Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1709–1713
DOI: 10.15439/2016F289
ACSIS, Vol. 8. ISSN 2300-5963

Java-HCT: An approach to increase MC/DC using Hybrid Concolic Testing for Java programs

Sangharatna Godboley[1], Arpita Dutta[2], Durga Prasad Mohapatra[3]
National Institute of Technology Rourkela
Odisha, India
Email: sanghu1790@gmail.com[1], arpitad10j@gmail.com[2], durga@nitrkl.ac.in[3]

***Abstract*—Modified Condition / Decision Coverage (MC/DC) is the second strongest coverage criterion in white-box testing. According to DO178C/RTCA criterion it is mandatory to achieve Level A certification for MC/DC. Concolic testing is the combination of Concrete and Symbolic execution. It is a systematic technique that performs symbolic execution but uses randomly-generated test inputs to initialize the search and to allow the tool to execute programs when symbolic execution fails. In this paper, we extend concolic testing by computing MC/DC using the automatically generated test cases. On the other hand Feedback-Directed Random Test Generation builds inputs incrementally by randomly selecting a method call to apply and find arguments from among previously-constructed inputs. As soon as the input is built, it is executed and checked against a set of contracts and filters.**

In our proposed work, we combine feedback-directed test cases generation with concolic testing to form Java-Hybrid Concolic Testing (Java-HCT). Java-HCT generates more number of test cases since it combines the features of both Feedback-Directed Random Test and Concolic Testing. Hence, through Java-HCT, we achieve high MC/DC. Combinations of approaches represent different tradeoffs of completeness and scalability. We develop Java-HCT using RANDOOP, jCUTE, and COPECA. Combination of RANDOOP and jCUTE creates more test cases. COPECA is used to measure MC/DC% using the generated test cases. Experimental study shows that Java-HCT produces better MC/DC% than individual testing techniques(feedback-directed random testing and concolic testing). We have improved MC/DC by $\times$1.62 and by $\times$1.26 for feedback-directed random testing and concolic testing respectively.

I. Introduction

SOFTWARE Testing is the technique to detect bugs in software. Manual software testing accounts for 50-80% of the cost of software development. Manually created test cases are expensive, error-prone, and generally not exhaustive [2]. Therefore, automated software testing techniques have been discovered [3], [4].

There exists some controversy regarding the relative advantages of random testing and systematic testing. Some work [5], [6] suggest that random testing is same effective as systematic testing techniques. Existing work [7] found that random test case generation achieves less code coverage than systematic generation techniques. These systematic generation techniques include chaining, exhaustive generation, model checking, and symbolic execution.

Pacheco et al. [8] proposed Feedback-Directed Random testing. They have addressed random generation of unit tests for object-oriented programs. Their proposed work indicates that feedback-directed random generation retains the benefits of random testing (scalability, simplicity of implementation), and avoids redundant test cases.

Concolic testing is a systematic technique that performs symbolic execution. Concolic testing uses randomly generated test cases to start the search and to allow the tool to make progress when symbolic execution fails due to limitation of the symbolic technique (e.g. native calls) [9].

MC/DC is a criterion for code coverage and was introduced by the RTCA DO-178B standard [10]. MC/DC must satisfy the following criteria [11]:

- All the entry and exit points of the input programs must be invoked at least once.
- All possible outcomes of a decision must be affected by the changes made to each condition.
- All possible outcomes of every decision must execute.
- All the conditions in a decision must execute.

According to Majumdar et al. [2], in hybrid concolic testing, the concolic testing phase is initiated whenever random testing saturates, i.e, does not find new coverage points even after running a predetermined number of steps. Majumdar et al. [2] observed that CUTE and jCUTE tools have ultimately run up against path explosion. Concolic testing can only cover a small fraction of branches, those that can be reached using “short" executions from the initial state of the program. Therefore concolic testing requires “deep" program status to be explored.

We have implemented Java-Hybrid Concolic Testing using RANDOOP, jCUTE, and COPECA, and applied it to achieve high Modified Condition/Decision Coverage for Java programs.

The rest of the article is organized as follows: Section 2 presents the background concepts. Section 3 presents the proposed approach Java-HCT. Section 4 shows the experimental study. Section 5 compares the proposed approach with some of the existing approaches. Section 6 concludes the paper and suggests some future work.

II. Background Concepts

In this section we discuss some important background concepts, which are required to understand our work.

Definition 1: Feedback-Directed Random Testing: “It is a combination of random and systematic approach that results a test suite consisting of unit tests for the classes under test. Systematic approach deals with Feedback-Directed, i.e

as soon as an input value is built, it is executed and checked against a set of contacts and filters. The result of the execution determines whether the input is redundant, illegal or useful for generation of more input [8].”

There is a tool available for Feedback-Directed Random Testing called RANDOOP[1]. RANDOOP stands for Random Tester for object-Oriented Programs. It is a fully automatic tool and requires no input from the user, and scales to realize applications with hundreds of classes.

Definition 2: Concolic Testing: “Concolic testing is defined as a variant of symbolic execution where symbolic execution is run simultaneously with concrete executions, i.e., the program is simultaneously executed on concrete and symbolic values, and symbolic constraints generated along the path are simplified using the corresponding concrete values. The symbolic constraints are then used to incrementally generate test inputs for better coverage by combining symbolic constraints for a prefix of the path with the negation of a conditional taken by the execution [9], [15].”

JCUTE[2] is a Java concolic unit test engine based on concolic testing to execute Java programs.

Definition 3: Java-HCT: “Java-Hybrid Concolic Testing is the combination of Feedback-Directed Random Testing and Concolic testing for Java programs to result high MC/DC coverage.”

Java-HCT is implemented using RANDOOP, jCUTE, and COPECA. RANDOOP and jCUTE are open source testing tools and used performing Random testing and Concolic testing respectively. We have developed the tool COPECA (COverage PErcentage CAlculator), which is plugged into RANDOOP and jCUTE to measure MC/DC%, using the generated test cases. COPECA is based on Extended Truth Table.

Definition 4: Modified Decision / Condition Coverage: “MC/DC is some kind of Predicate Coverage technique, where condition is a leaf level Boolean expression and decision controls the program flow. MC/DC% is defined as the total number of independently affected conditions (I) out of total conditions (C) present in a program [11] mathematically.”

$$MCDC\% = \frac{|I|}{|C|} * 100\% \quad (1)$$

III. PROPOSED APPROACH: JAVA-HCT

In this section, we discuss the detailed and algorithmic description of Java-HCT followed by the proposed steps of the technique.

A. Overview

Our proposed technique Java-HCT consists of seven modules. These are i) Syntax_Converter, ii) RANDOOP, iii) jCUTE, iv) TCs Extractor, v) TCs Combiner, vi) TCs Minimizer, and vii) COPECA. These modules are shown in Fig. 1. Java-HCT accepts a Java program and produces MC/DC%.

[1]https://github.com/randoop/randoop-eclipse-plugin
[2]http://osl.cs.illinois.edu/software/jcute/

TABLE I
CHARACTERISTICS OF DIFFERENT TARGET PROGRAMS

Sl. No.	Program Name	LOC	# of Predicates	# of Conditions	# of Variables
1	SwitchTest	84	1	2	2
2	StringBuffer	1369	5	10	3
3	ScopeCheck	148	8	18	8
4	MyQuickSort	87	1	2	3
5	MathCall1	190	13	26	4
6	MyInsertionSort	70	2	6	4
7	Condition	60	4	9	3
8	FruitSales	267	23	69	4
9	InsertionSort	163	7	14	6
10	Comparison1	128	17	43	4
11	DSort1	136	10	20	2
12	GradeCalculation	103	6	12	1
13	MarketSales1	179	8	17	4
14	FruitBasket1	209	12	38	2
15	BSTree	307	6	13	3
16	SwitchTest2	104	6	16	5
17	AssertTest	75	3	7	3
18	BubbleSort	142	6	14	7
19	DSort_BST	305	3	7	3
20	CAssume	63	3	7	3
21	Demo1	76	3	8	2
22	MarketSales2	230	24	49	7
23	MathCall2	160	7	14	4
24	Selection_Sort	163	7	14	6
25	Sorting_algo	336	25	50	9
26	SwitchTest3	80	2	2	1
27	StringBuffer1	485	5	15	4
28	StudentGrades	67	5	10	1
29	Testy	53	3	6	1
30	Weight	39	1	3	3
31	Weight_Exp1	114	10	22	3
32	Weight_Exp2	77	5	13	3
33	Wildlife1	17	9	28	3
34	Wildlife2	199	13	40	3
35	Zodiac	104	18	84	10
36	WBS	321	5	10	3
37	AssertTest2	91	7	21	7
38	HelloWorld	44	2	4	2
39	IFExample	82	2	4	2
40	IFSample	95	6	12	3

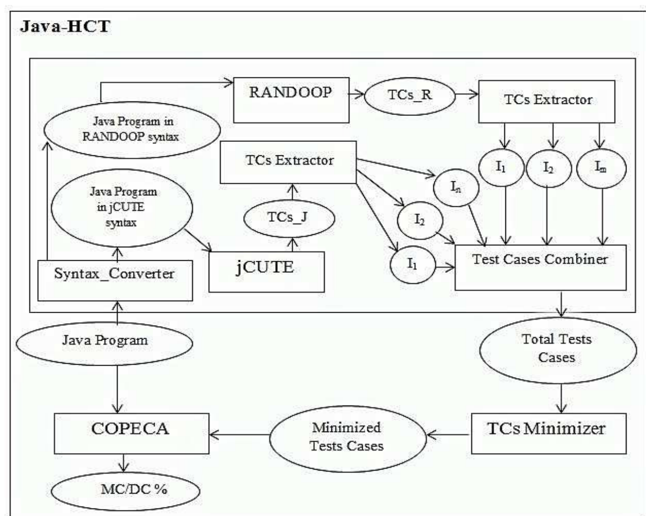

Fig. 1. Schematic representation of Java-HCT

Basically Java-HCT is the combination of RANDOOP and jCUTE which produce test cases combining. These tools RANDOOP and jCUTE are plugged into COPECA so that, the hybrid tool will be capable of computing MC/DC%. Our proposed technique provides deep as well as wide exploration of concolic execution.

B. Detailed Description

Java-Hybrid Concolic Testing is the best combination to achieve better MC/DC, and is the hybrid combination of Feedback-Directed Random testing and Java Concolic Testing. We have inspired from the core-idea proposed by Majumdar et al.[2]. They proposed a Hybrid Concolic Testing algorithm, that interleaves random testing with concolic execution to obtain both a deep and a wide exploration of program state space. They have implemented their algorithm on top of concolic tester (CUTE) and experimented to obtain high branch coverage for two large programs;*VIM 5.7* and *Red black tree*. Similarly, we extend the work of Majumdar et al.[2] by measuring MC/DC and that too for Java programs. Majumdar et al.[2] implemented their algorithm using undirected random testing and concolic testing, whereas we propose an efficient hybrid concolic testing for Java programs i.e. Feedback-Directed Random testing with concolic testing to obtain high MC/DC.

Fig. 1 shows the proposed tool for Java-Hybrid Concolic Testing (Java-HCT)[3]. Java-HCT is developed by integrating seven modules. The process starts by supplying a Java program. From Fig. 1 we can observe that, this Java program is converted into two different syntaxes using *Syntax_converter*. Since, we supply this Java program into both RANDOOP and jCUTE, it is essential to convert the original Java program into respective tool syntaxes. Now, the Java program in RANDOOP syntax is supplied to *Random tester for Object-Oriented Programs (RANDOOP)* to generate *TCs_R* automatically. Similarly, the Java program in jCUTE syntax is supplied to *Java Concolic Unit Testing Engine (jCUTE)* to generate *TCs_J* automatically. Unfortunately, TCs_R and TCs_J are not in same syntax. Therefore, *TCs Extractor* module is uses both the test suites to extract the input values those are present in TCs_R and TCs_J as described in Fig. 1. Then all extracted input values are supplied to *TCs Combiner* to produce Total test cases. Since, these test cases may be redundant and useless for MC/DC, therefore we have developed *TCs Minimizer* that accepts all the input values and checks which are essential to compute MC/DC percentage and removes rest of the those non-essential test cases. Now, the minimized test cases are supplied to *COverage PErcenatge CAlculator (COPECA)*. Since, we focus to increase MC/DC percentage, so we have developed this COPECA to measaure MC/DC percentage. COPECA accepts the minimized test cases along with the original Java program to produce MC/DC%.

[3]https://sourceforge.net/projects/java-hct/

C. Algorithmic Description

Algorithm 1 deals with the pseudocode of Java-Hybrid Concolic Testing (Java-HCT). We supply a Java program to our algorithm. Java-HCT to produce MC/DC%.

Algorithm 1 Java-HCT

Input: J (Java Program)
Output: MC/DC%

1: $J_R, J_J \leftarrow Syntax_Converter(J)$
2: $TCs_R \leftarrow RANDOOP(J_R)$
3: $TCs_J \leftarrow jCUTE(J_J)$
4: $Input_values \leftarrow TCs_Extractor(TCs_R, TCs_J)$
5: $Total_TCs \leftarrow TCs_Combiner(Input_values)$
6: $Minimized_TCs \leftarrow TCs_Minimizer(Total_TCs)$
7: $MC/DC\% \leftarrow COPECA(J, Minimized_TCs)$
8: **return** $MC/DC\%$

In Line 1 of Algorithm 1, the Syntax_Converter takes a Java program as input and produces a Java program in RANDOOP syntax (J_R) and a Java program in jCUTE syntax (J_J) as outputs. Line 2 shows the execution of RANDOOP tool. RANDOOP takes J_R as input and generates test cases (TCs_R) as output. Line 3 shows the execution of jCUTE tool. J_J as input and generates test cases (TCs_J) from jCUTE as output. Now, these two generated test case sets (TCs_R, TCs_J) are forwarded to Test Cases Extractor (TCs Extractor) modules to separate each input values as presented in Line 4.

Line 5 shows the execution of Test cases Combiner (TCs Combiner). This Combiner module collects all the input values created from TCs Extractor and generate a single set called Total Test Cases (Total TCs). Line 6 shows minimization of the test cases generated through Test Cases Minimizer (TCs Minimizer). This module produces the Minimized Test Cases (Minimized TCs).

Line 7 deals with the computation of MC/DC% through COPECA. COPECA takes the original Java program along with the Minimized TCs as input. Line 8 returns the final MC/DC% as output.

IV. Experimental Study

In this section we discuss our experimental setup, the result analysis, and threats to validity.

A. Setup

The experimental programs are ran on a computer system with 4GB of memory (RAM) Intel(R) Core(TM)i5 CPU 650 @ 3.20 GHz 3.19 GHz and 32-bit operating system.

B. Result Analysis

Table I deals with the characteristics of forty input Java programs. Column 3 shows the size of programs in Lines of codes (LOCs). Columns 4,5,6, show the Predicates, Conditions, and Variables respectively.

Table II presents the generated test cases and MC/DC% for RANDOOP, jCUTE, and Java-HCT. Column 3 shows the

TABLE II
RESULTS ON EXECUTION OF COPECA

Sl. No.	Program Name	RANDOOP TCs	jCUTE TCs	Total TCs	Minimized TCs	MC/DC_1%	MC/DC_2%	MC/DC_3%	Inc_1	Inc_2
1	SwitchTest	20	8	28	4	50	50	100	50	50
2	StringBuffer	12	9	21	20	40	50	80	40	30
3	ScopeCheck	20	25	45	30	77.77	83.33	100	23.23	16.67
4	MyQuickSort	5	5	10	3	100	100	100	0	0
5	MathCall1	70	10	80	45	16.66	46.15	69.23	52.57	23.08
6	MyInsertionSort	13	6	19	11	0	50	83.33	83.33	33.33
7	Condition	26	7	33	15	44.44	66.67	88.88	44.44	22.21
8	FruitSales	114	12	126	112	31.88	42.02	56.52	24.64	14.5
9	InsertionSort	28	10	38	20	71.42	78.57	85.71	14.29	7.14
10	Comparison1	93	27	120	81	27.90	41.86	58.13	30.23	16.27
11	DSort1	39	4	43	28	50	75	85	35	10
12	GradeCalculation	23	5	28	18	33.33	50	75	16.7	25
13	MarketSales1	43	8	51	23	52.94	64.70	88.23	35.29	23.53
14	FruitBasket1	271	8	279	59	39.47	50	60.52	21.05	10.52
15	BSTree	86	5	91	24	23.07	69.23	84.61	61.54	15.38
16	SwitchTest2	29	14	42	30	12.5	18.75	31.25	18.75	12.5
17	AssertTest	31	7	38	9	57.14	57.14	100	42.86	42.86
18	BubbleSort	43	8	56	21	35.71	42.85	64.28	28.57	21.43
19	DSort_BST	36	8	44	9	42.85	28.57	57.14	14.29	28.57
20	CAssume	73	6	79	10	71.42	85.71	100	28.58	14.29
21	Demo1	44	4	48	12	62.5	75	87.5	25	12.5
22	MarketSales2	313	11	324	78	69.38	73.46	73.46	4.08	0
23	MathCall2	38	11	49	20	57.14	64.28	71.42	14.28	7.14
24	Selection_Sort	53	9	62	18	35.71	42.85	50	14.29	7.15
25	Sorting_algo	343	9	352	73	28	50	70	42	20
26	SwitchTest3	5	11	16	4	50	100	100	50	0
27	StringBuffer1	15	7	22	23	86.66	86.66	100	13.34	13.34
28	StudentGrades	103	8	111	20	30	50	80	50	30
29	Testy	19	3	22	11	50	66.66	83.33	33.33	16.67
30	Weight	10	4	14	5	33.33	33.33	66.66	33.33	33.33
31	Weight_Exp1	25	10	35	33	95.45	95.45	95.45	0	0
32	Weight_Exp2	26	8	34	18	100	100	100	0	0
33	Wildlife1	40	6	46	32	7.14	17.85	53.57	46.43	35.72
34	Wildlife2	50	10	60	59	10	40	50	40	10
35	Zodiac	190	63	253	131	5.95	16.66	27.86	21.91	11.2
36	WBS	20	7	27	18	0	20	30	30	10
37	AssertTest2	42	13	55	40	38.09	66.67	76.19	38.1	9.52
38	HelloWorld	10	5	15	5	100	100	100	0	0
39	IFExample	12	7	19	7	50	100	100	50	0
40	IFSample	24	13	37	5	75	83.33	100	25	16.67

test cases generated by Feedback-Directed Random Tetsing. RANDOOP is the tool that generates these test cases. Column 4 presents the test cases generated by Java Concolic Unit Testing Engine (jCUTE). Column 5 shows the total test cases of RANDOOP and jCUTE. TCs Minimizer accepts these total test cases and only selects essential test cases according to MC/DC criterion. Column 6 presents the number of minimized test cases. Columns 7,8,9 deal with the MC/DC percentages achieved by RANDOOP, jCUTE, and Java-HCT respectively. These percentages are defined below:

Definition 5: MC/DC_1%: This MC/DC percentage is computed through RANDOOP and COPECA.

Definition 6: MC/DC_2%: This MC/DC percentage is computed through jCUTE and COPECA.

Definition 7: MC/DC_3%: This MC/DC percentage is computed through RANDOOP, jCUTE and COPECA or Java-HCT.

Column 10 and 11 deal with the increase in MC/DC. Column 10 is named as Inc_1 and shows the difference between MC/DC_1% and MC/DC_3% using Eq.2, whereas Column 11 named as Inc_2 shows the difference between MC/DC_2% and MC/DC_3% using in Eq.3.

$$\boxed{\text{Inc_1} = \text{MC/DC_3\%} - \text{MC/DC_1\%}} \quad (2)$$

$$\boxed{\text{Inc_2} = \text{MC/DC_3\%} - \text{MC/DC_2\%}} \quad (3)$$

We have experimented forty Java programs. We computed the values of Inc_1 and Inc_2 for these programs which are 29.91% and 16.26% (on average) respectively. According to the observation of our experimental study, Java-HCT achieved better MC/DC by $\times$ 1.62 as compared to RANDOOP and by $\times$ 1.26 as compared to jCUTE.

V. COMPARISON WITH RELATED WORK

Majumdar et al. [2] presented a hybrid concolic testing for C programs. They have proposed an algorithm that interleaved random testing with concolic testing to achieve both a deep and a wide exploration of program state space. They had implemented their algorithm on top of CUTE tool and applied it to achieve better branch coverage for two large C based applications. For the same testing budget, almost they obtained 4$\times$ branch coverage and 2$\times$ branch coverage of random testing and concolic testing, respectively. We inspired from Majumdar et al. [2]'s core idea and proposed a new technique called Java-Hybrid Concolic Testing, which is implemented in Java language. Java-HCT is the combination of Feedback-Directed Random Testing and Java Concolic Testing.

Ganai et al. [12] and Ho et al. [13] proposed a technique of VLSI design validation where a combination of formal (symbolic execution or BDD based reachability) and random simulation engines were used to improve the design coverage

for big scale designs. Our proposed approach combines the Feedback-Directed Random Testing and Java Concolic Testing for Java programs to obtain better MC/DC.

Pacheco et al. [14], [8] presented a technique that improved random test generation by incorporating feedback obtained from executing test cases as they were created. Their proposed approach produced a test suite consisting of Java unit tests for the classes to be tested. Their experimental study showed that, use of feedback-directed random test generation was far better than systematic and undirected random test generation in term of coverage and error detection. In our approach, we used this improved random testing with the combination of Java concolic testing to obtain high MC/DC.

Sen et al. [9] proposed concolic testing in Java version called jCUTE and it is available online. jCUTE automatically selects the input values both symbolically and concretely, simultaneously. In our proposed work, we used this jCUTE tool to form Java-Hybrid Concolic Testing and obtained increased MC/DC.

Godefroid et al. [15] proposed an improved random testing technique by providing Directed fashion (Systematic way) combined with symbolic execution to generate test input values. In our proposed work, we used feedback-directed random testing instead of only directed because feedback-directed provides better code coverage. According to Pacheco et al. [8] RANDOOP is better in completeness and scalability, as compare to other approaches like DART. So, we have chosen feedback-directed technique to use.

Godboley et al. [16] proposed an approach to improve distributed concolic testing. They have proposed an approach for code transformation that supported to enhance MC/DC by generating extra test cases for C programs. Godboley et al. [17], [18], [19] has also developed transformation techniques for object oriented Java programs. They have also proposed green computation of testing tools.

VI. Conclusion and Future Work

To improve existing concolic testing and obtain high Modified Condition/Decision Coverage (MC/DC), we proposed a novel technique called Java-Hybrid Concolic Testing (Java-HCT). This technique is called as hybrid because it is the combination of two testing techniques Feedback-Directed Random Test and Concolic Testing. We experimented Java-HCT for forty Java programs and found there is an increase of 29.91% and 16.26% (on average), when compared to feedback-directed random testing and concolic testing respectively. We have improved MC/DC by $\times 1.62$ and by $\times 1.26$ in comparison to feedback-directed random testing and concolic testing, respectively.

In our future work, we will extend the proposed work by plugging with some transformation techniques to obtain better results.

References

[1] Csallner C, and Yannis S. 2004. *JCrasher: an automatic robustness tester for Java*, Software: Practice and Experience, Volume(34), Number(11), 10.1002/spe.602 pages 1025–1050.

[2] Majumdar R, and Sen K, May 2007. “Hybrid concolic testing,” In proceedings of 29th International Conference on *Software Engineering 2007*, doi: 10.1109/ICSE.2007.41. ISSN 0270-5257 pages. 416–426.

[3] Bird D.L., and Munoz C.U., 1983.“Automatic generation of random self-checking test cases,” *IBM Systems Journal*, vol. 22, no. 3, pages. 229–245. doi:10.1147/sj.223.0229. 1983.

[4] Gupta N, Mathur A.P., and Soffa M.L., 1998. “Automated test data generation using an iterative relaxation method,” In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA: ACM, doi: 10.1145/288195.288321. ISBN 1-58113-108-9 pages. 231–244. [Online]. Available: http://doi.acm.org/10.1145/288195.288321

[5] Xia S, Vito B.D., and Muñoz C., 2005.“Automated test generation for engineering applications,” In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA: ACM. doi: 10.1145/1101908.1101951. ISBN 1-58113-993-4, pages. 283–286. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101951

[6] Xie T., Notkin D., and Marinov D, 2004. “Rostra: a framework for detecting redundant object-oriented unit tests,” In Proceedings of the 19th International Conference on *Automated Software Engineering*. doi: 10.1109/ASE.2004.1342737. ISSN 1938-4300, pages. 196–205.

[7] Visser W, Păsăreanu C.S., and Khurshid S.,2004. “Test input generation with java pathfinder,” In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA: ACM, doi: 10.1145/1007512.1007526. ISBN 1-58113-820-2 pages. 97–107. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007526

[8] Pacheco C., Lahiri S.K., Ernst M.D., and Ball T,2007. “Feedback-directed random test generation,” In *29th International Conference on Software Engineering. ICSE 2007.*. doi: 10.1109/ICSE.2007.37. ISSN 0270-5257, pages. 75–84.

[9] Sen K., and Agha G,2006. “CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools," *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA. Berlin, Heidelberg: Springer Berlin Heidelberg,* pages. 419–423. ISBN 978-3-540-37411-4. [Online]. Available: http://dx.doi.org/10.1007/11817963_38

[10] Ammann P, Offutt J, and Huang H, 2003.“Coverage criteria for logical expressions,” In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE ’03. Washington, DC, USA: IEEE Computer Society. ISBN 0-7695-2007-3. pages. 99–108.

[11] Kelly H.J., Dan V.S., John C.J., Leanna R.K., 2001. “A practical tutorial on modified condition/decision coverage,” Tech. Rep. Nasa.

[12] Ganai M.K., Aziz A., and Kuehlmann A.,1999. “Enhancing simulation with bdds and atpg,” In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. DAC ’99. New York, NY, USA: ACM. doi: 10.1145/309847.309965. ISBN 1-58113-109-7 pages. 385–390. [Online]. Available: http://doi.acm.org/10.1145/309847.309965

[13] Ho P.H., Shiple T., Harer K., Kukula J., Damiano R., Bertacco V, Taylor J, and Long J, 2000. “Smart simulation using collaborative formal and simulation engines,” In *Int. Conf. on Computer Aided Design (ICCAD)*, pages. 120–126.

[14] Pacheco C., Lahiri S.K., Ernst M.D., and Ball T, 2006. “Feedback-directed random test generation,” In *Technical Report MSR-TR-2006-125, Microsoft Research,*, pages. 75–84.

[15] Godefroid P., Klarlund N., and Sen K., 2005. “Dart: Directed automated random testing,” In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. New York, NY, USA: ACM, 2005. doi:10.1145/1065010.1065036. ISBN 1-59593-056-6 pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[16] Godboley S., Mohapatra D.P., Das A., and Mall R., 2016.“An Improved Distributed Concolic Testing", Software: Practices and Experiences, DOI: 10.1002/spe.2405

[17] Godboley S., Dutta A., Mohapatra D.P., Das A. and Mall R., 2016.“Making a concolic tester achieve increased MC/DC.,” Innovations in Systems and Software Engineering, pp.1-14, DOI:10.1007/s11334-016-0284-8 .

[18] Godboley S., Panda S., Dutta A. and Mohapatra D.P., 2016.“An Automated Analysis of the Branch Coverage and Energy Consumption Using Concolic Testing.,” Arabian Journal for Science and Engineering, pp.1-19, DOI:10.1007/s13369-016-2284-2.

[19] Godboley S., Dutta A., Besra B. and Mohapatra D.P., 2015, October. “Green-JEXJ: A new tool to measure energy consumption of improved concolic testing.,” *In proceedings of Green Computing and Internet of Things (ICGCIoT),IEEE*, pp. 36-41,DOI: 10.1109/ICGCIoT.2015.7380424.