-uses: i items
7 (1) lines: 29-29 c-uses: i items defs: item
6 (4) lines: 34-34 c-uses: vatPercent bill defs: vatAmount
9 (1) lines: 30-30 c-uses: item p-uses: customer
14 (4) lines: 35-35 c-uses: bill customer defs: discountAmount
11 (2) lines: 31-31 c-uses: item bill defs: bill
15 (4) lines: 37-37 p-uses: addVat
12 (2)
17 (5) lines: 38-38 c-uses: vatAmount bill defs: bill
10 (3)
18 (5)
8 (0) lines: 28-28 c-uses: i defs: i
16 (6) lines: 41-41 p-uses: customer
13 (0)
20 (7) lines: 42-42 p-uses: bill minSumForDiscount
22 (8) lines: 43-43 c-uses: discountAmount bill defs: bill
23 (8)
21 (9)
19 (10) lines: 47-47 defs: bill
24 (10) lines: 49-49 p-uses: bill customer
26 (11) lines: 50-50 defs: bill
27 (11) lines: 51-51 c-uses: bill customer defs: customer
29 (12) lines: 53-53 defs: bill
28 (11)
30 (12)
25 (13) lines: 56-56 c-uses: bill
2 (-1)

Figure 6. DUG for code in Listing 1 after the indication of methods changing object state

Table 2. The results of tests for method `doShopping`

Definition			Test Cases					
	Name	Line	1	2	3	4	**5**	6
1	`items`	13	Y(29)	Y(29)	Y(29)	Y(29)	Y(29)	Y(29)
2	`addVat`	14	N	Y(37)	Y(37)	Y(37)	Y(37)	Y(37)
3	`minSumForDiscount`	15	N	N	N	Y(42)	Y(42)	Y(42)
4	`vatPercent`	16	Y(34)	Y(34)	Y(34)	Y(34)	Y(34)	Y(34)
5	`customer`	25	Y(30)	Y(30)	Y(30)	Y(30)	Y(30)	Y(30)
6	`bill`	26	Y(31)	Y(31)	Y(31)	Y(31)	Y(31)	Y(31)
7	`i`	28,11	Y(28)	Y(28)	Y(28)	Y(28)	Y(28)	Y(28)
8	`i`	28,32	Y(28)	Y(28)	Y(28)	Y(28)	Y(28)	Y(28)
9	`item`	29	Y(30)	Y(30)	Y(30)	Y(30)	Y(30)	Y(30)
10	`vatAmount`	34	N	Y(38)	Y(38)	Y(38)	Y(38)	Y(38)
11	`discountAmount`	35/39	N	N	Y(43)	Y(43)	Y(43)	Y(43)
12	`bill`	31	Y(34)	Y(34)	Y(34)	Y(34)	Y(34)	Y(34)
13	`bill`	38/37	N	N	N	N	Y(43)	Y(43)
14	`bill`	43	–	–	–	–	–	–
15	`bill`	47	N	N	N	N	N	Y(49)
16	`bill`	50	Y(51)	Y(51)	Y(51)	Y(51)	Y(51)	Y(51)
17	`bill`	53	N	N	N	N	N	Y(56)
18	`customer`	51	–	–	–	–	–	–
Coverage – phase 1			64%	82%	91%	100%	100%	100%
Coverage – phase 2			56%	69%	75%	81%	87%	100%



Class or Variable	Method	Can modify object state	Uses object state
Customer	isSpecial(0)	No	Yes
Bill	getTotalSum(0)	No	Yes
ArrayList<Item>	size(0)	No	Yes
Bill	add(1)	Yes	Yes
Customer	getMoneyAmount(0)	No	Yes
Customer	need(1)	No	Yes
Customer	getDiscountPercent(0)	No	Yes
Bill	subtract(1)	Yes	Yes
Item	getPrice(0)	No	Yes
Bill	close(0)	Yes	No
Bill	cancel(0)	Yes	No
Customer	getFromAcount(1)	Yes	Yes
Bill	pay(0)	Yes	No
ArrayList<Item>	get(1)	No	Yes

Figure 5. Configuration screen in DFC

letter *Y*, if this definitions is in covered *def-use* pairs in test cases *1* or *2* or ..., *n*. In brackets the line number of the covered usage is given. Letter *N* in column of Table 2 means, that the covered pair does not contain this definition and char "–" is written, if reachable usage for this definition does not exist.

In the last two rows (Table 2) the dataflow coverage for the two phases are calculated. This coverage was calculated as the percentage of covered pairs to all, possible to cover pairs. The first phase needed the first four test cases to cover all *def-use* pairs. In the second phase the methods modifying the state of object and using it were marked manually in DFC on a screen shown in Figure 5. The same four test cases produced the coverage only 81% so some new test cases have to be constructed. The **test case number 5** revealed an error in `doShopping` method. In this test case `vat` is added to the bill and the client receives also a discount. In the code (Listing 1) the discount is calculated before the `vat` is added instead of being calculated after. The modified code is given in Listing 2. We tried not to change the line numbering as much as possible. In Table 2, in rows 11 and 13, the new line numbers are following slash character. After the modification all test were re–executed and another test case, **number 6**, was added to obtain the 100% coverage. In this example we showed, that the identification which method is modifying the object's state forced the tester to add a test case revealing an error.

In the code given in Listing 1, for definition of `bill` in line 43 and `customer` in line 51 no reachable usage exist. According to dataflow coverage rules applied to structural languages such situation can be seen as anomalies in the code. In object programs such situation is not an indicator of code anomalies. The `bill` is also defined in line 47. For simple variable two successive assignments are incorrect, the second one, erases the first one. For an object variable successive assignments e.g. changing object's state, may be reasonable. The `customer`, defined in line 51 is not used in the method. This is not an indicator of code anomalies because `customer` is referenced argument of `doShopping` method and can be used outside this method.

The success of testing (phase 2) strongly depends on the correct identification of methods defining and using the object's state. The test cases 1–6 from Table 1 are not testing program execution if `bill` is empty. This is tested in test case 7. This test case would be executed if the method `adds` in class `Bill` is marked as modifying object but not using it. To cover the definition of `bill` in line 26 (Listing 1 and Table 2) instruction in line 34 should be executed but the redefinition in line 31 should be skipped. In this example the simple coverage criteria *all-defs* was used. Other coverage criteria e.g. *all-uses* can reveal the error in the method `doShopping` without manually setting which method is modifying, and which one is only using the state of object.

Listing 2 Modified method `doShopping`

```
25) public Bill doShopping(Customer customer) {
26)     Bill bill = new Bill();
27)
28)     for (int i=0; i<items.size(); i++) {
29)        Item item = items.get(i);
30)        if (customer.need(item))
31)        bill.add(item.getPrice());
32)     }
33)
34)     double vatAmount = (vatPercent/100) *
                       bill.getTotalSum();
35)
36)     if (addVat) {
37)        bill.add(vatAmount);
38)     }
39)     double discountAmount = bill.getTotalSum() *
                (customer.getDiscountPercent()/100);
40)
41)     if (customer.isSpecial()) {
42)        if (bill.getTotalSum() > minSumForDiscount) {
43)        bill.subtract(discountAmount);
44)        }
45)     }
46)
47)     bill.close();
48)
49)     if (bill.getTotalSum() <=
                   customer.getMoneyAmount()) {
50)     bill.pay();
51)     customer.getFromAcount(bill.getTotalSum());
52)     } else {
53)       bill.cancel();
54)     }
55)
56)     return bill;
57)  }
```

4.1. DFC and JaBUTi

In JaBUTi [5] every call of a method in the context of an object variable is treated as using object state. In DFC a method call is treated as using the object state if the state of object variable is not changed. Let's look at a small example given in Listing 3.

Listing 3 Simple example

```
1) a = new Object();
2) if(....)
3) a.setState(...);
4) a.m();
```

JaBUTi will not notice the coverage of *def-use* pair in lines (3,4). In DFC it is possible, if the `setState` method is correctly indicated as modifying the object state. This simple example shows, that DFC is able to treat more instructions as defining, thus is able to show the coverage of greater number of *def-use* pairs and more errors can possibly be detected.

5. Conclusions

Many authors e.g. Beizer [18] suggest that effective testing can be achieved if different testing approaches e.g. functional and structural are used. In the development of software systems thorough testing can be the crucial issue. In this paper we presented DFC, an Eclipse plug-in, designed and implemented at the Institute of Computer Science Warsaw University of Technology, supporting dataflow testing of Java methods. By supporting dataflow testing of Java classes we provide opportunities to find error that may not be uncovered by black box testing. In Eclipse environment there are other tools available for testing Java programs using different techniques e.g. JUnit [19], EclEmma [20] or TPTP [21]. EclEmma provides information about instruction coverage. In DFC tester can design tests to achieve e.g. *def-uses* or *all-uses* coverage criteria which also guarantee instruction coverage (as proved by Rapps and Weyuker in 1985 [7]).

It has been shown that the data flow is an effective testing technique (e.g. in 1994 Hutchins et.al. [22]), very useful to fault localization (e.g. in 2009 Santelices et.al. [23]) but it is not used in industry. This phenomenon can be explained by the fact, that tools supporting data flow testing are not scalable for large systems due to the costs associated with tracking *def-u* associations at the run time. Recently (2013), Chaim and de Araujo [13] proposed a novel algorithm, called Bitwise Algorithm (BA) to tackle this problem. The new algorithm utilizes efficient bitwise operations and data structures to track the intra procedural *def-u*. They also showed, that BA is at least as good as the most efficient data flow instrumentation techniques, and that it can be up to 100% more efficient. In 2014 de Araujo and Chaim [14] presented the BA implementation for programs compiled into byte codes. Maybe theirs results will encourage vendors to consider including data flow testing in commercial testing tools.

In 2011 Bluemke and Kulesza [24] compared the dataflow and the mutation testing of several Java programs. Experiments were conducted in the Eclipse environment. DFC plugin was used to support the dataflow testing while MuClipse [25] and Jumble [26] plugins were used for the mutation testing. The results of testing six Java programs using data flow and mutation techniques shown, that the effectiveness of mutation testing is higher than the effectiveness of dataflow testing. Mutation technique appeared also to be more expensive than the data flow one, if time and effort are considered.

Finally, we outline the direction for the future research. An interesting and important study would be to apply DFC to industry projects to evaluate the cost and benefits of dataflow based criteria in testing Java programs. Unfortunately during several years we are not able to find interest on this subject in software industry (the tool is available for free and only three researchers from university downloaded it, none from industry). One of the reasons may by the effort needed in this approach. In DFC tester manually identifies defining and using methods (Figure 4). However this process is time consuming, we are not going to make it automatically. To identify, if a method is defining or using object state, the analysis of the source code must be performed. In complex, industry programs, many libraries are used so the access to the source code is limited. Decompilation of the library code preceding the analysis process, or the comparison of the value returned by `hashCode()` before and after the method call (this approach needs additional code instrumentation and re–execution of test cases) might be the solution. Comparing the effort needed with the possible results obtained, we think it is not worthy to implement these approaches. In JaBUTi [5], other tool supporting dataflow testing of Java program, every call of a method is treated as using object state. In section 4 we have demonstrated by example, that for some programs the identification of methods defining object's state enables to find more errors.

References

[1] M. R. Woodward and M. A. Hennell, "On the relationship between two control-flow coverage criteria: all jj-paths and mcdc," *Information and Software Technology*, Vol. 48, No. 7, 2006, pp. 433–440.

[2] N. Malevris and D. F. Yates, "The collateral coverage of data flow criteria when branch testing," *Information and Software Technology*, Vol. 48, No. 8, 2006, pp. 676–686.

[3] P. Herman, "A data flow analysis approach to program testing," *Australian Computer Journal*, Vol. 8, No. 3, 1976, pp. 92–96.

[4] M.-H. Chen and H. M. Kao, "Testing object-oriented programs-an integrated approach," in *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on.* IEEE, 1999, pp. 73–82.

[5] Jabuti homepage. (Accessed 12.2007). [Online]. http://jabuti.incubadora.fapesp.br/

[6] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *Software Engineering, IEEE Transactions on*, No. 3, 1983, pp. 347–354.

[7] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *Software Engineering, IEEE Transactions on*, No. 4, 1985, pp. 367–375.

[8] T. J. Ostrand and E. J. Weyuker, "Data flow-based test adequacy analysis for languages with pointers," in *Proceedings of the symposium on Testing, analysis, and verification.* ACM, 1991, pp. 74–86.

[9] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *ACM SIGSOFT Software Engineering Notes*, Vol. 19, No. 5. ACM, 1994, pp. 154–163.

[10] M. J. Harrold and M. L. Soffa, "Interprocedual data flow testing," in *ACM SIGSOFT Software Engineering Notes*, Vol. 14, No. 8. ACM, 1989, pp. 158–167.

[11] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro, "Coverage testing of java programs and components," *Science of Computer Programming*, Vol. 56, No. 1, 2005, pp. 211–230.

[12] J. Laski and W. Stanley, *Software verification and analysis: An integrated, hands-on approach.* Springer Science & Business Media, 2009.

[13] M. L. Chaim and R. P. A. De Araujo, "An efficient bitwise algorithm for intra-procedural data-flow testing coverage," *Information Processing Letters*, Vol. 113, No. 8, 2013, pp. 293–300.

[14] R. P. A. d. Araujo and M. L. Chaim, "Data-flow testing in the large," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on.* IEEE, 2014, pp. 81–90.

[15] M. Vivanti, "Dynamic data-flow testing," in *Companion Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 682–685.

[16] A. Rembiszewski, "Data flow coverage of object programs," Master's thesis, Institute of Computer Science, Warsaw University of Technology, 2009, (in Polish).

[17] I. Bluemke and A. Rembiszewski, "Dataflow testing of java programs with dfc," in *Advances in Software Engineering Techniques*, ser. Lecture Notes in Computer Science, T. Szmuc, M. Szpyrka, and J. Zendulka, Eds. Springer Berlin Heidelberg, 2012, Vol. 7054, pp. 215–228. [Online]. http://dx.doi.org/10.1007/978-3-642-28038-2_17

[18] B. Beizer, *Software system testing and quality assurance.* Van Nostrand Reinhold Co., 1984.

[19] Junit homepage. (Accessed 12.2008). [Online]. http://www.junit.org/

[20] Eclemma 1.2.0. (Accessed 04.2008). [Online]. http://www.eclemma.org/

[21] Tptp: Eclipse test & performance tools platform project. (Accessed 2008). [Online]. http://www.eclipse.org/tptp/

[22] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria," in *Proceedings of the 16th international conference on Software engineering.* IEEE Computer Society Press, 1994, pp. 191–200.

[23] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on.* IEEE, 2009, pp. 56–66.

[24] I. Bluemke and K. Kulesza, "A comparison of dataflow and mutation testing of java methods," in *Dependable Computer Systems.* Springer, 2011, pp. 17–30.

[25] Muclipse homepage. (Accessed 01.2011). [Online]. http://muclipse.sourceforge.net/index.php

[26] Jumble homepage. (Accessed 12.2008). [Online]. http://jumble.sourceforge.net/index.ht