Dataflow testing of Java programs with DFC

Ilona Bluemke and Artur Rembiszewski

Institute of Computer Science, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
I.Bluemke@ii.pw.edu.pl

Abstract. The objective of this paper is to present a tool supporting dataflow coverage testing of Java programs. 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. DFC finds all definition-uses (def-*u*) pairs in tested unit and provides also the def-*u* graph DUG for methods. After the execution of tests the information which def-*u* pairs were covered is shown. An example of usage of DFC and the comparison with JaBUTi data flow testing tool of Java program is also presented.
Keywords: Dataflow coverage testing, tool supporting testing.

1 Introduction

One of the key issues in developing high quality software systems is effective testing. Popular approaches to testing include ”black box” and ”white box” [1,2]. In white box approach the test cases can be derived from the code of the unit under test. Code based approach can be divided into two main types: data flow coverage methods [3,4,5,6,8] and control flow coverage e.g. [7]. In dataflow testing relationships between data are used to select the test cases. This approach was introduced for structural programming languages [3] and later adopted for object languages [4,5,6]. Although experiments show [9] 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 [10] 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. DFC, presented in Section 3, finds all definition-uses (def-*u*) pairs in tested unit (Section 2) and provides also the def-*u* graph (DUG) for methods. The basics of dataflow testing are described in Section 2. After the execution of test, the tester is provided with the information which def-*u* pairs were covered so (he or she) 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 is used to explain the data flow coverage testing and to show some advantages of DFC. Section 5 contains some conclusions.

2 Dataflow Testing

Dataflow testing is one of "white box" techniques, it means that the test cases are derived from the source code. In dataflow testing [3,4] 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 a 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 by *d* over some program path. Test criteria are used to select particular definition-use pairs. A test satisfies def-*u* pair, if executing the program with this test causes traversal of a subpath 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 that will cause it to be executed. Data-flow testing criteria [1] use the def-*u 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. Many def-*u* criteria have been proposed and compared [3,4,6]. One 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 $\langle i, j \rangle$ is covered. In node *j* this variable is used $v \in use(j)$ and on this path variable *v* is not redefined.

The first dataflow technique [3] was proposed to structural programming languages and was not able to detect dataflow interactions that arise when methods are invoked in an arbitrary order. In [5] an algorithm called PLR, was proposed. PLR finds 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. This method can be adapted to global variables, class attributes and referenced method arguments in testing object programs.

For object programs three levels of dataflow testing were proposed in [4]:

- *Intra-method* – testing, based on the basic algorithm [3], is performed on each method individually; class attributes and methods interactions can not be taken into account.
- *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.
- *Intra-class* – interactions of public methods are tested, when they are called in various sequences. Since the set of possible public methods calls sequences is infinite, only a subset of it is tested.

For each of the above described testing levels appropriate def-*u* pairs were defined i.e. *intra-method, inter-method* and *intra-class*.

2.1 Related Work

The process of testing software is extremely expensive in terms of labor, time and cost so many tools supporting this process have been developed. Many of these tools are standalone applications e.g. JaBUTi [10], Emma [11], PurifyPlus [12], some are Eclipse plug-ins e.g.: EclEmma [13], TPTP [14]. Tools supporting "white box" testing are often dedicated to a programming language e.g. to C - [15] or to Java - [11]. Tools providing information about code coverage are often integrated with CASE tools e.g. RSA (Rational Software Architect version 7.5) [16] or with programming environments e.g. Visual Studio C++ [17].

The majority of tools supporting "white box" testing are code (instruction) coverage analyzers (e.g. PurifyPlus, TPTP, RSA, EclEmma, Emma). They provide information about line, methods, class, package, file or even project coverage. The information is hierarchically ordered. Usually not covered code is displayed in red. These tools are able to store information concerning separate test cases and later produce a summary report for whole test suite.

We were able to find only one tool, named JaBUTi (Java Bytecode Understanding and Testing) [10], supporting dataflow testing of Java programs. This tool is able to analyze code coverage and dataflow coverage as well. JaBUTi analyzes following criteria based on DUG def*-u graph*:

1. Control flow coverage:
   - *All-Nodes-ei* – every node of the DUG graph, reachable through an exception-free path, is executed at least once.
   - *All-Nodes-ed* – every node of the DUG graph, which can be reached only if Java exception was thrown, is covered.
   - *All-Edges-ei* – all DUG edges, except edges for which Java exception are called, are covered.
   - *All-Edges-ed* – all DUG edges which can be reached only if Java exception was thrown were covered.
2. Dataflow coverage:
   - *All-Uses-ei* – all-uses criterion is fulfilled, paths throwing Java exception are not covered.
   - *All-Uses-ed* – all-uses criterion is dedicated to paths throwing Java exception.

The abbreviations *ei* and *ed* mean accordingly *exception independent* and *exception dependent*. Fulfilling two criterions: *All-Nodes-ei* and *All-Nodes-ed* is equivalent to instruction coverage, fulfilling criterions *All-Edges-ei* and *All-Edges-ed* is equivalent to conditions coverage and both *All-Uses-ei* with *All-Uses-ed* guarantees *all-uses* dataflow coverage [6].

Dataflow testing of object programs can reveal many errors. An experiment described in [9], shows, that in testing C++ programs using dataflow methods and information about polymorphism and inheritance, 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 dataflow testing of Java programs presented in next section.

3 DFC – a Tool for Dataflow Testing

Dataflow testing can not be applied in isolation so we decided to implement a tool supporting this approach, DFC - Data Flow Coverage (Fig. 1), as an Eclipse plug-in. In Eclipse Java programming environment and testing tools e.g. JUnit [18] are available. DFC finds all def-*u* pairs in testing Java code and after 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. In preparing test cases tester can also use def-*u graph* (DUG) for a method provided by DFC.

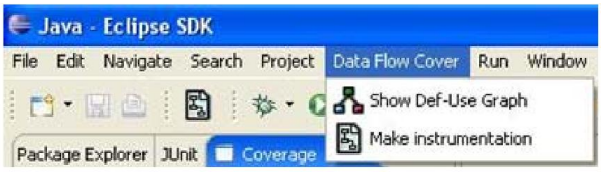

Fig. 1. DFC menu.

In object languages the dataflow testing ideas proposed [3] 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 tested method. Definitions of variable `x` are e.g.:

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

An instruction is using a variable `x` e.g.:

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

In DFC tester may decides which method is defining and which one is using object state. Exemplary screen used while setting methods properties is given in Fig. 2. Initially all methods are assumed as modifying and using object state.

In Fig. 3 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

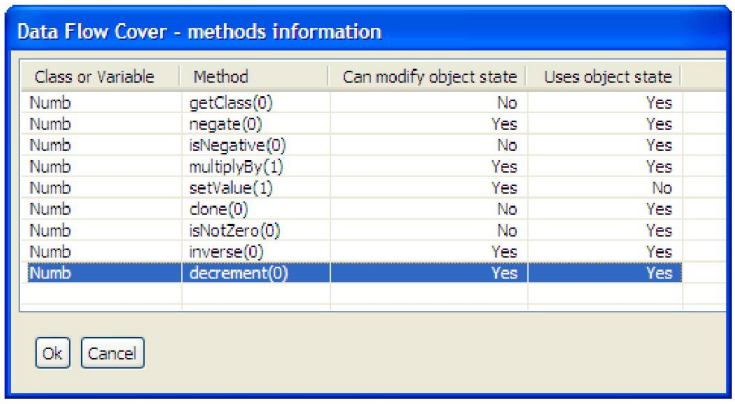

Fig. 2. DFC configuration screen for code from Table 4 - marked methods as modifying/using object state.

Java source code (SRC in Fig. 3). Module `Knowledge base` analyses the source code and generates the list of classes and methods. On this list tester can mark methods as modifying or using object state (Fig. 2). The module `Instrumentation` instruments source code (adds extra instructions needed for finding dataflow coverage) and builds def-u graph (DUG). To instrument the code user should press the instrumentation button shown in Fig. 4. Example of instrumented code is given in Table 1.

Table 1. Example of instrumented code.

```
if (addVat)
  bill.add(vatAmount);
```

```
44) dfc_runtime_report.add(4);
45) if (addVat) {
46)   bill.add(vatAmount);
47)   dfc_runtime_report.add(5);
48) }
```

DUG contains information concerning 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 pairs. The detailed description how the pairs are being determined is given in [19]. The algorithm is not able to deal with variable aliasing. The instrumented code should be compiled and run in Eclipse environment. The extra code added by `Instrumentation` module sends data concerning the pair coverage to DFC.

Module `Analyzing` is locating covered and not covered def-u pairs in tests. Results of this module are presented in Fig. 5. Other information on DFC implementation can be found in [19].

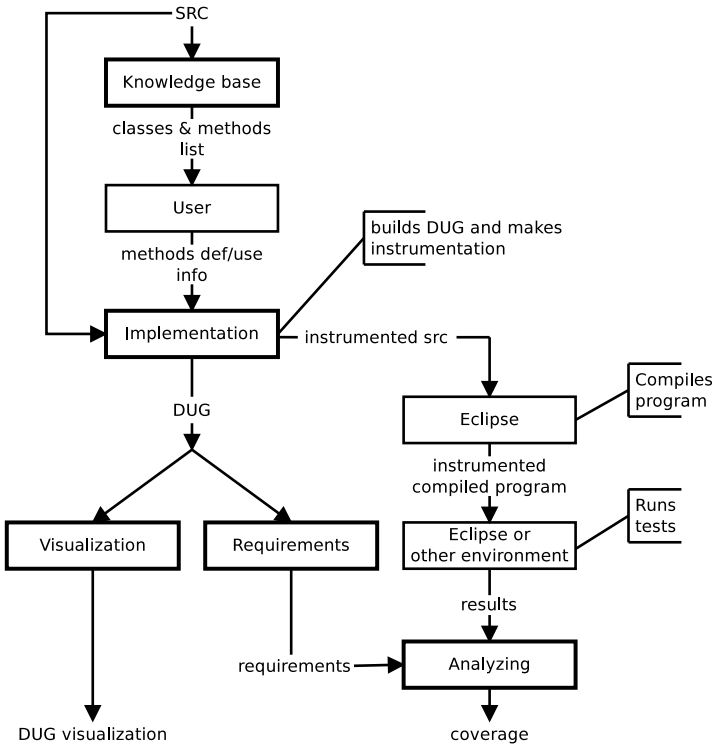

Fig. 3. Process of testing with DFC.

Fig. 4. Make instrumentation button in Eclipse with DFC.

4 Example

In this section some functions of DFC are shown on a small example. In Table 2 Java source code calculating x^y is given. This implemented algorithm was used by Rapps and Weyuker in [3] as an example to present the data flow coverage method.

According to introduced in [1] definitions and usage of variables, for variable `pow` (code in Table 2) definitions can be found in lines: 15, 18, 24 and for variable `z` in: 14, 20, 23, 28. DUG graph is presented in Fig. 6.

In Table 3 pairs def-u are given for variables `pow` and `z`. These pairs are represented as a pair of numbers (n_{def}, n_{use}), where n_{def} denotes the number of a line containing variable definition and n_{use} represents the number of line containing the usage of variable. These pairs can be used to find appropriate data flow coverage criterion.

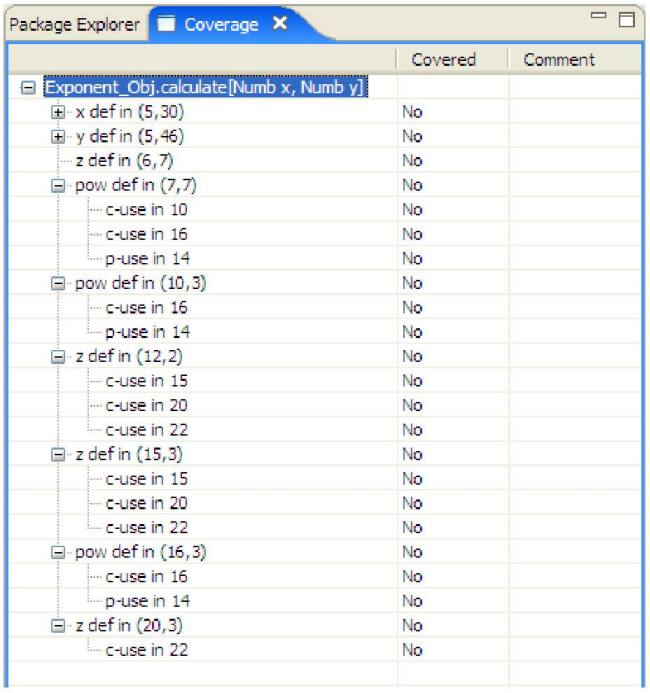

Fig. 5. DFC screen presenting coverage.

In Table 4 the algorithm from Table 2 is rewritten using object variables. Method `calculate` uses arguments compatible with interface `Numb`. This interface may be implemented in classes for different types of numbers so the algorithm in class exponent is general. Writing this method we were trying to keep the line numbering as in Table 2 to make the comparison easier. For code in Table 4 the definitions are following: for variable `pow` – line 15 and for variable `z` – line 14. These variable definitions were calculated according to dataflow coverage criterion proposed in [4]. Such criterion is used in JaBUTi tool. Our DFC tool may be used to find DUG graph. Implicit DFC setting recognizes all methods of `Numb` interface as not modifying object state and using it. The DUG is presented in Fig. 7 and definition-use pairs for this piece of code are given in Table 5.

Analyzing def-u pairs from Table 5 we can notice that not all such pairs were identified. The state of object (code in Table 4) is modified by calls of methods not only by assignment to an object variable. Instructions in lines 18 and 28 are not treated as definitions of variable pow but as a method call. Assignment to analyzed variables (`pow` and `z`) is made only once, so some def-u pairs, with definitions in following lines were not detected. The differences can be observed on DUG graphs in Fig. 7 and 8.

JaBUTi identifies set of pairs def-u as shown in Table 5 and above we proved that sometimes this tool is not able to correctly identify all def-u pairs. As variable defini-

Table 2. Java code for x^y calculation.

```
11) public class exponent_RW {
12)
13) public static double calculate(double x/*base*/,
                                   int y/*exponent*/) {
14)   double z; //result
15)   int pow = y;
16)
17)   if (y < 0)
18)     pow = - pow;
19)
20)   z = 1;
21)
22)   while (pow!=0) {
23)     z = z * x;
24)     pow = pow - 1;
25)   }
26)
27)   if (y < 0)
28)     z = 1 / z;
29)
30)   return z;
31) }
32) }
```

Table 3. def-*u* pairs for code from Table 2.

variable	def-usage pairs
`pow`	(15,18), (15,22), (15,24), (18,22), (18, 24), (24,22), (24,24)
`z`	(20,23), (20,28), (20,30), (23,23), (23,28), (23,30), (28,30)

tions should be treated also the calls of following methods: `negate`, `multiplyBy`, `invers`, `decrement`, `setValue` (according to the concept of `Numb` interface).

All def-*u* pairs for code given in Table 4 will be correctly detected by our DFC tool after setting appropriate options. The above indicated methods should be marked as modifying object state. These methods, except `setValue`, should be also marked as using the state of object. After setting appropriate methods attributes, described above, DFC will also find definitions for variable `pow` in lines: 15, 18, 24, and for variable `z` in lines: 14, 20, 23, 28. Afterwards DFC finds def-*u* pairs shown in Table 6. It is worth noticing that the definition of variable z in line 14 can not be reached by any usage of this variable. The DUG graph obtained after setting appropriate methods as defining/using object state is presented in Fig. 5 and DFC configuration window in Fig. 2.

Our approach based on setting of methods attributes as modifying/using object state enabled the correct identification of all def-*u* pairs in the code (Table 4), the same as in structural code given in Table 2.

Table 4. Object code for code in Table 2.

```
11) public class Exponent_Obj {
12)
13) public static Numb calculate(Numb x/*base*/,
                                 Numb y/*exponent*/) {
14)   Numb z = NumbFactory.createTypeOfResult(
        x.getClass(), y.getClass()); //result
15)   Numb pow = y.clone();
16)
17)   if (y.isNegative())
18)     pow.negate();
19)
20)   z.setValue(1);
21)
22)   while (pow.isNotZero()) {
23)     z.multiplyBy(x);
24)     pow.decrement();
25)   }
26)
27)   if (y. isNegative())
28)     z.inverse();
29)
30)   return z;
31) }
32) }
```

Table 5. def-u pairs for code from Table 4, DUG graph in Fig. 4.

variable	def-usage pairs
pow	(15,18), (15,22), (15,24)
z	(14,20), (14,23), (14,28), (14,30)

Table 6. def-u pairs – code from Table 4, marked methods modifying object state.

variable	def-usage pairs
pow	(15,18), (15,22), (15,24), (18,22) , (18,24), (24,22) ,(24,24)
z	(20,23), (20,28), (20,30), (23,23), (23,28), (23,30), (28,30)

5 Conclusions

Many authors e.g. [1] suggest that effective testing can be achieved if different testing approaches e.g. functional and structural are used. In the development of high quality software systems thorough testing can be the crucial issue. In this paper we present 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

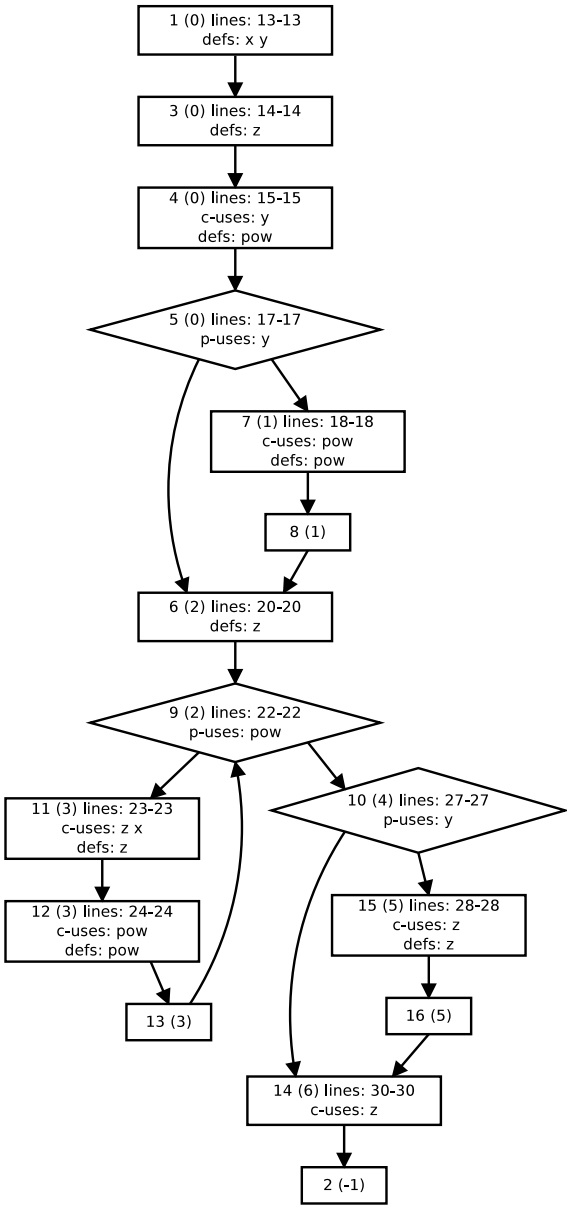

Fig. 6. DUG graph for source code from Table 2.

that may not be uncovered by black box testing. The detected errors depend on the test cases designed by a programmer, DFC checks, if specific paths derived from test cases are covered. In Eclipse environment there are other tools available for testing Java programs, some of them are listed in Subsect. 2.1. These plug-ins use different testing

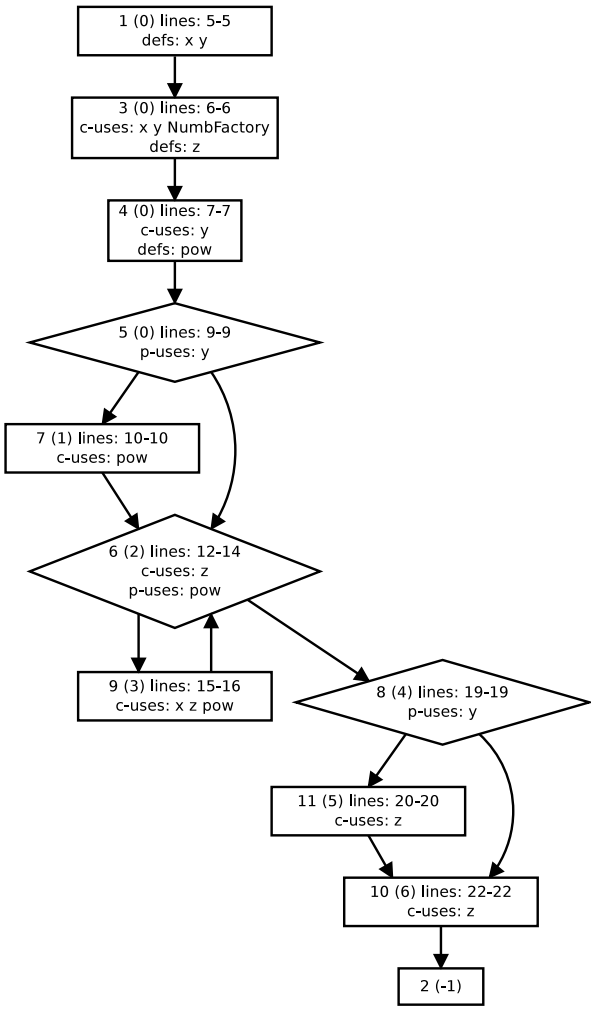

Fig. 7. DUG graph for source code from Table 4.

techniques and, to our best knowledge, none of them provides dataflow coverage testing. In DFC tester can design tests to achieve e.g. def-*u* or *all-uses* coverage criteria which also guarantee instruction coverage [3].

In DFC tester can identify defining and using methods (Fig. 3). However this process is time consuming, we are not going to make it fully automatic. To identify if a method is defining or using object state, the analysis of the source code must be performed. For simple classes this analysis may be automatic but 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, might be a solution. Such approach needs additional code instrumentation and re-execution of test cases and we

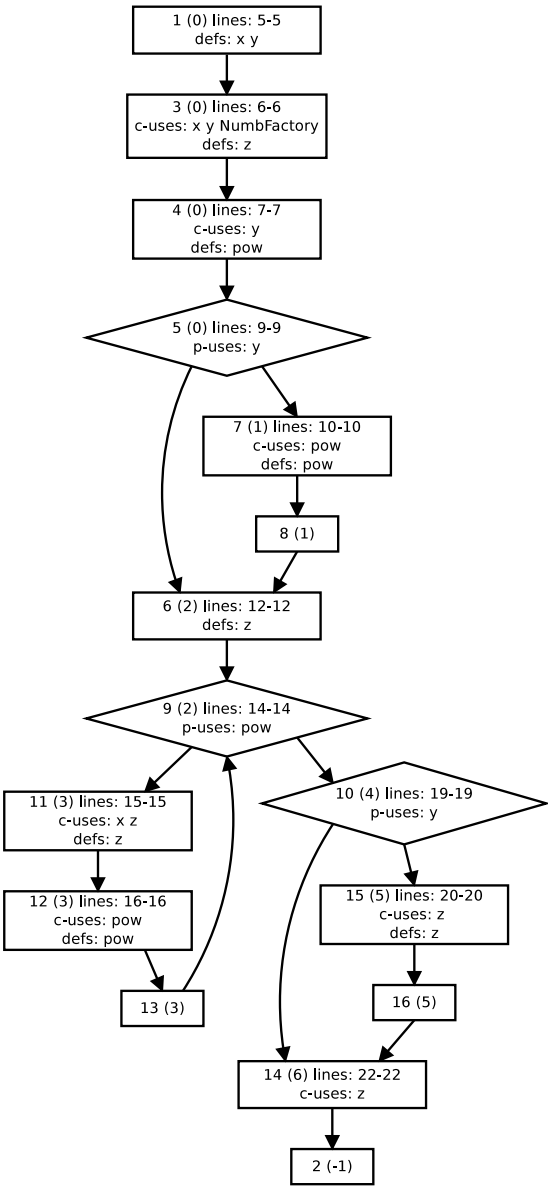

Fig. 8. DUG graph for source code from Table 4 with marked methods as modifying object state.

think it is not worthy to implement it. To simplify the implementation of DFC we also assumed to ignore variable aliasing.

In JaBUTi [10] (Subsect. 2.1) every call of a method is treated as using object state. In DFC tester can determine methods as modifying or/and using object state. This approach is novel and is not implemented in other data flow testing tools. 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.

Finally, we outline the direction for our 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. In addition, we want to extend the *intra-method* testing criteria to wider, *inter-method* level, so more errors could be detected.

Acknowledgments

The authors would like to acknowledge interesting remarks given by three anonymous reviewers.

References

1. Bezier, B.: Software System Testing and Quality Assurance. Van Nostrad Rheinhold, New York (1984)
2. Binder, R.V.: Testing Object Oriented Systems, Addison Wesley (1999)
3. Rapps, S., Weyuker, E.J.: Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering 11, 367–375 (1985)
4. Harrold, M.J., Rothermel G.: Performing Data Flow Testing on Classes. In: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 154–163 (1994)
5. Harold, M.J., Soffa, M.L.: Interprocedural Data Flow Testing. In: Proceedings of the 3rd Testing, Analysis, and Verification Symposium, 158– 167 (1989)
6. Vincenzi, A.M.R., Maldonado J.C., Wong, W.E., Delamaro, M.E.: Coverage Testing of Java Programs and Components. Science of Computer Programming 56, 211–230 (2005)
7. Woodward, M.R., Hennell, M.A.: On the Relationship Between Two Control-flow Coverage Criteria: All JJ-paths and MCDC. Information & Software Technology 48, 433–440 (2006)
8. Malevris, N., Yates, D.F.: The Collateral Coverage of Data Flow Criteria When Branch Testing. Information and Software Technology 48, 676–686 (2006)
9. Chen, M.H., Kao, H.M.: Testing Object-Oriented Programs An Integrated Approach. In: Proceedings of the 10th International Symposium on Software Reliability Engineering, pp. 73–83 (1999)
10. JaBUTi Homepage, http://jabuti.incubadora.fapesp.br (access 12.2007)
11. Emma, http://emma.sourceforge.net (access 2008)
12. PurifyPlus, IBM Rational, http://www-128.ibm.com/developerworks/rational/products/purifyplus (access: 03.2008)
13. EclEmma 1.2.0, http://www.eclemma.org/ (access 2008)
14. TPTP: Eclipse Test & Performance Tools Platform Project, http://www.eclipse.org/tptp/ (access 3.2009)
15. Horgan, J.R, London, S.: A Data Flow Coverage Testing Tool for C. In: International Symposium on Software Testing and Analysis'91, pp. 87–97 (1991)
16. Rational Software Architect, IBM: http://www.ibm.com/developerworks/downloads/r/rsd/learn.htm
17. Visual: http://www.microsoft.com/visualstudio/en-us/products/default.mspx
18. JUnit, http://www.junit.org/ (access 12.2008)
19. Rembiszewski, A.: Data Flow Coverage of Object Programs. Msc Thesis, Institute of Computer Science, Warsaw University of Technology, (in polish) (2009)