White Box Testing Tool Prototype Development

Arlinta Christy Barus, Dian Ira Putri Hutasoit, Joel Hunter Siringoringo, Yusfi Apriyanti Siahaan
Informatics Engineering Study Program
Del Institute of Technology
Jl. Sisingamangaraja, Kec. Laguboti, Kab. Tobasa, Sumatera Utara, Indonesia
arlinta@del.ac.id, dian.hutasoit@del.ac.id, hunterharada@gmail.com, yusfisiahaan@gmail.com

***Abstract*—nowadays, software testing is viewed as an important phase in software engineering life cycle as it aims to improve the quality of software under development. Due to limited software testing tools available for free, many developers cannot do a comprehensive testing to the software under development before launching the software. For this reason, this research aims to provide a prototype of a software testing tool that can be used as an initial base model for further development. The tool implements white box testing method that means the testing explores source codes of software under test to find the errors inside. We focus on one of data flow coverage techniques known as *all p-uses* coverage. The tool is able to calculate the *all p-uses* coverage percentage of the software under testing.**

Keywords—software testing; white box testing; data flow coverage; all p-uses

I. INTRODUCTION

Software testing is a process of executing a program with the intent of finding errors [1]. The activity can spend about 50%-80% of the total cost of software development [2]. It is indeed an important activity that should be conducted by developers before the software is ready to launch or deploy. It helps to ensure that software under development meets the customers' requirements. In other words, software testing facilitates the improvement of the quality of the software under development.

Prior to this research, a survey was conducted to a batch of Del Institute's alumnae that have been working as IT practitioners, aiming to find out how well software testing techniques had been used in the routine software development industry, particularly in Indonesia. Approximately 90% of the respondents answered that the techniques were rarely used due to their limited knowledge of software testing and limited availability of free software testing tools. Mostly, the free tools are just automating the execution of the software under test (such as JUnit [13] and Selenium [14]) whilst the test cases and the expected output have to be provided manually by testers and then pass them as the inputs to the tools.

Therefore, this research focusses on building a prototype of a software testing tool that implements one of software testing techniques. Its main function is to calculate the *all p-uses* coverage on software tested. In the future, it is then expected to be expanded as a complete tool with having more features and better interface and performance.

The rest of the paper is organized as follow: Section II addresses the literature study on various related software testing and Section III presents the analysis, design and implementation of the prototype. The testing of the prototype itself is discussed in Section IV. The related work is discussed in Section V and then is concluded in the last Section, Section VI.

II. LITERATURE STUDY

A. White Box Testing

White box testing, also known as code-based testing or glass box testing or structural testing, is one of the most important software testing techniques. It generates test cases based on the source code and internal workings of the source. It is very effective in validating design, decision, assumptions and finding errors program in software [4, 11, 12].

In white box testing, testing aims to check which parts of the source program that have been and have not been touched by a series of executed test cases. By knowing this information, testers may be able to control the selection of test case execution to make sure all parts are covered. There are two types of coverage criterion that can be approached i.e. Control Flow Coverage and Data Flow Coverage [9, 10].

B. Control Flow Coverage

Control flow criterion measures the flow of control between statements and sequences of statements [5, 9, 10]. There are several different techniques using this criterion such as Statement Coverage (SC), Condition Coverage (CC), Decision Coverage (DC), Condition/Decision Coverage (C/DC), Modified Condition/Decision Coverage (MC/DC), and Multiple Condition Coverage (M-CC). The brief description of each can be seen in Table I.

TABLE I. CONTROL FLOW CRITERIA

Coverage Criteria	**SC**	**CC**	**DC**	**C/DC**	**MC/DC**	**M-CC**
Every statement in the program has been invoked at least once	√					
Every condition in a decision in the program has taken all possible outcomes at least once		√		√	√	√
Every decision in the program has taken all possible outcomes at least once			√	√	√	√
Every condition in a decision has been shown to independently affect that decision's outcome					√	√
Every combination of condition outcomes within a decision has been invoked at least once						√

978-1-4673-7319-7/15/$31.00 ©2015 IEEE

C. Data Flow Coverage

Data flow criteria measures the flow of data between variable assignments and references to the variables [5,8]. In data flow coverage testing, three types of data usage are defined as followings:

1. Define (def.)
   Giving a value for variable, and includes passing values into parameters, e.g. x=1
2. Computational-use (c-use)
   Using variable's value in a predicate/decision, e.g. if (x==1)
3. Predicate-use (p-use)
   Using variable's value for computation, e.g. y=x+1

Testers need to determine the data usage type of each variable in the source code of software under test and then followed by determining path from each variable.

Data flow testing also has six criteria explained as follows [6]:

1. **All Definitions**
   For each variable x and each node i, in which x has a global definition in node i, selects a complete path which includes a definition clear path from node i to
   - Node j having a global c-use of x, or
   - Edge (j, k) having a p-use of x
2. **All c-uses**
   For each variable x and each node i, in which x has a global definition in node i, selects a complete paths which includes a definition clear path from node i to all nodes j such that there is a global c-use of x in j.
3. **All p-uses**
   For each variable x and each node i, in which x has a global definition in node i, selects a complete paths which includes a definition clear path from node i to all edges (j, k) in which there is a p-use of x (j, k).
4. **All p-uses/some c-uses**
   This criterion is identical to the all-p-uses criterion except when a variable x has no p-use. If x has no p-use, then this criterion reduces to the some-c-uses criterion.
   Some-c-uses: For each variable x and each node i, in which x has a global definition in node i, selects complete paths which include def-clear paths from node i to some nodes j such that there is a global c-use of x in j.
5. **All c-uses/some p-uses**
   This criterion is identical to the all-c-uses criterion except when a variable x has no c-use. If x has no global c-use, then this criterion reduces to the some-p-uses criterion.
   Some-p-uses: For each variable x and each node i, in which x has a global definition in node i, selects complete paths which include def-clear paths from node i to some edges (j, k) in which there is a p-use of x on (j, k).
6. **All uses**
   This criterion produces a set of paths due to the all-p-uses criterion and the all-c-uses criterion.

   In this tool, the criterion which is used in the tool is *all p-uses* criterion.

D. All P-Uses Coverage Percentage Calculation

The formula used to calculate *all p-uses* coverage percentage is shown in formula below.

$$\%\ Coverage = \frac{\sum \text{All p-uses paths passed by test case}}{\sum \text{All p-uses path}} \times 100\%$$

Coverage percentage of a test case can be known by dividing ∑ all p-uses paths which are passed by test case with ∑ all p-uses path in program.

The example of source code of program P that will be used to illustrate the calculation of *all p-uses* coverage percentage is displayed in Fig. 1 below.

```
#include <stdio.h>
#include <stdlib.h>
int main(int a, int b)
{
    printf ("Input the value of a : ");//Node-1(N1)
    scanf ("%d", &a);
    printf ("Input the value of b : ");
    scanf ("%d", &b);
    if (a>0 && b>0)//Node-2(N2)
        {
            a=b+2;//Node-3(N3)
            printf ("The value of a = %d ", a);
            printf ("\n");
    }else if (a>0 || b>0)//Node-4(N4)
    {     a=b-2;//Node-5(N5)
          printf ("The value of a = %d ", a);
          printf ("\n");}
    else {
         a=0;//Node-6(N6)
         printf ("The value of a = %d ", a);
         printf ("\n");
         b=0;
         printf ("The value of b = %d ", b);
        }
        printf ("Finish");//Node-7(N7)
        system ("pause");}
```

Fig 1. The example of source codes

The control flow graph from source code program P is drawn in Fig. 2 below.

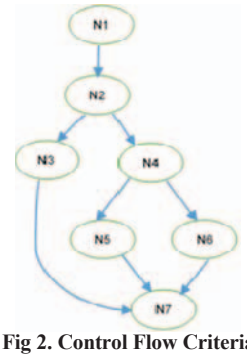

Fig 2. Control Flow Criteria

The first step to calculate all p-uses coverage percentage is determining possible paths of in program P. Then, next is to determine *all p-uses* variables.

Thus, based on the program P, all possible paths are as follows:

- N1-N2-N3-N7
- N1-N2-N4-N5-N7

- N1-N2-N4-N6-N7

Definition variable in the program are as follows:

- Variable a in N1
- Variable b in N1
- Variable a in N3
- Variable a in N5
- Variable a in N6
- Variable b in N6

P-use variable in the program are as follows:

- Variable a in *edge* N2-N3
- Variable b in *edge* N2-N3
- Variable a in *edge* N2-N4
- Variable b in *edge* N2-N4
- Variable a in *edge* N4-N5
- Variable b in *edge* N4-N5
- Variable a in *edge* N4-N6
- Variable b in *edge* N4-N6

From the definition and p-use variable above, we can determine all p-use variables as follows:

- Variable a in Node 1 (as the last definition in *Node* 1, and as *p-use* in *edge* N2-N3)
- Variable a in Node 1 (as the last definition in *Node* 1, and as *p-use* in *edge* N2-N4)
- Variable a in Node 1 (as the last definition in *Node* 1, and as *p-use* in *edge* N4-N5)
- Variable a in Node 1 (as the last definition in *Node* 1, and as *p-use* in *edge* N4-N6)
- Variable b in Node 1 (as the last definition in *Node* 1, and as *p-use* in *edge* N2-N3)
- Variable b in Node 1 (as the last definition in *Node* 1, and as *p-use* in edge N2-N4)
- Variable a in Node 1 (as the last definition in *Node* 1, and as *p-use* in *edge* N4-N5)
- Variable a in Node 1 (as the last definition in *Node* 1, and as *p-use* in *edge* N4-N6)

And full paths of the all p-use variables in the program are as follows:

- Node 1 : a → N1-N2-N3-N7 , N1-N2-N4-N5-N7 , N1-N2-N4-N6-N7
- Node 1 : b → N1-N2-N3-N7 , N1-N2-N4-N5-N7 , N1-N2-N4-N6-N7
- Node 2 : -
- Node 3 : -
- Node 4 : -
- Node 5 : -
- Node 6 : -
- Node 7 : -

From the two *all p-uses* variables above, we suppose that a test case has variable *a* and variable *b* with values 0 and 1 respectively. The execution of the test case will pass the first path which is N1-N2-N4-N5-N7, out of the three *all p-uses* paths. So, the *all p-uses* coverage for the first test case is = 1/3 * 100% = 33.3 %.

Next, suppose the second test case has values 0 for both variable *a* and variable *b*. The execution of the test case will pass the second path which is N1-N2-N4-N6-N7. The cumulative *all p-uses* coverage (of first and the second test cases) is = 2/3 * 100% = 66.67 %.

Suppose the last test case has variable *a* and variable *b* with values 5 and 6 respectively. The test case will pass the last path that has not been executed by previous test cases which is N1-N2-N3-N7. The calculation of cumulative *all p-uses* coverage percentage became 3/3 * 100% = 100 %. Thus, we can say that the three test cases have achieved 100% of *all p-uses* coverage. We also can conclude that the program needs at least three test cases to reach 100% of *all p-use* coverage.

The resume of the calculation can be seen as Table II follows:

TABLE II. ALL P-USES COVERAGE PERCENTAGE CALCULATION

No	Test Case	Test Case Path	Coverage
1	a=0, b=1	N1-N2-N4-N5-N7	1/3 * 100% = 33.33%
2	a=0, b=0	N1-N2-N4-N6-N7 + accumulation from previous test case.	2/3 * 100% = 66.67%
3	a=5, b=6	N1-N2-N3-N7 + accumulation from previous test case.	3/3 * 100% = 100%

III. ANALYSIS, DESIGN, AND IMPLEMENTATION

Based on the calculation of all p-uses coverage, the features of the tool can be described as follows:

1. *Calculating coverage feature*
   This feature calculates the all p-uses coverage percentage of the program tested.
2. *Resetting feature*
   This feature resets the calculation of all p-use coverage percentage thus the calculation is not accumulated with the previous calculation.
3. *Simplifying looping path feature*
   This feature simplifies the test case path having some looping node caused by 'while' program.
4. *Finding node of source code of program*
   This feature finds the node of source code of program based on test case inputted by tester.

The business processes of the tool to calculate the *all p-uses* coverage sequentially are as follows (as depicted by Fig. 3):

1. Tool reads the program properties inputted by tester. The properties are nodes, paths, definition variables and the node, p-uses variables and the edges.
2. Tool searches global definition variables in every node.
3. Tool retrieves p-use from variable that has global definition.
4. Tool determines definition clear path of all global definition variables to all of p-use variables.
5. Tool determines path of all p-uses variables.
6. Tool reads test case paths inputted by tester.
7. Tool determines whether the path passed by test case is all p-uses path or not.
8. Tool calculates the percentage of all p-uses coverage.
9. Tool shows the percentage of all p-uses coverage.

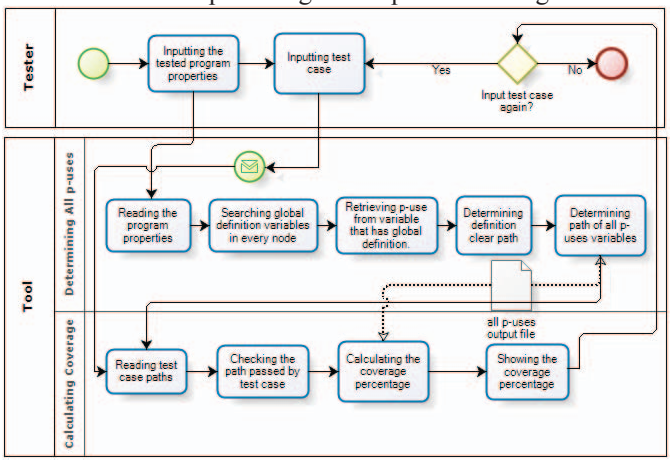

Fig 3. The flow of actions inside the tool

After analyzing and designing the tool, the main form tool that is shown to tester is in Fig. 4 below:

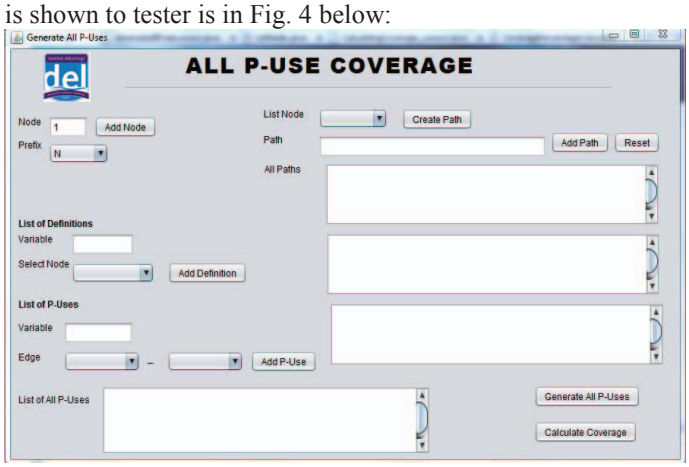

Fig 4. Developed Tool Main Form

After generating all p-use, next the tester can calculate the all p-uses coverage as is shown in the following Fig. 5:

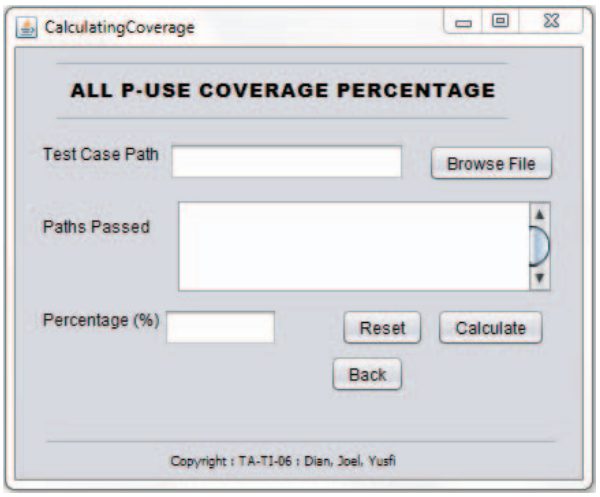

Fig 5. All p-use coverage Percentage Form

The tool was developed using Java Programming language. All of properties inputted by tester are processed in List, and for all p-uses coverage calculation, the tester must modify the program to produce paths which are passed by test case inputted to the program. All the test case paths are saved in a text file as the following:

Node's value +'-'+ Node's value + '-'+ Node's value +…

which node's value = 1-2-3-4

IV. TESTING

In order to test the tool, we provided three small programs as the programs under test, as followings:

1. If Program
   A Java program that has only one condition.
2. Nested If Program
   A C-program that has a nested condition enclosed.
3. Combination of If and While Program
   A Java program having one condition and one while (looping).

The tool was tested using hardware with the specification which is shown in Table III below:

TABLE III. HARDWARE SPECIFICATION

Specification	Type
Processor	Intel® Pentium® CPU P6300 @ 2.27 GHz
Memory	3.00 GB
System type	32-bit Operating System
Operating System	Windows 7

The steps to calculate the *all p-uses* coverage by using the developed tool is shown in the following Fig. 6:

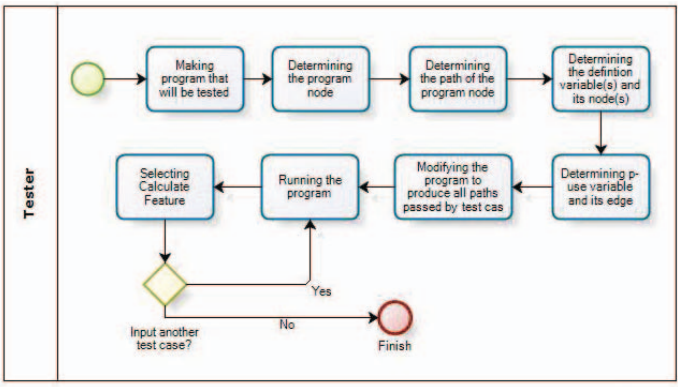

Fig 6. Calculation Steps

One of the testing scenarios to calculate the all p-uses coverage can be seen in Table IV below:

TABLE IV. CALCULATE COVERAGE TESTING SCENARIO

Test Case	Calculate Coverage testing		
Purpose	to verify whether the tool can calculate the all p-uses coverage percentage of the program tested.		
Description	if the testing is successful, the *all p-uses* coverage percentage is calculated. The calculation is accumulated with the previous coverage percentage calculation.		
Pre-condition	- All p-uses paths have been generated by tool. - Input file containing all paths passed by test case has been determined by tester.		
Test Scenario			
1. Tester clicked the Calculate Coverage button at the main form. 2. Tool showed Calculating Coverage form. 3. Tester clicked Browse File button. 4. Tester chose the file input location. 5. Tester clicked Calculate button. 6. Tool showed the all p-uses coverage percentage calculation. 7. If tester wants to retest the calculation by adding other paths, back to test scenario step 3. 8. If tester wants to back to main form, the tester must click Back button. 9. If tester wants to reset the calculation, the tester must click Reset button and then back to step 3.			
The Result Evaluation Criteria			
Tool successfully calculated the all p-uses coverage percentage.			
Case and Test Result (Normal Data)			
Input Data	**Expected Result**	**Observation**	**Conclusion**
Text file contained: 1-2-4-6-7 1-2-3-7	At the paths Passed text area, the tool showed: N1-N2-N4-N6-N7 N1-N2-N3-N7 and the calculation of all p-uses coverage was 66,67%.	At the paths Passed text area, the tool showed: N1-N2-N4-N6-N7 N1-N2-N3-N7 and the calculation of all p-uses coverage was 66,67%.	**Accepted**
The same	The all p-uses	The all p-uses	**Accepted**
text file inputted before. Text file contained: 1-2-4-6-7 1-2-3-7	coverage was still 66,67% because the paths inputted were same.	coverage was still 66,67% because the paths inputted were same.	
Text file contained: 1-2-4-5-7	At the paths Passed text area, the tool showed: N1-N2-N4-N5-N7 and the calculation of all p-uses coverage became 100%.	At the paths Passed text area, the tool showed: N1-N2-N4-N5-N7 and the calculation of all p-uses coverage became 100%.	**Accepted**
Case and Test Result (Normal Data)			
Input Data	**Expected Result**	**Observation**	**Conclusion**
No input.	Tool neither calculated nor showed the percentage calculation.	Tool neither calculated nor showed the percentage calculation.	**Accepted**
.doc file contained : 1-2-4-6-7 1-2-3-7	Tool neither calculated nor showed the percentage calculation.	Tool neither calculated nor showed the percentage calculation.	**Accepted**
Empty text file	Tool neither calculated nor showed the percentage calculation.	Tool neither calculated nor showed the percentage calculation.	**Accepted**
Note			
If the abnormal data inputted to the tool, the tool did not calculate the coverage percentage			

The results of the testing of the tools can be described in Table V below:

TABLE V. TESTING RESULT

Test Case	**Test Result**	**Description**
Add Node	Passed	Tool successfully saved all nodes based on the value inputted by tester.
Create Path	Passed	Tool successfully saved all node which will create path inputted by tester
Add Path	Passed	Tool successfully saved path inputted by tester.
Reset Path	Passed	Tool successfully reset a path inputted by tester.
Add Definition	Passed	Tool successfully saved all definition variables and the node of them.
Add P-use	Passed	Tool successfully saved all of p-use variables and the edge of them.
Generate All P-uses	Passed	Tool successfully generated all of all p-uses paths based on properties inputted tester to the tool.
Calculate	Passed	Tool successfully calculated the all p-

Test Case	Test Result	Description
Coverage		uses coverage percentage
Reset	Passed	Tool successfully reset the calculation of all p-uses coverage percentage

V. RELATED WORK

There has been a similar work done by Purdue University [15]. They built a tool for data flow coverage testing called ATAC (Automatic Test Analysis for C). ATAC is a tool for evaluating test set completeness based on data flow coverage measures. It allows the tester to create new tests intended to improve coverage by examining code which is uncovered. It is currently implemented for UNIX for programs written in C language [16]. The differences between ATAC and our tool are as follows:

1. ATAC supports testing for C programs but our tool supports testing for Java and C programs.
2. The data flow coverage criteria which is used in ATAC is based on the coverage criteria definitions of Rapps and Weyuker [16] i.e. blocks, decisions, definitions, p-uses, c-uses, all-uses, and du-paths but ours focuses only on all p-uses.
3. ATAC works on Linux or Unix machines but ours can work on both Linux/Unix and Windows machines.
4. ATAC is a command line program whereas ours is a GUI-based tool.

VI. CONCLUSION

At the end of the research, it is concluded that the tool has already been able to successfully generate all of *all p-uses* paths, and then calculate and show the *all p-uses* coverage e of test cases that are entered as the inputs of the software under test.

The limitation of the tool is that prior to calculating the percentage, the tester must input properties of the software under test to the tool, such as nodes, paths, definition variables and the node, p-uses variables and the edges. This model of tool may force the prospective users of the tool to have knowledge about how to define the control flow graph including nodes, edges, and the usage types of the variables exist in the software under test.

In the future, it is worthy to expand the research by improving the tool so that the tool may be able to accept source codes of the software under test and a set of test cases as the inputs, and automatically calculate the *all p-uses* coverage of the test cases. With these full features, the tool will definitely assist testers to do white box testing using *all p-uses* coverage as the criterion. Furthermore, it will be a more powerful testing tool if it has more coverage criteria included in the features.

REFERENCES

[1] G. J. Myers. "The Art of Software Testing". *John Wiley and Sons, second edition, 2004*

[2] Collofello, J. S. dan Woodfield, S. N. "Evaluating the effectiveness of reliability-assurance techniques". *Journal System and Software. Vol. 9, No. 3, Hal. 191–195, 1989.*

[3] Jiantao Pan, "Software Testing", Carniege Mellon University, 1999.

[4] Mohd. Ehmer Khan, "Different Approaches to White Box Testing Technique for Finding Errors", *All Musanna College of Technology, Sultanate of Oman, in press.*

[5] Kelly J. Hayhurst, Dan S. Veerhusen, John J Chilenski, Leanna K. Rierson, "A Practical Tutorial on Modified Condition/Decision Coverage", *NASA, pp.7-11, 2001*.

[6] Janvi Badlaney, Rohit Ghatol, Romit Jadhwani, "An Introduction to Data-Flow Testing", *North Carolina State University, 2006, in pres*s.

[7] Kshirasagar Naik, Priyadarshi Tripathy, "Software Testing and Quality Assurance, Theory and Practice", *University of Waterloo, July 200*

[8] Clarke, L.A., A. Podgurski, D.J. Richardson, S.J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering, Vol. 15, No. 11, November 1989*.

[9] A. Aho, R. Sethi, and J. Ullman. "Compilers: Principles, Techniques and Tools". *Addison-Wesley, Reading, MA, 1986*.

[10] G. Ammons, T. Ball, and J. Larus. "Exploiting hardware performance counters with flow and context sensitive profiling". *ACM SIGPLAN Notices, 32(5):85-96, June 1997. Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. "Exe: automatically generating inputs of death". *In CCS '06: Proceedings of the 13th ACM conference on Computer and communications security, pages 322–335, New York, NY, USA, 2006. ACM Press.*

[12] P. Godefroid, N. Klarlund, and K. Sen. DART, "directed automated random testing". *SIGPLAN Notices, 40(6):213–223, 2005.*

[13] http://junit.org/

[14] http://www.seleniumhq.org/

[15] J. R. Horgan, S. London, "A data flow coverage testing tool for C", *Bellcore, Proceedings of the Second Symposium on Assessment of Quality Software Development, 1992, pages 2-10, IEEE Press.*

[16] J. R. Horgan, S. London, "Data flow coverage and the C language", *Bellcore, Proceedings of the symposium on Testing, analysis, and verification, 1991, pages 87-97, ACM Press.*