CSIT (September 2017) 5(3):217–233
DOI 10.1007/s40012-017-0157-9

ORIGINAL RESEARCH

Green-J^3 Model: a novel approach to measure energy consumption of modified condition/decision coverage using concolic testing

Sangharatna Godboley[1] · **Arpita Dutta**[1] · **Durga Prasad Mohapatra**[1] · **Rajib Mall**[2]

Received: 25 February 2016 / Accepted: 20 January 2017 / Published online: 7 February 2017

Abstract The power and energy consumption of computer systems have posed challenges to the environment. Few researchers focused on energy consumption of software, some have measured the energy consumption of hardware. Software energy consumption deals with amount of power consumed over time for a particular software. We know that, software testing is now moving towards automation. We use different testing tools to perform testing on programs and software. We propose our approach in regards to contribute in the field of Green Software Testing. Modified condition/decision coverage and concolic testing are very critical practices in Aerospace and Nuclear safety critical systems. However, these methodologies do not take into consideration the amount of energy and power consumptions, which is an important issue in Green Software Testing. In this article, we propose an energy consumption analysis for modified condition/decision coverage using concolic testing. We have developed a testing tool called Green-J^3 Model (Green-JPCT JCUTE JCA Model). Our experimentation with forty-five Java programs shows that our approach makes on an average 21.32% increase in modified condition / decision coverage. Green-J^3 Model consumes 747.51 kJ of energy for forty-five programs.

Keywords Energy consumption · Concolic testing · MC/DC

✉ Sangharatna Godboley
sanghu1790@gmail.com

Arpita Dutta
arpitad10j@gmail.com

Durga Prasad Mohapatra
durga@nitrkl.ac.in

Rajib Mall
rajib@cse.iitkgp.ernet.in

[1] DOS Lab, Department of Computer Science and Engineering, National Institute of Technology Rourkela, Rourkela, Odisha, India

[2] Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal, India

1 Introduction

In 2007, a report by ITCs-Liteary published that IT industries were responsible for more than 2% of global carbon emission. One more report by 1E in 2007 estimated that a company using 10,000 PCs wasted approximately $165,000 in electricity bill every year due to the computer systems being left on overnight. The report concluded that the company could reduce atmospheric Carbon Dioxide (CO_2) content by approximately 1381 tons by switching off their computer systems. Consumption of power and energy of PCs is a very important issue that should be taken into consideration seriously. Green IT tries to minimize the energy consumption of Information and Communication Technologies, which are induced during software development life cycle. Now a days, the current practices mainly emphasized on the data centers. Green Information Technology idea deals with decreasing the environmental impacts. This is done through Information Technology solutions. Most of the Information and Communication Technology solutions are based on software, so the power and energy consumption of software are very crucial to analyze.

To improve software quality, software testing targets to detect and rectify possible bugs in a software. Traditionally, software testing engineers couldn't detect all possible errors in a software through software testing techniques. It

was a technically big challenge to produce test data that cover all execution paths in an automated testing approach, due to presence of many complex conditional expressions and loops. To address these issues, Dynamic Symbolic Execution analysis or CONCOLIC (Concrete and Symbolic) testing to produce test data for finding maximum possible paths of a program, is introduced. In accordance with DO178B/RTCA standard, code coverage testing is one of the acceptable types of testing. Coverage based testing is a white-box testing technique. There are several coverage based testing techniques such as Statement Coverage(SC), Condition Coverage(CC), Branch Coverage(BC), Modified Condition/Decision Coverage (MC/DC), and Multiple Condition Coverage (MCC) testing. Modified Condition/Decision Coverage (MC/DC) is mandatory for Aerospace and Nuclear safety critical systems. Energy Consumption is a core area of concern for Green IT, Green Software Engineering, and Green Software Testing, whereas energy consumption of concolic and MC/DC testing are not yet proposed.

In this paper, we propose a framework to measure the modified condition/decision coverage and the amount of energy consumption in this process. We have named our tool Green JPCT JCUTE JCA Model (Green-J^3 Model). The Green-J^3 Model consists of mainly five modules. (1) Java Program Code Transformer (JPCT), (2) Java Concolic Tester (JCUTE), (3) Java Coverage Analyzer(JCA), (4) JouleMeter, and Energy Calculator. In this proposed approach, we develop JPCT to improve the MC/DC of Java programs by converting the original Java program into a transformed version. Our tool Green-J^3 Model uses JCUTE to generate test data. We have developed JCA to compute MC/DC percentage of the original and transformed Java programs. To measure the energy consumption in this process, Green-J^3 Model uses the tool JoulMeter. The abbreviations, we used in our work are listed in Table 1 for easy reference.

The rest of the paper is organized as follows: Sect. 2 deals with some of the fundamental concepts. Section 3 explains the proposed Green-J^3 Model and discusses the proposed algorithms. Section 4 explains the experimental study of our proposed work. In Sect. 5, we present some important threats to validity. In Sect. 6, we compare our work with some of the existing related work. Section 7 concludes the paper with some insights into our future work.

Table 1 Abbreviations used in our approach

Sl. no.	Short-form	Full-form
1	AC	Ammeter clamps
2	APCT	Advanced program code transformer
3	BC	Branch coverage
4	BCT	Binary code transformer
5	CA	Coverage analyzer
6	CONBOL	CONcrete and symBOLIic testing
7	CONCOLIC	CONCrete +symbOLIC
8	DAE	Data aggregator and evaluator
9	EC	Energy consumption
10	EDTSO	Energy directed test suite optimization
11	Green-ABCE	Green architecture model for branch coverage enhancement
12	Green-IT	Green Information Technology
13	Green SE	Green Software Engineering
14	ICTs	Information and communications technologies
15	JPCT	Java program code transformer
16	PC	Power consumption
17	PCT	Program code transformer
18	PM	Power model
19	Power TOP	Power temporal optimization
20	SCORE	Scalable COncolic testing tool for reliable embedded software
21	SUT	System under test
22	VIM	Virtual instrument machine
23	WG	Workload generator
24	WLS	Work loader simulator
25	XNCT	Exclusive-NOR code transformer

2 Fundamental concepts

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

2.1 Green Software Engineering

To classify and sort some concerns of green and sustainable software and their engineering technology, Kern et al. [1] developed GREENSOFT Model as shown in Fig. 1. GREENSOFT Model consists of the following four parts: Life cycle of software products, Sustainability criteria for software products, Procedure models, and Recommendations for Action and tools. Green Software Engineering was the attempt to apply the "green" principles known from hardware products on software products, software development processes and their underlying software process models. For more detail knowledge on Green Software Engineering, the readers are suggested to refer [1, 2].

2.2 Energy consumption

The System Under Test (SUT) is the computer hosting the application whose induced energy consumption will be computed. The power readings define the energy consumption recorded with respect to timestamps. The total energy consumption for the entire run may be simply obtained by summing up the power values measured in Watts for the duration of interest. Since each value represents the power used in one second, the sum gives the total energy consumption in Joules.

2.3 Branch coverage

For achieving branch coverage, each decision in a program, should take all possible outcomes at least once, i.e. either true or false.

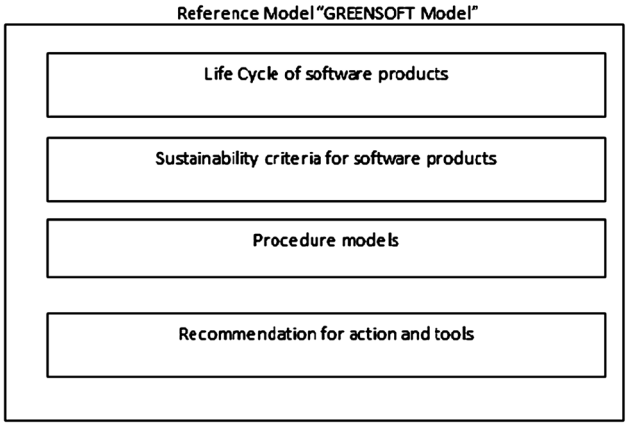


Fig. 1 GREENSOFT Model [1, 2]

Table 2 Truth table for "$if(A \bigwedge B)$"

TC no.	A	B	$A \bigwedge B$
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

Table 3 Extended truth table for "$if(A \bigwedge B)$"

TC no.	A	B	$A \bigwedge B$	A	B
1	T	T	T	3	2
2	T	F	F		1
3	F	T	F	1	
4	F	F	F		

2.4 MC/DC

The RTCA standard of DO-178B stands for Radio Technical Commission for Aeronautics [3]. It is mandatory to achieve Modified Condition /Decision Coverage (MC/DC) for Level A certificate of safety critical applications [4]. MC/DC necessitates satisfaction of the followings:

- 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.

Let us discuss MC/DC with an example. Consider a program containing only one complex condition "$if(A \bigwedge B)$". Table 2 shows the truth table for "$if(A \bigwedge B)$" and Table 3 shows the Extended truth table for the condition taken, which represents the core idea of MC/DC. From Table 3, we can observe that the test cases (TC) numbers {1,2,3} are the unique test cases that satisfy the MC/DC criterion for the given condition. Thus, the resultant test cases are {(T, T), (T, F), (F, T)}. Here, both the conditions "A" and "B" are independently affected by changing the value of each condition and by observing the fact that the final outcome of the whole decision is changed. Therefore, two independently affected conditions out of two simple conditions, show that 100% MC/DC is achieved. It may be noted that the test suite to achieve MC/DC for a given program is in general not unique. It may be observed that the test cases (T, T) is needed as it is the only one that returns a *true* output value. The test case (F, T) is needed as it is the only test case that modifies the value of A and also it affects the outcome of the complex condition,

by establishing the independent influence of A. In a similar way, the test cases (T, T) and (T, F) are nedded to show the independent influence of B.

2.5 Concolic testing

Concolic testing is the combination of concrete and symbolic execution, that performs exhaustive testing. Concolic testing was proposed to automatically generate test data that can execute all the reachable branches of a program [5]. It first generates a random input value. Then, it simulates execution of the program with the input values. Simultaneously, the symbolic constraints at every branch point with the execution path are collected. A set of selected input values corresponding to each of the branch point is made available at the end of the execution path. A *path constraint* is the conjunction of all these constraints. A component from a path constraint is chosen and it is negated to explore a new path. The last branch point from a path is generally chosen for negation but the selection can also be done randomly. A constraint solver is used to solve a new path constraint to generate some concrete input values that satisfy these paths. For more information on this approach, the reader is referred to [5].

We explain the concolic testing approach using an illustrative example. Consider the function *weightCategory* shown in Fig. 2. The process starts by executing the program with randomly generated inputs. Suppose the parameters *mass* and *length* are randomly set to 22.0 and −5.0. During execution, both the concrete and symbolic values (22.0 and $mass_0$) are selected for the specific

```
1 static int weightCategory (double mass, double length) {
2       if (mass > 0.0 )
3          continue;
4       else
5          return ;
6       if (length > 0.0 )
7          continue;
8       else
9          return;
10
11       double bmi = mass / (length * length);
12
13       if (bmi < 18.5)
14          return UNDERWEIGHT;
15
16       else if (bmi < 25.0)
17          return NORMAL;
18
19       else
20          return OVERWEIGHT;}
```

Fig. 2 A sample program

executed path. The first branch instruction is encountered at Line 2. Here, the output of thepredicate evaluates to true because *mass* is set to 22.0. For an input value to take the same branch, it is necessary that the branch constraint ($mass > 0.0$) holds.

The next branch point is detected when executing the *if* expression in line 6. Now the predicate evaluates to false, because *length* is set to a negative value. The branch constraint associated with this branch is:

$$\neg(length > 0.0) \tag{1}$$

This branch constraint value is merged with the preceding branch constraint value to form the following new path constraint:

$$(mass > 0.0) \wedge \neg(length > 0.0) \tag{2}$$

After line 8 is executed, the *else* expression of the function *weightCategory* exits. The path constraint is obtained by negating one of the branch constraints. When the last branch constraint is negated, the resultant path constraint becomes:

$$(mass > 0.0) \wedge (length > 0.0) \tag{3}$$

This new path constraint is then passed to a constraint solver to check if there exists a set of input values that makes the constraint true. A solution that satisfies this constraint is *length* = 1.0 and *mass* = 50.0. In the next loop iteration, the function is exercised with this set of input values and the path constraint value is again collected. The execution affects the function to return the VALUE *OVERWEIGHT*. This execution path corresponds to satisfaction of the following constraint:

$$(mass > 0.0) \wedge (length > 0.0) \wedge \neg(bmi < 18.5) \wedge \neg(bmi < 25.0) \tag{4}$$

This step of manipulating the symbolic path constraint values, solving them to generate concrete inputs, and executing the resulting test inputs is repeated until a predeclared stopping criterion is met. The stopping criterion is usually either when the number of iterations exceeds a prespecified threshold or when all the reachable branches are covered.

The constraint solver is invoked with another modified path constraint when it fails to calculate a set of test cases that satisfy the modified path constraints. The new modified path constraint is found by negating some other branch constraint value of the previous path constraint value. When a constraint solver is failing to evaluate any test cases for a path, the path is considered to be infeasible, i.e. no input exists that can satisfy the constraint.

Algorithm 1 Green-J^3 Model

Input: J^3 Model ▷ *JPCT, JCUTE, and JCA modules*
Output: Total time taken(T), Power Consumption (PC), Energy Consumption (EC)
Begin

1: Start executing JouleMeter to compute power consumption with respect to timestamps of 1 second period.
2: Start saving all energy consumption values.
3: Generate TS_1 using JCUTE and compute MC/DC_1% using JCA. ▷ *TS_1 and MC/DC_1% are values recorded without JPCT*
4: Execute JPCT to convert Java program to the transformed version.
5: Generate TS_2 using JCUTE for the transformed program and compute MC/DC_2% using JCA for original program. ▷ *TS_2 and MC/DC_2% are values recorded with JPCT*
6: Measure the time taken to compute MC/DC, speed of test case generation, difference between MC/DC_2% and MC/DC_1%. ▷ *Achieving high Modified Condition/ Decision Coverage*
7: Stop saving all energy consumption values.
8: Stop execution of JouleMeter which was recording the process.
9: Measure Total time taken ($T_m - T_n$), and Total power consumption (PC).
10: Compute Energy consumption using:

$$EC = (T_m - T_n) \times \sum_{n}^{m} (PC_n) \tag{5}$$

where, n ⟶ first timestamp and m ⟶ last timestamp
11: Exit

3 Proposed approach: Green-J^3 Model

In this section, we discuss about the proposed Green-J^3 Model in detail. First, we discuss the block diagram of Green-J^3 Model. Then, we present the algorithmic description in detail.

3.1 Block diagram of Green-J^3 Model

Figure 3 represents the schematic representation of Green-J^3 Model. The block diagram consists of mainly five modules: JPCT, JCUTE, JCA, JouleMeter and Energy Calculator. The flow starts with the initiation of recording power consumption for each 1 second of JPCT, JCUTE, and JCA by JouleMeter. JPCT takes a Java program as input and converts it into its transformed version. Then, the transformed program is supplied to JCUTE to generate test cases automatically. Now, the original program and the generated test cases are fed into JCA to compute the MC/DC. After these steps are performed, we stop recording the power consumption. Finally, using the total time taken by the above process and total power consumed by the technique, our tool Green J^3-Model measures the total energy consumption in Joules through Energy Calculator. Therefore, through Green-J^3 Model, we measure the time taken, power consumed, and energy consumed for our proposed software testing technique.

3.2 Algorithmic description of our proposed approach

In this section, we discuss the algorithmic description of the Green-J^3 Model and J^3 Model in more detail.

3.2.1 Green-J^3 Model

The overall pseudo-code for the proposed approach is mentioned in Algorithm 1. We can observe from Algorithm 1 that the proposed process requires the followings: (1) a Java program, (2) JPCT to transform the Java program, (3) JCUTE concolic tester to generate test cases, (4) JCA to measure MC/DC percentage, (5) JouleMeter to measure energy consumption and (6) Energy Calculator. Finally Algorithm 1 results in the total Energy Consumption(EC).

First two steps of Algorithm 1 are performed by JouleMeter to start the process. Step 3 deals with generation of Test_Suite_1 (TS_1) using JCUTE and measuring MC/DC_1% using JCA. The first experiment shows the execution of Java program without JPCT. Step 4 shows the execution of Java program through JPCT to convert it into transformed version. Step 5 shows the generation of Test_Suite_2(TS_2) using JCUTE and measures MC/DC_2% using JCA. It may be noted that in Step 5, the test suite is generated for the transformed Java program and computation of MC/DC_2% is done for the original Java

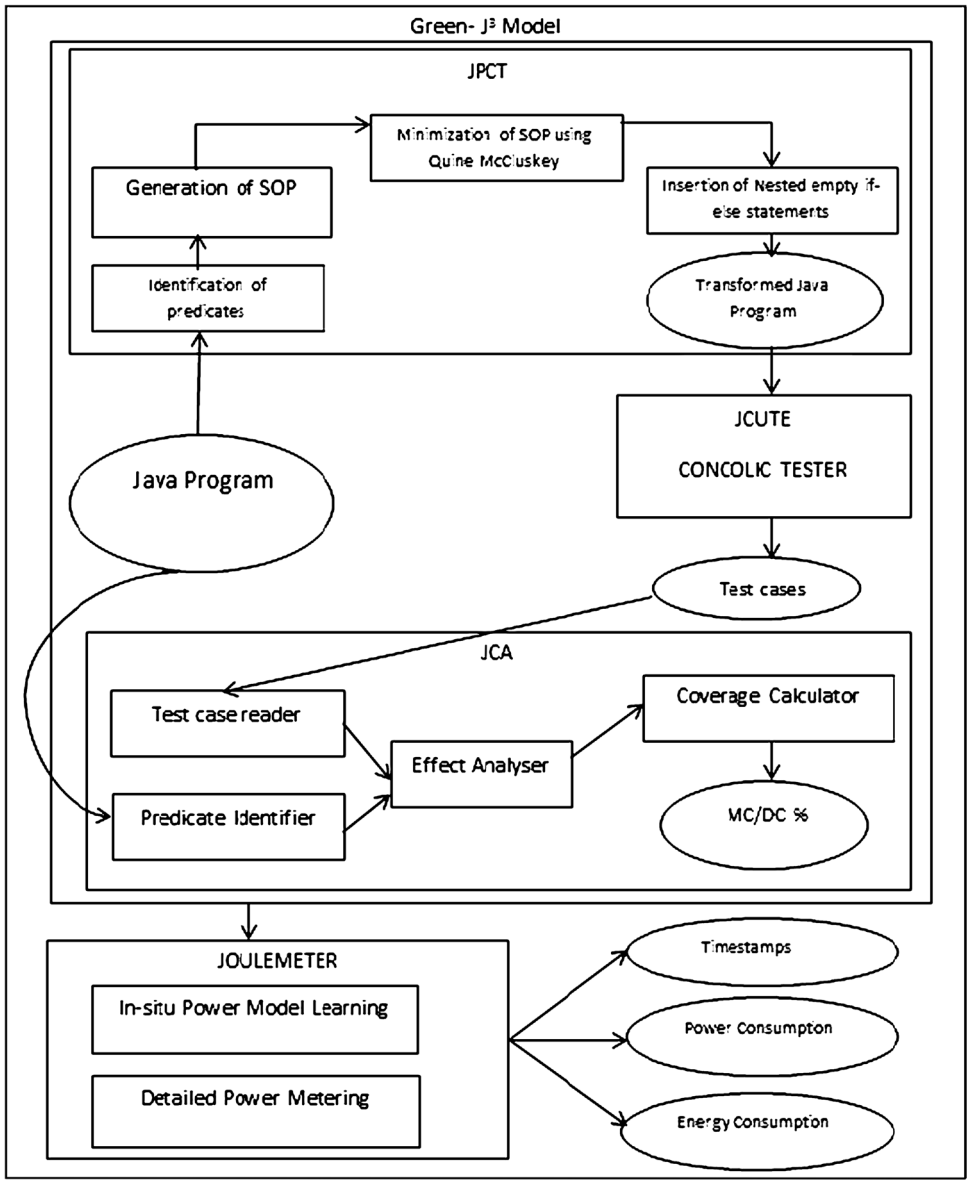


Fig. 3 Schematic representation of Green-J[3] Model

program and TS_2. Step 6 finds the time taken to compute MC/DC, speed of test case generation, and difference of two MC/DCs. Steps 7 and 8 stop saving the energy consumption values and execution of JouleMeter. Step 9 finds the total time taken and power consumption. Step 10 finally computes the total energy consumption in Joules.

Here, we discuss about JouleMeter. JouleMeter [6] is a Microsoft developed open source tool. It is easily available on Internet[1]. It is used to measure the power consumption of a software application, running in the computer system. JouleMeter computes the power usage of an application through a Power Model. The Power model relates the hardware power state and the computer resource usage in power consumption calculation. The hardware power state consists of processor frequency, monitor off/on state, screen brightness, process utilization and disk utilization. But in our proposed work, we have considered only a specific software application's power usage. JouleMeter sets some power number parameters for the proper running of Power Model. The power number parameters are: Processor Peak Power (PPP—high frequency), Base Power, Processor Peak Power (PPP—low frequency) and monitor power. The power consumption of any application program indicates only the power consumption of the specific application program. The energy consumption of any software application is computed by adding the power usage per time stamp from the start of the application to the end of execution of the application. Then, the total power usage is multiplied with the time interval to get the energy consumption, which is defined in Energy Calculator.

[1] http://research.microsoft.com/en-us/projects/joulemeter/.

3.2.2 J^3 Model

Algorithm 2 presents the concept of J^3 Model, that accepts a Java program as input and produces the MC/DC percentage as output. Steps 1–7 of Algorithm 2 are executed to perform JPCT. Initially program J is fed into JPCT to result in the transformed program. Step 2 identifies the predicates in the Java program under test. Steps 3 and 4 form sum of product (SOP) for each and every identified predicate. This SOP may be complex in nature, therefore we apply Quine–McCluskey method to minimize the SOP. Now, to test the parts of predicate more exhaustively and rigorously, we use Boolean Derivative Method. By using this method we insert empty nested if-else conditional expressions in the original Java program. These extra conditional expressions contribute to enhance MC/DC, since they help to generate more test cases so that they cover more branches and explore maximum paths of the execution tree while compiling and executing through concolic tester. Step 7 gathers all the statements and original programs that form the transformed version of the original Java program. Step 8 deals with execution of the transformed program through JCUTE and generates test cases. Steps 9–14 deal with JCA to measure MC/DC. In Step 9, we supply the generated test cases and original Java program into JCA. In Step 10, the Test case reader reads each test case and subsequently the predicate identifier identifies all predicates in the input Java program in Step 11. Now, in Step 12 we apply the concept of MC/DC to identify the independently affected conditions (I) and Number of Simple Conditions (C). Coverage Calculator uses these two values I and C, to compute MC/DC percentage as presented in Step 13. In Step 14, the algorithm exits.

Let us consider the function *testLogical* shown in Fig. 4. The function accepts three Boolean values as parameters and based on the satisfaction of the decision *(a&& (b || c))* (Line 2), the *true* and *false* branches are executed. This technique transforms the function *testLogical* into the code with the dummy branches as shown in Fig. 5 to achieve MC/DC. The expressions shown in Lines 2–13 ensure to reach each branch, due to which we achieve high MC/DC.

Algorithm 2 J^3 Model

Input: J ▷ *Program J is in Java syntax*
Output: MC/DC percentage
Begin

1: JPCT accepts a Java program J as input and converts it into a transformed Java program J′.
2: JPCT identifies the boolean operators of J, wherever it finds && or || operators and copy the whole line and saves it to one text file.
3: For each identified predicate, convert each condition with variables and operators with '.'and '+' with respect to && or || respectively.
4: Obtain the Sum Of Product (SOP) form from Step 3. Here, SOP may be complex in nature, so to minimize the SOP it is required to use a minimization technique.
5: Perform Quine-McCluskey minimization technique to minimize SOP under process.
6: Apply Boolean Derivative Method to insert empty nested if-else expressions for each minimized SOP.
7: Collect all extra conditional statements and merge them with the original program to form the transformed Java program.
8: Execute concolic tester i.e. JCUTE for the transformed Java program and generate the test suite.
9: Supply program J and test cases to JCA to measure MC/DC%.
10: Sub-module test case reader of JCA reads the test suite.
11: Again JCA's sub-module predicate identifier detects all predicates.
12: Effect anlayzer module of JCA uses concept of MC/DC method to identify the independently affected conditions and total number of simple conditions using the identified predicates and generates test cases.
13: JCA uses coverage calculator to compute MC/DC using the following formula:

$$MC/DC\_\% \leftarrow \frac{\sum_u^v (List\_of\_I)}{\sum_u^v (List\_of\_C)} \times 100 \tag{6}$$

where, u ⟶ First condition, v ⟶ Last condition, List_of_I ⟶ Independently affected conditions, and List_of_C ⟶ Total number of simple conditions.
14: Exit.

End

```
1. void testLogical(bool a, bool b, bool c){
2.       if(a && (b||c))
3.       {.........}
4.       else
5.       {.........}
6. }
```

Fig. 4 Example function with a boolean expression

```
1.  void testLogical(bool a, bool b, bool c){
2.       if(a)
3.       {
4.         if(b){}
5.         else{}
6.       }
7.       else{}
8.       if(a)
9.       {
10.         if(c){}
11.         else{}
12.       }
13.       else{}
14.       if(a && (b||c))
15.       {.........}
16.       else
17.       {.........}
18.  }
```

Fig. 5 Transformed code of the example function given in Fig. 4 to achieve MC/DC

4 Experimental studies

We did our experimentation on forty-five benchmark Java programs taken from the OSL Repository[2] and some students assignments. We have our experimental setup as follows: We ran our programs on Windows-7 operating system with 4 GB RAM and Intel(R) Core(TM) i5 CPU having 2.40 GHz as processing speed. We have performed our experiments in Java environments JDK 1.6.0 and JRE 1.6.0. Table 4 explains the different properties of the Java programs which we have considered. Column 3 shows LOCs of original Java program and Column 4 shows LOCs of the transformed Java program. Here, we have considered programs up to a maximum of approximately 3000 LOCs. Columns 5 and 6 deal with the number of functions invoked and number of classes present in a Java program, respectively. In our proposed work, we basically work on concolic and MC/DC testing that strictly require predicates and branches covered. Columns 7 and 8 present the number of identified predicates and branches of the execution tree, respectively. Some important properties such as number of variables, DFS (Depth First Search) information, and errors are reported in Columns 9, 10, and 11 respectively.

For our experimentation, we have considered two scenarios. These two scenarios are defined below:

Scenario 1 It is the process of test case generation and measuring MC/DC percentage through JCUTE and JCA respectively.

Scenario 2 It is the process of transforming a Java program into it's transformed version, generating the test cases, and measuring the MC/DC percentage through JPCT, JCUTE and JCA respectively.

Table 5 presents the important parameters for our experiments. We use JCUTE to test a Java program in a concolic manner. JCUTE automatically selects input values for different constraints to explore the possible paths in a execution tree. These different input files are the test cases, that are generated automatically through JCUTE tool. Total number of test cases generated for Scenarios 1 and 2 are reported in Column 3. It may be observed that for thirty-seven programs, Scenario 2 produces more test cases as compared to Scenario 1 due to the transformation of Java programs. In support to MC/DC percentage metric to improve quality of software, we measure total computational time to compute MC/DC and speed of test case generation. Column 4 presents the computational time for both the Scenarios 1 and 2. In Scenario 1, we have two modules i.e. JCUTE and JCA. We record the time values individually, then sum up them for getting the total time. On the other hand for Scenario 2, we have three modules i.e. JPCT, JCUTE, and JCA. Again, we record the time values individually, then sum up them for setting the total time. It may be observed that for almost thirty-two programs, Scenario 2 takes little more time as compared to Scenario 1. The reason is the use of additional module for obtaining the transformed version of the program in Scenario 2. It may be that, since we have two scenarios, therefore, we have executed our experiments for two times. Different execution of modules take different times.

We have already mentioned that, we record time constraints individually for each modules. So, using time values through JCUTE and JPCT+JCUTE, we compute the speed of test case generation for Scenarios 1 and 2 respectively. Column 5 reports the speed of test case generation for Scenarios 1 and 2. We observed that for thirty programs Scenario 2 is delayed as compared to Scenario 1, due to the extra module JPCT added in Scenario 2.

Column 6 deals with MC/DC percentage analysis. We compute MC/DC_1% and MC/DC_2% for Scenario 1 and 2 respectively. To show enhancement in MC/DC, we take the difference of both the MC/DCs and this has been shown in third sub-column of Column 6. Here, we may observe that for thirty-one programs, we successfully achieved increased MC/DC. Among forty Java programs,in an average we achieved 24.03% high MC/DC. Figure 6 shows the MC/DC percentage analysis of the forty programs. In Fig. 6, the program are taken along X-axis. Y-axis represents the MC/DC percentage. Figure 7 presents the comparison of the two MC/DCs i.e. MC/DC_1 and MC/DC_2.

[2] http://osl.cs.illinois.edu/software/jcute/index.html.

Table 4 Characteristics of different experimental programs

SL. no.	Program name	LOC invoked	LOC′	# of function classes	# of predicate	# of branches	Total #	# of variables	DFS information	Errors
1	Condition	21	32	1	1	2	15	3	0/0/1	0
2	Weight	24	42	1	1	1	20	4	1/0/3	0
3	QuickSort	68	76	3	4	2	18	8	1/1/2	0
4	Nonce	286	350	10	11	25	146	14	0/2/3	1
5	StringBuffer	421	466	8	4	17	56	6	0/0/2	0
6	SwitchTest	59	71	2	3	4	15	5	1/1/3	0
7	Producer Consumer	20	27	4	2	2	8	4	2/1/4	2
8	BSTree	255	297	3	5	6	28	8	1/3/1	0
9	ArraySort	27	33	2	2	2	15	3	0/0/2	0
10	Static InstanceProblem	24	32	1	1	1	5	2	1/0/3	1
11	CAssume	43	65	1	1	1	2	1	0/0/1	0
12	CTest3	94	117	2	3	2	4	2	0/0/2	1
13	Deadlock	91	125	1	1	1	2	1	0/0/0	0
14	Demo	50	72	1	1	1	2	2	0/0/1	0
15	DemoLock	67	95	1	2	1	2	2	0/0/1	0
16	CTest2	96	114	2	3	2	4	4	0/0/2	0
17	CTest1	96	126	0	3	2	2	4	0/0/2	3
18	DemoL0ck2	60	97	1	1	1	2	2	1/1/4	0
19	DSort	95	121	5	3	4	18	4	16/1/18	1
20	ErrorTest	71	98	0	1	4	9	3	0/0/0	0
21	If-Else	77	118	1	2	1	6	2	1/1/3	2
22	InterfaceJava	51	350	1	1	1	2	1	1/1/1	0
23	LockHeld	73	97	2	1	4	12	4	0/0/0	0
24	MultiLock	62	106	1	1	2	4	3	3/3/3	0
25	NewJava	61	118	1	1	1	4	2	0/0/0	0
26	NoPredictive	69	108	2	3	3	8	2	0/0/1	0
27	NoPredictive2	58	84	2	3	2	4	2	0/0/2	4
28	NS	313	715	10	4	24	146	15	28/1/43	0
29	NS2	403	806	11	4	28	188	19	0/0/46	0
30	FinallyTest	45	45	0	1	0	0	2	0/0/0	0
31	OneWrite	45	45	0	1	0	3	1	0/0/3	24
32	Regression	176	256	1	1	15	8	24	2/1/5	0
33	SampleXA-CMLPolicy	90	125	2	1	1	24	5	0/0/6	0
34	StackTest	33	47	0	1	1	0	1	0/0/0	0
35	Static InstanceProblem2	51	68	1	1	1	4	2	0/0/2	0
36	StringBuffer Module	1364	1896	7	3	9	52	15	0/0/8	0
37	StringBuffer1	486	976	8	3	8	60	18	13/4/21	1
38	StringBuffer2	541	1012	8	3	10	64	22	13/4/21	1
39	Struct	47	95	1	1	2	8	3	2/1/5	1
40	Testme	51	78	1	1	1	4	3	0/0/2	1
41	Copeca	2043	2744	24	6	118	433	102	19/3/72	48
42	Smup	1849	2102	17	3	78	392	83	28/0/5	6
43	Pbct	708	1284	8	2	46	128	28	0/0/56	13
44	Gecco	1512	1931	15	3	53	272	75	1/14/3	37
45	JExj	1246	1764	13	3	62	188	96	2/19/33	19

Table 5 Result analysis for different programs

Sl. no.	Program	# of generated test cases		Computational time (ms)		Speed of test case generation (# of TCs/s)		MC/DC% analysis		
		Scenario 1	Scenario 2	Scenario 1	Scenario 2	Scenario 1	Scenario 2	Scenario 1	Scenario 2	Difference
1	Condition	4	5	4827	6159	3.41	2.13	100	100	0
2	Weight	5	6	6278	9174	3.15	1.39	66.66	100	33.33
3	QuickSort	1	1	7118	8841	3.36	0.88	50	66.66	16.66
4	Nonce	24	24	28533	32585	1.51	0.87	66.66	76.47	9.81
5	StringBuffer	8	8	135081	164548	0.06	0.04	75.51	83.67	8.16
6	SwitchTest	6	7	4229	5244	6.06	5.16	75	100	25
7	ProducerConsumer	1	2	3060	3616	2.02	3.12	60	80	20
8	BSTree	6	8	12323	13031	2.22	2.42	61.90	76.19	14.29
9	ArraySort	4	5	7903	8642	3.69	3.59	42.85	71.42	28.37
10	StaticInstanceProblem	3	3	4220	4847	8.08	6.12	100	100	0
11	CAssume	2	2	6025	5797	6.62	1.29	100	100	0
12	CTest3	2	3	6453	6796	6.66	2.96	40	80	40
13	Deadlock	1	2	2693	6021	5.68	0.75	100	100	0
14	Demo	1	3	3629	5545	6.53	1.60	50	100	50
15	DemoLock	1	3	3408	3702	6.49	2.54	33.33	66.66	33.33
16	CTest2	2	5	5153	4775	6.36	5.06	50	75	25
17	CTest1	1	2	6199	5812	2.36	1.94	33.33	66.66	33.33
18	DemoL0ck2	3	7	5303	6724	6.65	4.69	100	100	0
19	DSort	5	12	7651	7176	2.73	4.81	50	83.33	33.33
20	ErrorTest	1	2	1257	976	7.14	7.81	0	0	0
21	If-Else	4	6	4269	5260	4.98	2.30	66.66	83.33	16.67
22	InterfaceJava	2	3	4056	5719	6.57	2.43	50	100	50
23	LockHeld	1	2	2546	3528	6.57	1.56	33.33	66.66	33.33
24	MultiLock	1	1	4924	6132	5.10	1.75	20	60	40
25	NewJava	1	1	5825	6982	4.95	0.43	66.66	100	33.33
26	NoPredictive	2	5	5832	8556	6.92	1.77	42.85	85.71	42.86
27	NoPredictive2	4	6	5382	6015	6.71	2.56	25	80	55
28	NS	43	53	208538	202172	0.21	0.26	56.45	75.80	19.35
29	NS2	30	47	196481	190121	0.15	0.25	54.43	77.21	22.78
30	FinallyTest	1	1	1234	1133	6.66	5.34	0	0	0
31	OneWrite	1	1	7179	4618	0.30	0.41	0	0	0
32	Regression	5	12	2919	3653	6.53	5.74	73.07	94.23	21.16
33	SampleXACMLPolicy	11	19	8347	9042	5.21	4.92	33.33	100	66.66
34	StackTest	1	2	1565	1390	7.14	4.03	100	100	0
35	StaticInstanceProblem2	3	4	4127	3055	6.81	8.51	50	100	50
36	StringBufferModule	6	11	7150	31933	3.10	0.40	51.51	77.75	24.24
37	StringBuffer1	8	17	74966	83003	0.11	0.22	77.77	92.59	14.82
38	StringBuffer2	8	19	75591	88166	0.11	0.22	54.54	81.81	27.27
39	Struct	5	7	3849	4857	6.80	3.04	60	100	40
40	Testme	3	5	5688	5548	6.60	3.51	33.33	66.66	33.33
41	Copeca	71	33	12385	13013	5.732	7.146	73.52	93.13	19.61
42	Smup	68	75	9783	10713	6.950	7.008	3253	5060	1807
43	Pbct	27	38	5431	6615	4.971	5.744	66.07	71.42	5.35
44	Gecco	35	47	7677	7137	4.559	6.585	50	50	0
45	JExj	49	62	6321	6793	7.751	9.127	78.12	83.33	5.21

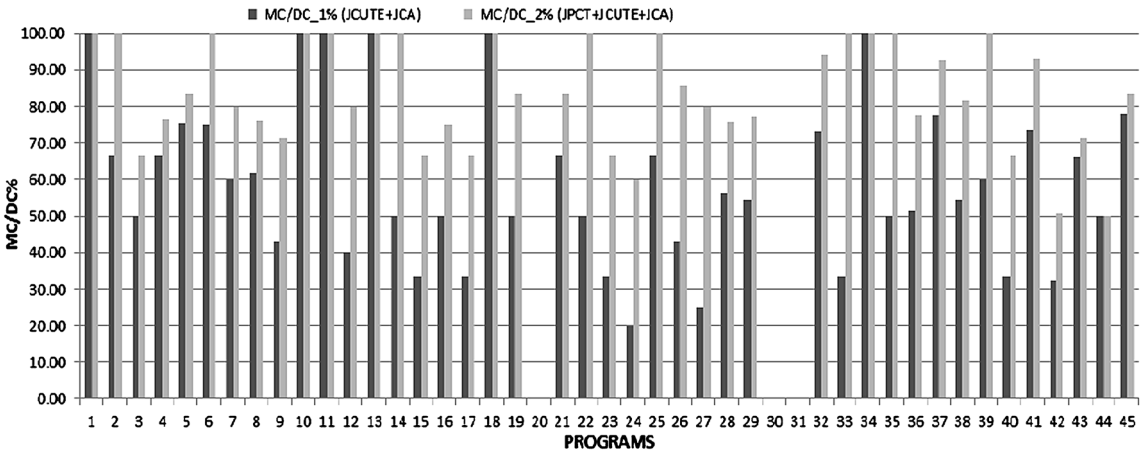
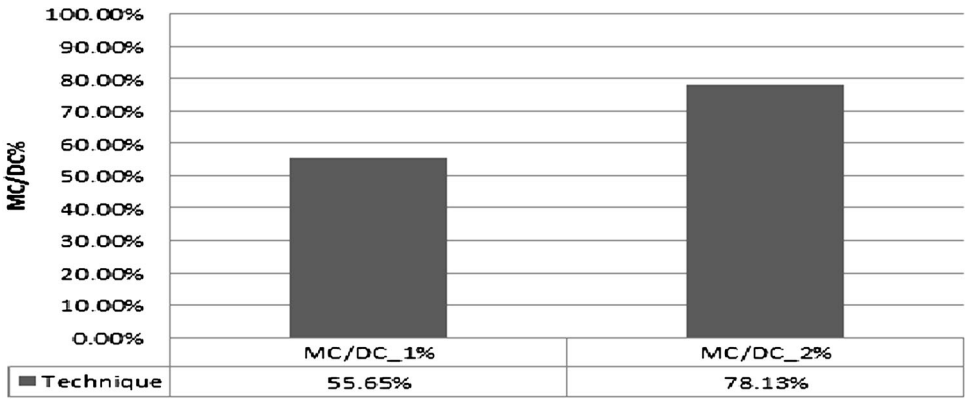


Fig. 6 Comparison of MC/DC for different programs

Fig. 7 Difference between the two MC/DC percentages



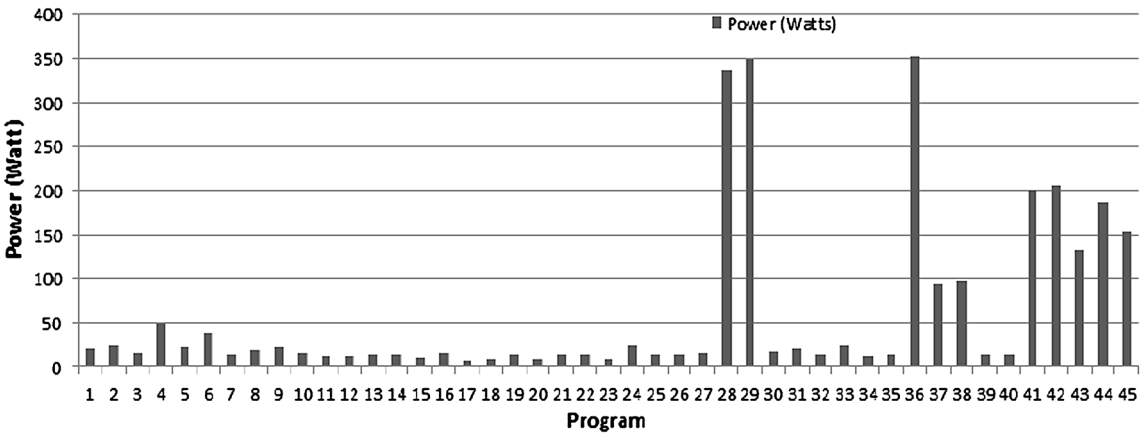


Fig. 8 Recorded power consumption over time of interest

It is clearly reflected from Fig. 7 that we get significant increase in MC/DC.

Table 6 discuses total energy consumption analysis of both the scenarios for forty-five programs as discussed in Algorithm 1. Columns 3 to 5 present timestamps recorded from, to, and total time in milliseconds respectively. Column 6 presents the consumption of power of an application in *watt* over the time taken. Figure 8 shows the correlation between time and energy consumption. Here, X-axis shows the input programs and Y-axis shows the total Power consumed by program. This is the graph of consumption of power vs time taken. We may observe three programs consume power more than 100 watts as reported in Column 6. We present energy consumption for all forty-five

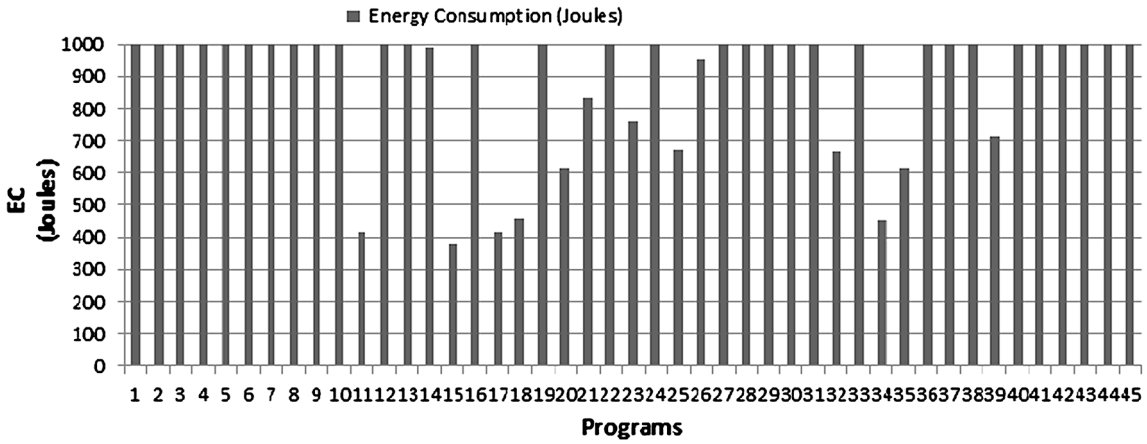


Fig. 9 Comparison of computed energy consumption for different programs

programs in Column 7. The range of energy consumption is from 379.8578 to 162100.0568 J. Figure 9 shows the comparison of consumption of energy for all forty-five programs. We have scaled down the value of energy consumption to show the comparison of all Java programs. In this graph we have taken 1000 J as threshold value. In Fig. 9, the programs are taken along X-axis. Y-axis represents the energy consumption in Joules. Here, we conclude that the total energy consumption for forty-five programs is 747.51 kJ.

5 Threats to validity

Here, we present some important threats to the validity of Green-J^3 Model.

1. The programs taken for experimental studies are amenable to concolic testing, this is our first threat to the proposed work.
2. Our second threat to validity is related to some shortcomings of symbolic execution engine that is used in JCUTE. Some possible shortcomings of concolic tester are given below:
   - It may not scale up if the domain of input is large.
   - If the program exhibits non-deterministic behavior, it may follow a difference path than the intended one. This can lead to non-deterministic of the search and poor coverage.
   - Even in a deterministic program, a number of factors may lead to poor coverage, including imprecise symbolic representations, incomplete theorem proving, and failure to search the most fruitful portion of a large or infinite path tree.
3. The third threat to validity is that the system configuration JCUTE and JolueMeter are developed for windows 7 operating system only.
4. The proposed program transformation using Quine McCluskey technique works fine for small and moderate sized programs. But, for very large programs with many-many alternative paths, the time complexity and space complexity will significantly increase. When we compare QM technique with K-Map, then QM results in better output because K-Map is able to handle only upto four or five variables in a decision and QM is capable of handling *n* number of variables present in a decision. But, for very large programs, we have to bear this overhead, w.r.t. space and time.

6 Comparison with related work

In this section we discuss some existing related work to compare our proposed work.

Li et al. [7] proposed an approach for Energy-Directed Test Suite Optimization (EDTSO). EDTSO is a new test suite minimization approach that allows software testers to generate energy-efficient, and minimized test suites. Their proposed technique is based on encoding minimizing problems as integer linear programming problems. In our proposed work, we have achieved higher code coverage and computed energy consumption for the whole process.

Amsel and Tomlinson [8] have proposed a tool called Green Tracker. Green Tracker estimates the energy consumption of software in order to help the concerned users in taking suitable decisions about the software they use. Amsel and Tomlinson [8] aimed at creating awareness about the potential environmental hazards associated with

Table 6 Power consumption and energy consumption of different programs

Sl. no.	Program name	Timestamps			Application	Energy
		From	To	Total time (ms)	Power (W)	Consumption (J)
1	Condition	63569135723137	63569135838565	115428	21.4	2470.1592
2	Weight	63569172791592	63569172893971	102379	24.4	2498.0476
3	QuickSort	63569187653423	63569187910201	256778	15.6	4005.7368
4	Nonce	63569187589724	63569187693480	103756	48.8	5063.2928
5	StringBuffer	63569188734218	63569188829946	95728	22.4	2144.3072
6	SwitchTest	63569188811419	63569188964976	153557	38.7	5942.6559
7	ProducerConsumer	63569189642109	63569189730062	87953	13.2	1160.9796
8	BSTree	63569189841239	63569189915821	74582	18.9	1409.5998
9	ArraySort	63569190842256	63569190907903	65647	22.5	1477.0575
10	StaticInstanceProblem	63569190981435	63569191064177	82742	15.6	1290.7752
11	CAssume	63569191523451	63569191560327	36876	11.3	416.6988
12	CTest3	63569191662352	63569191747875	85523	12.6	1077.5898
13	Deadlock	63569191813578	63569191886269	72691	14.4	1046.7504
14	Demo	63569191921478	63569191996091	74613	13.3	992.3529
15	DemoLock	63569210099898	63569210138659	38761	9.8	379.8578
16	CTest2	63569210257963	63569210328445	70482	15.2	1071.3264
17	CTest1	63569210425916	63569210484492	58576	7.1	415.8896
18	DemoL0ck2	63569210495431	63569210551276	55845	8.2	457.9290
19	DSort	63569210613798	63569210689054	75256	14.3	1076.1608
20	ErrorTest	63569210724516	63569210798836	74320	8.3	616.8560
21	If-Else	63569210813556	63569210875797	62241	13.4	834.0294
22	InterfaceJava	63569210914213	63569211002665	87852	14.5	1273.854
23	LockHeld	63569211035467	63569211116969	81502	9.3	757.9686
24	MultiLock	63569211249862	63569211295400	45538	23.8	1083.8044
25	NewJava	63569211332584	63569211379961	47377	14.2	672.7534
26	NoPredictive	63569211546891	63569211612024	65133	14.6	950.9418
27	NoPredictive2	63569211731542	63569211806923	75381	15.4	1160.8674
28	NS	63569211917241	63569212309578	392337	336.8	132139.1016
29	NS2	63569212405213	63569212869417	464204	349.2	162100.0368
30	FinallyTest	63569212971345	63569213036686	65341	18.2	1189.2062
31	OneWrite	63569213125721	63569213211048	85327	21.3	1817.4651
32	Regression	63569213245798	63569213291627	45829	14.5	664.5205
33	SampleXACMLPolicy	63569213310728	63569213420967	110239	24.8	2733.9272
34	StackTest	63569213521668	63569213558640	36972	12.3	454.7556
35	StaticInstanceProblem2	63569213613792	63569213656897	43105	14.2	612.0910
36	StringBufferModule	63569213799811	63569214190883	391072	352.3	137774.6656
37	StringBuffer1	63569214281997	63569214438211	156214	94.5	14762.223
38	StringBuffer2	63569214534425	63569214656207	121782	98.2	11958.9924
39	Struct	63569214735421	63569214789443	54022	13.2	713.0904
40	Testme	63569214898526	63569214972368	73842	14.5	1070.709
41	Copeca	63569214981235	63569215006733	25498	200.67	5626.6436
42	Smup	63569215013241	63569215033837	20596	205.32	4228.770
43	Pbct	63569215033989	63569215046135	12146	133.36	1619.79
44	Gecco	63569215051013	63569215065927	14914	186.35	2779.22
45	JExj	63569215071032	63569215084246	13214	153.32	2025.97

software and improving software engineering techniques to reduce the energy consumption of software. Like Amsel and Tomlinson [8], we also intend to spread the awareness on Green Software Testing. In our work, we have discussed the energy consumption analysis of a testing tool.

Dick et al. [9] presented a method to compute and rate software-induced energy consumption of stand-alone software applications on desktop computers as well as interactive transaction-based software applications on servers. They have intended to support software developers, administrators, purchasers, and users in making informed decisions on software architecture and implementation as well as on software products they use or plan to use. In our proposed approach, we measure the performance of a software testing tool by computing some metrics. Like, Dick et al. [9], we also intend to advice software testers to choose energy efficient software testing tools. We advice the testers to choose a software which performs the same objectives as others, but with less power, and less energy.

Chen et al. [10] proposed and developed a tool called StressCloud to measure the performance and energy consumption for cloud based applications. In our proposed work, we also measure the energy consumption, but for a software testing tool J^3-Model through Green J^3-Model.

Capra et al. [11] have developed the hardware kit that used ammeter clamps and a workload simulator tool. Ammeter clamps were used to measure the energy absorbed by the server machine. Workload simulator tool generated benchmark workloads for different categories of applications. Capra et al. [11] measured the energy efficiency of software applications. In our proposed work, we have developed Green-J^3Model, that takes a software testing tool and measures the energy consumption, as Capra et al. [11] has measured.

Table 7 Comparison of different works on concolic and coverage based testing

S. no	Authors	Framework	Description
1	Li et al. [7]	EDTSO	Based on encoding minimization problem as integer linear programming problem
2	Amsel and Tomlinson [8]	GreenTracker	Estimates the energy consumption of software in order to help concerned uses make informed decision about the software they use
3	Dick et al. [9]	PM,WG,DAE	How to measure the energy consumption of software
4	Chen et al. [10]	StressCloud	Analyzing the performance and energy consumption of a cloud application
5	Capra et al. [11]	WLS, AC, VIM	Proposed method for software energy efficiency for application software
6	Brown and Reams [12]	–	Article suggests an overview of all approaches to energy efficiency in computing system
7	Saxe [13]	PowerTOP	This work shows the extent of the waste, while also showing which software is responsible
8	Das et al. [14]	BCT,CREST, CA	Based on concolic testing and MC/DC testing. BCT uses K-Map as minimization technique
9	Bokil et al. [19]	AutoGen	Analysis of all all coverage criteria with time effort
10	Godboley et al. [15, 17]	PCT,CREST,CA	Based on concolic testing and MC/DC testing. PCT uses QM as minimization method
11	Burnim et al. [20] BC	CREST	Worked on heuristic concolic testing and branch coverage
12	Godboley et al. [15, 18]	XNCT,CREST,CA	Approach deals with concolic testing and MC/DC testing. EX-NCT uses Exclusive-NOR operation
13	Kim et al. [22]	CREST	Based on concolic testing
14	Godboley et al. [15, 16]	APCT,CREST,CA	Approach based on concolic testing and MC/DC testing. APCT uses Modified–QuineMcMluskey method
15	Kim et al. [25]	CONBOL	Based on concolic testing. Analysis for branch coverage and time taken
16	Majumdar et al. [23]	CUTE	Worked on HCT and branch coverage
17	Kim et al. [24]	SCORE	Approach is build on distributed concolic testing
18	Sen et al. [5]	CUTE,JCUTE	Concolic tool developed for C and Java programs
19	Kim et al. [21]	SMT Solver,CREST	Based on HCT and analysis on reduction ratio
20	Hoing et al. [26]	SEEP	Spread the awareness on energy consumption on programming using symbolic execution
21	Godboley et. al.[27]	Green-JEXJ	Spreading awareness on energy consumption using JEXJ framework
22	Proposed Work	Green-J^3 Model	Spread the awareness on energy consumption analysis on Software testing techniques

Brown and Reams [12] discussed some approaches to achieve energy efficiency in computing systems. This is nothing but a literature survey on power consumption and energy consumption. In our work, we presented a detailed experimental study on energy consumption on a software testing tool.

Saxe [13] proposed and developed a tool called PowerTOP. Their work shows the explanation of e-waste. Also, they suggested which application was responsible for this e-waste. In our proposed approach, we have computed energy consumption of a testing tool. When our tool can compute, the energy consumption of different testing tools, then our tool can suggest the most energy efficient testing tool.

Das et al. [14] and Godboley et al. [15–18] have proposed several code transformation techniques which supported to achieve higher code coverage. They have considered CREST tool as concolic tester which generated the test cases. Also, they have developed a coverage analyzer which accepts a program along with test cases as input and produces to measure MC/DC% as output. In our work, we also proposed the same core idea but using different technique i.e. Java Program Code Transformer (JPCT). In addition, we spread the awareness on green analysis.

Bokil et al. [19] have proposed and developed an AutoGen tool. This tool produced test cases and computed code coverages. When it compares the execution time of automated testing tools as compared to manual testing strategy, then they found that automated testing tools save one third time. In our work, we also computed the execution time of the process.

Burnim et al. [20], Kim et al. [21, 22], Majumdar et al. [23], Kim et al. [24], Sen et al. [5], and Kim et al. [25] have proposed and developed several concolic testers. Some of them have been implemented in C language and some of them have used Java language to implement the tool. Some of the works were implemented in distributed environment. In our proposed work, we have used jCUTE as the concolic

Table 8 Characteristics of different approaches on concolic and coverage based testing

S. no	Authors	Generated test cases	Measuring coverage%	Determined time constraints	Measured speed	Computed power consumption	Computed energy consumption
1	Li et al. [7]	✓	✓	✓	X	X	✓
2	Amsel and Tomlinson [8]	X	X	X	X	✓	✓
3	Dick et al. [9]	X	X	X	X	✓	✓
4	Chen et al. [10]	X	X	X	X	X	✓
5	Capra et al. [11]	X	X	✓	X	✓	✓
6	Brown and Reams [12]	X	X	X	X	✓	✓
7	Saxe [13]	X	X	X	X	✓	✓
8	Das et al. [14]	✓	✓	X	X	X	X
9	Bokil et al. [19]	✓	X	✓	X	X	X
10	Godboley et al. [15, 17]	✓	✓	X	X	X	X
11	Burnim et al. [20]	✓	X	X	X	X	X
12	Godboley et al. [15, 18]	✓	✓	X	X	X	X
13	Kim et al. [22]	✓	X	X	X	X	X
14	Godboley et al. [15, 16]	✓	✓	✓	X	X	X
15	Kim et al. [25]	✓	✓	✓	X	X	X
16	Majumdar et al. [23]	✓	X	X	X	X	X
17	Kim et al. [24]	✓	✓	X	✓	X	X
18	Sen et al. [5]	✓	✓	✓	X	X	X
19	Kim et al. [21]	✓	X	X	X	X	X
20	Hoing et al. [26]	✓	X	X	X	✓	✓
21	Godboley et al. [27]	✓	✓	✓	✓	✓	✓
22	Proposed Work	✓	✓	✓	✓	✓	✓

tester to produce test cases. It may be noted that jCUTE is compatible with Java.

Hoing et al. [26] and Godboley et al. [27] have proposed and developed some tools that spread the awareness on energy consumption analysis of software applications. Our proposed work also targets the same objective to spread the awareness regarding the energy consumption.

Table 7 summarizes the comparison of some related work. We present the frameworks developed and used by various authors in the third column of Table 7. Brief description of various mentioned research work is provided in the fourth column of Table 7. We can observe from Table 7 that authors mentioned in sl. no. 1 to 7 proposed their research work based on power consumption and energy consumption. These work help to spread awareness for GREEN IT and GREEN Software Engineering. Authors listed in sl. no.number 8 to 19 explain about concolic testing and coverage based testing. Last row of Table 7 shows our proposed work. Here, we present Green-J^3 Model, which is based on Concolic Testing, Branch Coverage, and Green Software Engineering. Green-J^3 Model helps to enhance awareness about the importance of energy consumption in software testing.

Table 8 presents the comparison of different characteristics of existing approaches. These characteristics are Test Cases, Coverage%, Time Constraints, Speed, Power Consumption, and Energy Consumption. Among all the existing works, only Li et al. [7] have done the analysis of energy consumption for software testing. Please note that authors listed sl. no. 2–7 proposed approaches only for energy consumption and power consumption. They have not focused on software testing techniques. Again please note that the authors listed sl. no. 8–19 have only focused on software testing since they are unaware of Green IT and Green Software Engineering. Last row in Table 8 shows our proposed work. We have done our research on all the characteristic mentioned in Table 8. Our proposed work deals with software testing as well as Green IT, and Green Software Engineering.

7 Conclusion and future work

We proposed a tool named Green-J^3 Model to measure the energy consumption of modified condition/decision coverage using concolic testing. We discussed Green-J^3 Model along with the model overview, the block diagram, and algorithmic description in detail. The experimental results show that the proposed approach of test case generation achieved better MC/DC in comparison to the existing methods. Green-J^3 Model achieved 21.32% of average enhancement in MC/DC for forty-five programs. The total energy consumption of the whole experimental process is 747.51 kJ.

In the future, we will rectify some of the significant identified threats to validity of our work. We will develop other code transformers to experiment with Java program to achieve high MC/DC as compared to existing approaches. It is very important to compute the energy consumption of each modules of a software testing tool, therefor in our future work we will extend our work by computing individual energy consumptions. We will try to work on comparison of energy consumption of difference software testing techniques in concolic and MC/DC testing.

References

1. Kern E, Dick M, Naumann S, Guldner A, Johann T (2013) Green software and green software engineering–definitions, measurements, and quality aspects. In Proceedings of ICT4S, pp 87–94 ETH Zurich, February 14–16 2013
2. Naumann S, Dick M, Kern E, Johann T (2011) The GREENSOFT model: a reference model for green and sustainable software and its engineering. Sustain Comput Inform Syst 1(4):294–304
3. RTCA, Inc. (1992 December) RTCA/DO-178B, Software considerations in airborne systems and equipment certification, Washington, DC
4. Hayhurst KJ, Veerhusen DS, Chilenski JJ, Rierson LK (2001) A practical tutorial on modified condition/decision coverage, NASA/TM-2001-210876, May 2001
5. Sen K, Agha G (2006) CUTE and jCUTE: concolic unit testing and explicit path model-checking tools (tools paper). DTIC Document
6. JouleMeter: computational energy measurement and optimization. http://research.microsoft.com/en-us/projects/joulemeter/
7. Li D, Sahin C, Clause J, Halfond WGJ (2013) Energy-directed test suite optimization. In: 2nd international workshop on green and sustainable software (GREENS), pp 62–69, New York, USA, May 2013
8. Amsel N, Tomlinson B (2010) Green tracker: a tool for estimating the energy consumption of software. In: Proceedings of the 28th international conference on human factors in computing systems, CHI, pp 3337–3342, Atlanta, Georgia, April 10–15 2010
9. Dick M, Kern E, Drangmeister J, Naumann S, Johann T (2011) Measurement and rating of software induced energy consumption of desktop PCs and servers. EnviroInfo 2011: innovations in sharing environmental observations and information, Shaker Verlag Aachen
10. Chen F, Grundy J, Schneider JG, Yang Y, He Q (2014) Automated analysis of performance and energy consumption for cloud applications. In: Proceedings of the 5th ACM/SPEC international conference on performance engineering, ACM, pp 39–50, Dublin, Ireland
11. Capra E, Francalanci C, Slaughter SA (2012) Measuring application software energy efficiency. IT Prof 14(2):54–61
12. Brown DJ, Reams C (2010) Toward energy-efficient computing. Commun ACM 53(3):50–58
13. Saxe E (2010) Power-efficient software. Commun ACM 53(2):44–48
14. Das A, Mall R (2013) Automatic generation of MC/DC test data. Int J Softw Eng Acta Press 2(1):1–8
15. Godboley S (2013) Improved modified condition/ decision coverage using code transformation techniques. Thesis (MTech) NIT Rourkela

16. Godboley S, Mohapatra DP (2013) Time analysis of evaluating coverage percentage for C program using advanced program code transformer. In: 7th CSI international conference on software engineering, pp 91–97, Nov 2013
17. Godboley S, Prashanth GS, Mohapatra DP, Majhi B (2013) Increase in modified condition/decision coverage using program code transformer. In: IEEE 3rd international advance computing conference (IACC), pp 1400–1407, Feb 2013
18. Godboley S, Prashanth GS, Mohapatra DP, Majhi B (2013) Enhanced modified condition/decision coverage using exclusive-nor code transformer. In: 2013 international multi-conference on automation, computing, communication, control and compressed sensing (iMac4s), pp 524–531, March 2013
19. Bokil P, Darke P, Shrotri U, Venkatesh R (2009) Automatic test data generation for C programs. In: 3rd IEEE international conference on secure software integration and reliability improvement
20. Burnim J, Sen K (2008) Heuristics for scalable dynamic test generation. In: Proceedings of ASE, Washington, DC, USA, pp 443–446
21. Kim M, Kim Y, Choi Y (2012) Concolic testing of the multi-sector read operation for flash storage platform software. Form Asp Comput 24(3):355–374
22. Kim M, Kim Y, Jang Y (2012) Industrial application of concolic testing on embedded software: case studies. In: 2012 IEEE fifth international conference on software testing, verification and validation (ICST), pp 390–399
23. Majumder R, Sen K (2007) Hybrid concolic testing. In: Proceedings of the 29th international conference on software engineering, IEEE Computer Society, Washington, DC, USA, pp 416–426
24. Kim M, Kim Y, Rothermel G (2012) A scalable distributed concolic testing approach: an empirical evaluation. In: 2012 IEEE fifth international conference on software testing, verification and validation (ICST), pp 340–349, April 2012
25. Kim Y, Kim Y, Kim T, Lee G, Jang Y, Kim M (2013) Automated unit testing of large industrial embedded software using concolic testing. In: IEEE/ACM 28th international conference on automated software engineering (ASE), pp 519–528, Nov 2013
26. Hönig T, Eibel C, Kapitza R, Schröder-Preikschat W (2012) SEEP: exploiting symbolic execution for energy-aware programming. SIGOPS Oper Syst Rev 45(3):58–62
27. Godboley S, Dutta A, Besra B, Mohapatra DP (2015) Green-JEXJ: a new tool to measure energy consumption of improved concolic testing. In: 2015 international conference on green computing and internet of things (ICGCIoT), pp 36–41, Oct 2015