Infeasible Paths Detection Using Static Analysis

Burhan Barhoush
Faculty of Sciences & IT
JUST University
bbarhoush@yahoo.com

Izzat Alsmadi
CIS Department
IT & CS Faculty
Yarmouk University
ialsmadi@yu.edu.jo

ABSTRACT

Software testing is the process that aims to detect the errors in the software product by using test cases, and to discover the components of the software that are responsible of these errors. The testers need to ensure that every component of the software is tested correctly in order to achieve high coverage in terms of testing one or more of the software aspects such as: code, user interface, etc. Many coverage aspects were proposed in testing research field such as node, edge, edge-pair coverage, prime path coverage, etc. By reading many of these studies, we can notice that they propose many solutions for detecting and discovering infeasible paths.

An infeasible path is simply any path that cannot be traversed by test cases. Some of the causes of the infeasible paths are dead codes, correlated predicates with respect to a certain variable (which is one of the main reasons for infeasibility in the software programs) or according to the test cases itself.

In this paper, a tool is developed to automatically detect the infeasible paths that may exist in source code and that are caused by the logically inconsistent predicates related to dead codes, and by the correlated conditional statements with respect to a certain variable.

Our application tool is evaluated against four source codes, and the experimental results showed that the tool can effectively detect infeasible paths except in the source codes that contain while loop structures.

Keywords

Software coverage, software testing, static code analysis, infeasible paths.

1. INTRODUCTION

Testing coverage is an important quality aspect that enables developers or testers to evaluate software testing activities. In coverage assessment, one aspect of software such as: code, requirements, code paths, statements, branches, etc. is evaluated and then after generating and executing the set of proposed test cases, coverage for the test cases is calculated based on the percentage of the paths that was tested or visited by those test cases.

Ultimately, the goal is to reach 100 % coverage, which is impractical in many cases. For the least of our goal is to increase this value for a point close to 100 %

For code assessment, several coverage aspects can be assessed. Path coverage is one of those aspects whereas graph (e.g. call, decision, and control flow graph) is generated for the source code under test. Test cases are then generated and evaluated. We want to see is the percentage of paths that were tested or visited by all those test cases.

It was noticed that in some cases, especially for large software products, some paths, may never or hardly be tested or visited by any test case. Detecting such paths is very important task in which discovering and detecting such paths helps in saving time and resources, rather than trying to test them.

There are many reasons for such cases; one of the main causes of infeasible paths is the presence of dead code. Dead code is the statement(s) that can never be executed. It causes infeasible paths because the statements cannot be reached. An example of dead code is the predicates that composed of conflicting clauses, such as (y = = 40 || x > 20 && x <= 10). Another cause of infeasible paths is the existence of correlated conditional statements with respect to a certain variable (which is one of the main reasons for infeasibility in the software programs [1], [2]). Herein, the variable must be definition clear (def-clear) between the correlated conditional statements to cause the infeasible paths. Actually, "only about 13% of correlated conditional statements can be timely detected during compilation" [2].

Figure 1 shows a simple example of how can some test paths be infeasible. The simple code can have four total test paths: TP1: 1,2,4,6,8,10, TP2: 1,2,4,7,9,10, TP3: 1,3,5,6,8,10, TP4: 1,3,5,7,9,10. According to the two conditional statements: (if(x<0) and if(x>2) and due to the contradicted logic of the two conditions, we can easily notice that TP4 and TP1 can never be executed and hence will be always infeasible paths.

Infeasible paths can be also subjective and depend on the actual program usage. For example, if we know that this is a program to take students grades (which can never be zero), TP1 and TP2 can be infeasible paths.

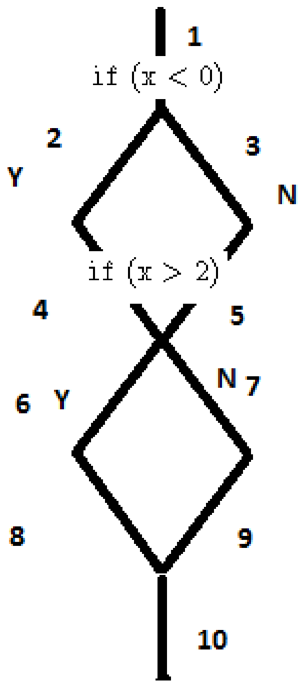


Figure 1: A conditional infeasible path example

Static analysis tools analyze the source code statically based on control flow graph and data flow analysis of the source code, i.e. without real execution process of the program under test (PUT). In addition to typical errors that testing tools can reveal, static analysis tools may indicate warnings to show areas in the code that need to be improved for code quality despite they may not cause direct compilation errors. Static infeasible path detection approaches also depends on the control flow and data dependencies between the components of PUT.

As part of the tool that is developed in this paper, a GraphXML file is created. This file is an XML file that can be used in order to describe the control flow and data flow dependencies between the components of the software program. Then it will be used with the corresponding source code as an input for the developed application tool in order to detect infeasible paths.

The rest of the paper is organized as the following: Section 2 presents the related works, section 3 depicts the methodology, in section 4 the experimental results are shown, section 5 is the conclusion and future works.

2. RELATED WORK

Symbolic execution uses techniques based on symbols and mathematical equations in order to symbolically execute programs without real execution. The search based testing is either random or dynamic. The random search based testing searches randomly for test cases. But the dynamic type applies test cases dynamically (at run time) to find the optimum test cases.

There are many papers that were conducted to address issues in this area as we will see below in this section.

2.1 Infeasible Paths Detection

This section lists some of the studies that were conducted in order to detect the infeasible paths in the software programs. Some of these studies use static analysis techniques, and others use dynamic techniques.

Delahaye, et al. proposed an algorithm in [3] for detecting an infeasible path, which can be used to generalize a new family of infeasible paths. They claimed that the experimental results of their proposed method (using a consistency check) can save a considerable time over the methods that do not use the generalization algorithm. The idea is based on determining the path condition (constraint system) as a conjunction of individual constrains on the input symbolic variables, and detecting the minimal set of constraints that are inconsistent along the path condition, where such constraints can indicate that path as an infeasible path. These inconsistent constraints are named as explanations. Then the explanation can be used to generalize the role of detection for infeasible paths.

Ngo, et al. Tan identified in their paper [4] four patterns that will cause four types of infeasible paths which have common properties. Also in this paper, they proposed and implemented four algorithms for detecting the infeasible paths (one algorithm for each type of the patterns). The experiment results that are conducted on seven systems showed that the proposed algorithms succeeded in detecting 82.3% of all infeasible paths in these systems [4].

Authors' paper presents an empirical approach to the problem of infeasible path detection. This approach is based on the fact that many infeasible paths exhibit some common properties which are caused by four main code patterns: identical/complement-decision, mutually-exclusive-decision, check-then- do and looping-by-flag pattern. Through realizing these properties from source code, many infeasible paths can be precisely detected. Binomial tests have been conducted which give strong statistical evidences to support the validity of the empirical properties.

The novelty of the approach that is followed in the paper lies in the use of empirical properties that have been statistically validated.

In paper [5] Suhendra, et al. suggested an approach for exploiting the infeasible paths in software timing analysis. Their method based on detecting the conflict between a pair of branch nodes, (ex. $(x<2)$ and $(x>4)$) or between a branch node and an assignment node (ex. $(x=2)$ and $(x>4)$).

The approach firstly started by detecting the conflict branches (BB_Conflict), and the conflict between assignment and branch nodes (AB_Conflict). This step can be done by examining the conflicts between the outcomes of two conditional nodes for BB_Conflict and also examining the conflicts between assignment node and the outcomes of a conditional node. Then, by starting from the sink node towards the source node, a path is created that contains an empty set for BB_Conflict and AB_Conflict until reaching the conflicting nodes. According to the type of the conflict (BB_Conflict or AB_Conflict), fill the appropriate conflict set with the partial path from conflicting node to the sink node. The complete path containing that partial path will be considered as infeasible path. Thus, avoiding this path from execution paths is the soul of WCET estimation.

This approach tries to guarantee avoiding enumeration of a large number of execution paths. But the drawbacks of this approach is that it detects only the pairwise conflicts and fails in the case of arbitrary infeasible paths, and it doesn't account for infeasibility in procedure calls, and cannot capture infeasible paths across loops [5].

In the paper [6] the authors proposed a method to track reversely the execution paths in the program under test in order to detect the infeasible paths. Their method was motivated by the fact that new malwares hides malicious code in the infeasible paths to be not detected. The method first scans the source code and slices to the blocks of the conditional nodes, then it detects the end points of the program, and then it connects the blocks together to obtain the CFG. After that, it reversely traverses the paths starting from the end points, and keeps track of the information about the execution paths. Last, it generates test cases and performs analysis on the execution paths. By using this method, the testers can traverse each path in the program including the feasible and infeasible paths, although it causes processing overhead, which can be reduced by using computers with high processing capability.

Paper [7] performed an extension to Boogie tool [13] for detecting the infeasible code. The authors of this paper claimed that their tool, Joogie, is fully automatic. It first translates the code into Boogie (an intermediate representation of the program's bytecode). Then it calls the Boogie program verifier that uses the weakest liberal precondition calculus to check the existence of infeasible code.

For the control statements, the tool uses an auxiliary variable that checks weather the execution goes through the control location or not, the initial value of the auxiliary variable is set to zero, and if the execution goes through the control location, its value become 1. This check is automated by computing formulae to represent the weakest liberal precondition. To compute this formula, any loop must be replaced by three unwindings; the first and last one represent the first and last iteration of the loop, and the middle unwinding is replaced by non-deterministically assigning values to the variables that modified inside the loop.

For the recursive method calls, Joogie also replaces it by non-deterministically assigning values to the variables that were modified through method calls.

2.2 Test Case Generation and Coverage Issues Related to Infeasible Paths

This section surveys some of the papers that proposed techniques to generate automatically test cases for testing the software programs. These techniques detect automatically, through a program, the infeasible paths during the test case generation in order to avoid them and traverse through the feasible paths.

In paper [8] the authors suggested an approach that reduces the number of test data needed to examine the coverage of a source code by constructing what is called as critical branches CBs [8], that if they were covered by test cases, the remaining branches of the program were definitely covered. The idea in this paper is based on constructing a control flow graph of the program under test, and then constructing an edge pre and post dominator trees from that CFG. Then, merging them to obtain the edge dominator graph1, after that, detecting the strongly connected components in the edge dominator graph to polish the graph and obtain the edge partition dominator graph. In the strongly connected component (edge partition), each node dominates each other nodes in the same edge partition. In the edge dominator graph, if any node in the edge partition is covered, the other nodes in the same edge partition are also covered.

In the resulting edge dominator graph, if any leaf node is covered by a test case, the ancestor nodes of that leaf are also covered. Thus, we need just to generate test cases for examining the coverage of these leaves nodes. The leaves nodes are called the critical branches. In order to automatically generate test cases, the Genetic Algorithm is used in [8]. Two metrics were used to prioritize the CBs; the first metric gives a high priority is assigned for the CBs whose source nodes are critical nodes, and the second one gives the CBs with higher weights a higher priority. The weight of the CB is given by computing the total number of branches in the its edge partition and all its ancestors edge partitions in the EPDG [8]. A coverage table is used to keep track of the information about CBs execution. This paper used the information about the covered CBs and its ancestor's branches $B\_\mathrm{cov}er$, and the number of all branches in the original CFG B_all to compute the coverage rate as in the equation 2.2.1 [8]:

$$BCR = \frac{B\_\mathrm{cov}er}{B\_all} \times 100\%$$

In paper [9] the authors used a multi objective evolutionary search approach with an executable behavioral model EFSM (Extended Finite State Machine) to generate feasible test paths. The goal of using the EFSM model is to obtain feasible paths dynamically. The multi objective search approach includes achieving test purpose coverage with a minimal sequence size.

The method used in this paper utilizes a model analyzer to obtain slicing information for the parts of the model that affect the transition of interest. The test purpose, which is described as a set of conditions, defines the parts of the system that need to be tested. Thus, by obtaining the slicing model, we can identify the critical transitions, which are the transitions that deviated from the transitions of the slice model.

After that, and once the executable model were produced, and used to trigger the transitions that satisfy the associated guards, the test sequence generator comes into play to evaluate the solution. The test sequence generator uses a population that composed of three parts; test sequence size (used as a design variable that allows to generate sequences of different numbers of input events), sequence of input events, and parameters of all input events.

In the first step in the test sequence generator, the population is created randomly. Then, each design variable is mutated one at a time to generate a mutated population, and then the mutated population will be evaluated by the objective functions, and each design variable returns its original value.

After that, one of the objective functions is chosen randomly and each design variable will gave a fitness value by the fitness function.

After that, the design variables are ranked according to their fitness value; the first position of the ranking means that the design variable is least adaptive.

Raquel in [10] proposed an algorithm for automatic test data generation using scatter search approach (TCSS) that utilizes the diversity property in order to generate test cases that can satisfy branch coverage, thus, reaching all the possible nodes throughout the control flow graph of the PUT. Raquel Blanco also proposed an extension to their approach (TCSS-LS) with the help of local search approach in the backtracking process in addition to the scatter search to cover the difficult branches.

As mentioned above, TCSS aims to achieve cover all of the branches along the program under test, it divide the goal into sub goals, and each sub goal preform search for test cases that reach to the end node, or to a certain branch node

The process starts by creating the reference set, which is initially be empty, at the root node and each branch node. Each node uses its own reference set to store particular information during test case generation; this information helps in detecting what are the branches that are reached. The reference set composed of four components; the test case that reaches the node (which is called solution), the path that is covered by the solution, the distance to the sibling node (the sibling node is the node whose input decision is the negation of the current node [10]) which indicates “how close the solution came to cover the sibling node”, and the distance to the child node which indicates “how close the solution came to cover the child node”.

In the first iteration of the proposed approach, the diversity generation method generates random solutions and stores them in root node reference set. In the later iterations, the generator selects the node and uses its reference set to combine the best solutions (in terms of diversity criteria which is a measure of the number paths covered by all solutions of the set) in order to produce new solutions. If the reference set of a particular node doesn’t contain at least two solutions to be combined, the process of backtracking takes place.

There two options of backtracking process; the first option is to use and combine the solutions of the parent node that were not used to be combined in the previous iteration. But if all of the solutions of the parent node had been combined, the backtracking process depends on mutating some solutions of the parent node to generate new solutions to be combined.

The second version of TCSS (TCSS-LS) uses the local search method in addition to scatter search to allow intensification of the search on test case that can cover the difficult branches.

The authors of paper [11] proposed a framework for dynamically generating test data that based on genetic algorithms. The proposed technique focuses on unit testing and supports primitive data types, arrays, loops, if statements.

The first component of the framework is Basic program analyzer system (BPAS) [11] which is composed of five layers; IOExecutive, the parser, the Walker, the Static Analyzer, and the Dynamic Analyzer. The IOExecutive layer is responsible for providing information about the methods, and the variables of the program. The parser layer is responsible for parsing the source code. The Walker is responsible for building the control flow graph. The Static Analyzer analyzes the program without executing the program. The last layer is responsible for analyzing the behavior of the program through real execution.

The second component of the proposed framework is the Automatic Test Case Generation System (ATCGS) [11]. It uses two algorithms; the Batch-Optimistic, and the Close-Up.

The first algorithm uses Genetic Algorithm that builds a chromosome from a set of test cases that describes the names and types of the variables, the input values, the values of the input variables at a specific state, and the covered nodes of the control flow graph with respect to the input variables.

The crossover process in this algorithm is of two types; intra and inter crossover; the intra crossover selects the crossover point within the same gene and performs swapping of the internal parts of the same gene. The inter crossover selects the crossover point between two chromosomes to do the swapping.

The resoled offspring chromosomes are then added to the list of chromosomes for the next generation by the mutation process. The mutation process is classified into two types; intra-mutation and inter-mutation; the intra-mutation is used to mutate the entire gene, and the inter-mutation is used to mutate specific parts of the gene.

This algorithm ends when the full coverage criterion of the edges/condition is achieved, or when a maximum number of generations are reached.

When the Batch-Optimistic terminates without full coverage of the program, the Close Up comes into play to search for test cases for the edges/conditions that are not covered by Batch-Optimistic algorithm; first, the Close Up algorithm determines the targets that are not covered and finds the possible paths to that targets. It utilizes the GA to find test cases that can cover the targets and if such a test case is found, the Close Up terminates and tries to find a test case for the next target; otherwise, it terminates after a predefined number of generations and the target is then classified as unreachable.

3. METHODOLGY

This section describes the research methodology that is followed to complete the study. Figure 2 displays the steps that are followed in infeasible paths detection.

The first step is generating GraphXML file for describing the Control Flow Graph (CFG) and Data Flow Graph (DFG) of the program under test (PUT). The second step is analyzing the CFG and DFG in order to detect the logically inconsistent predicates and detect the correlated conditional statements, which are the third and fourth steps. The fifth step is the detection of the conflicting branches that caused by the conditional correlations. In the final step, we can detect the infeasible paths that contain the conflicting predicates and the inconsistent predicates.

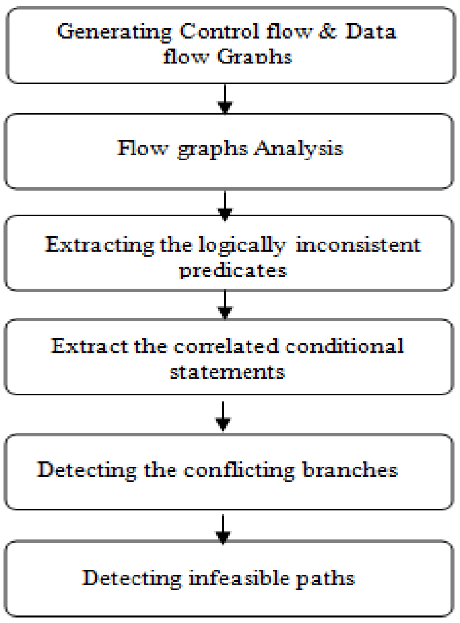

Generating Control flow & Data flow Graphs
Flow graphs Analysis
Extracting the logically inconsistent predicates
Extract the correlated conditional statements
Detecting the conflicting branches
Detecting infeasible paths

Figure 2 Experiment Methodology

4. EXPERIMENTS AND RESULTS

The process is started by generating an XML file for describing the control flow and data flow graphs for the evaluated code. Using this file helps in knowing all needed information about each node; its sequence number, type, start line, end line, etc. and the information about the edges; its name, type, source node, target node, etc. and also all information about the paths of PUT.

After obtaining this information we built the nodes, edges, paths, and pathconditions. The pathconditions represents the conditional statements along each path in PUT. We utilized it in detecting the logically inconsistent predicates, and also in detecting the conflicting branches of the correlated conditional statements.

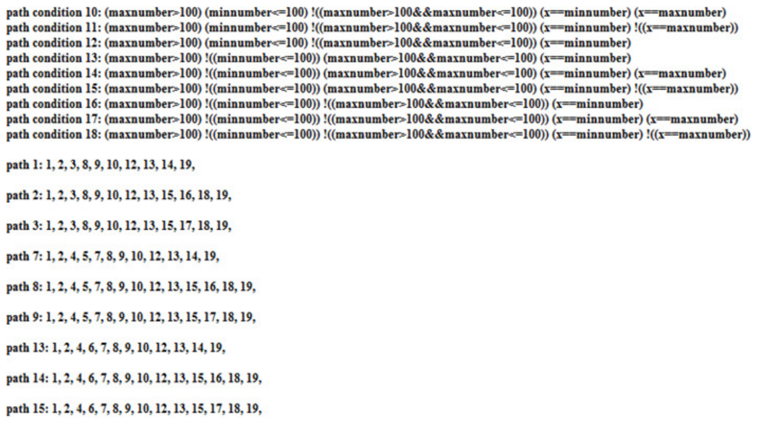
This section exhibits the results of the experiments that were performed against four examples of source codes. Each source code and its related GraphXML file are used as input for our tool.

Experimental results showed that our tool succeeded in detecting the infeasible paths that may be caused by the existence of the logically inconsistent predicates that composed of the && operation, and by the correlated conditional statements with respect to certain variable. But in the case of the existence of while loop, the application tool can detect correctly just the infeasible paths (if exists) that not traverse through the loop. The problem of loops will be maintained in the future work.

Table 1 summarizes the results of our study on four examples, the column named Predicate lists the conditional statements that are responsible for the existence of infeasible paths in each sample code. For instance, in the first example the paths that contain the predicate (maxnumber>100 && x<=100) are infeasible because the variable x equals maxnumber at some point of the code before reaching to this conditional statement. Thus, this predicate composed of two contradicting clauses that are related by && conditional operator. Figure 3 depicts the result for sample 1.

Table 1. Experimental results of this study

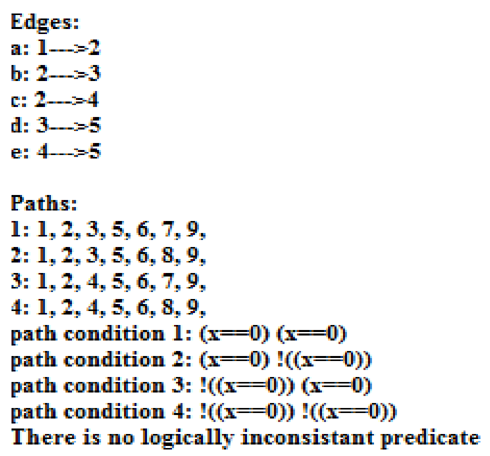
Sample	Predicate	NO. of detected infeasi. paths	No. of Actual infeas. paths	Reason of infeasible paths
Sample 1	(maxnumber>100 && x<=100)	9	9	Inconsistent predicate
Sample 2	(x == 0) … … (x == 0)	2	2	Correlated predicates & x is def-clear
Sample 2 v1	(x == 0) … … (x == 0)	3	3	Correlated predicates & x is not def-clear
Sample 2 v2	(x==0 && y!=0)	2	2	Inconsistent predicate
Sample 2 v3	(x==0 ‖ y!=0)	2	2	Inconsistent predicate
Sample 3	(a<18)…(a<30)	12	12	Correlated predicates & a is def-clear
Sample 4 (wrong results)	(input>0) … (input<=0) …	4	2	Correlated predicates & input is def-clear
Sample 4 (partial correct	(input>0) … (input<=0) …	1	2	Correlated predicates & input is



```
path condition 10: (maxnumber>100) (minnumber<=100) !((maxnumber>100&&maxnumber<=100)) (x==minnumber) (x==maxnumber)
path condition 11: (maxnumber>100) (minnumber<=100) !((maxnumber>100&&maxnumber<=100)) (x==minnumber) !((x==maxnumber))
path condition 12: (maxnumber>100) (minnumber<=100) !((maxnumber>100&&maxnumber<=100)) (x==minnumber)
path condition 13: (maxnumber>100) !((minnumber<=100)) (maxnumber>100&&maxnumber<=100) (x==minnumber)
path condition 14: (maxnumber>100) !((minnumber<=100)) (maxnumber>100&&maxnumber<=100) (x==minnumber) (x==maxnumber)
path condition 15: (maxnumber>100) !((minnumber<=100)) (maxnumber>100&&maxnumber<=100) (x==minnumber) !((x==maxnumber))
path condition 16: (maxnumber>100) !((minnumber<=100)) !((maxnumber>100&&maxnumber<=100)) (x==minnumber)
path condition 17: (maxnumber>100) !((minnumber<=100)) !((maxnumber>100&&maxnumber<=100)) (x==minnumber) (x==maxnumber)
path condition 18: (maxnumber>100) !((minnumber<=100)) !((maxnumber>100&&maxnumber<=100)) (x==minnumber) !((x==maxnumber))

path 1: 1, 2, 3, 8, 9, 10, 12, 13, 14, 19,

path 2: 1, 2, 3, 8, 9, 10, 12, 13, 15, 16, 18, 19,

path 3: 1, 2, 3, 8, 9, 10, 12, 13, 15, 17, 18, 19,

path 7: 1, 2, 4, 5, 7, 8, 9, 10, 12, 13, 14, 19,

path 8: 1, 2, 4, 5, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19,

path 9: 1, 2, 4, 5, 7, 8, 9, 10, 12, 13, 15, 17, 18, 19,

path 13: 1, 2, 4, 6, 7, 8, 9, 10, 12, 13, 14, 19,

path 14: 1, 2, 4, 6, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19,

path 15: 1, 2, 4, 6, 7, 8, 9, 10, 12, 13, 15, 17, 18, 19,
```

Figure 3. Infeasible paths results 1

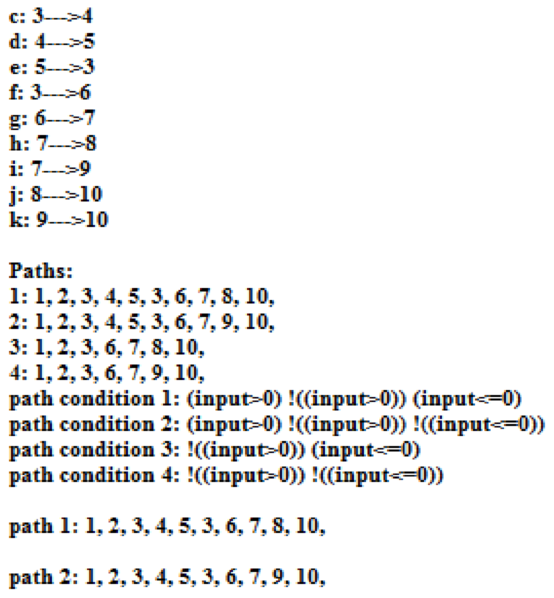
In sample 2, we can notice that our tool can effectively detect the infeasible paths that may be caused by two correlated conditional statements, the variable x is def-clear between the two conditions. Also in the case that the variable x is not def-clear between the two conditions, the decidability about the feasibility of the paths that contain these conditions can be decided based upon the branches outcome of these conditions, which can be effectively done by our tool if the variable x values before each condition is known; meaning that the variable x is assigned a value in PUT before each of the two conditions. Sample 2 v1 is an example of this case. Figure 4 depicts the result for sample 2.



```
Edges:
a: 1--->2
b: 2--->3
c: 2--->4
d: 3--->5
e: 4--->5

Paths:
1: 1, 2, 3, 5, 6, 7, 9,
2: 1, 2, 3, 5, 6, 8, 9,
3: 1, 2, 4, 5, 6, 7, 9,
4: 1, 2, 4, 5, 6, 8, 9,
path condition 1: (x==0) (x==0)
path condition 2: (x==0) !((x==0))
path condition 3: !((x==0)) (x==0)
path condition 4: !((x==0)) !((x==0))
There is no logically inconsistant predicate
```

Figure 4 Infeasible paths results 2

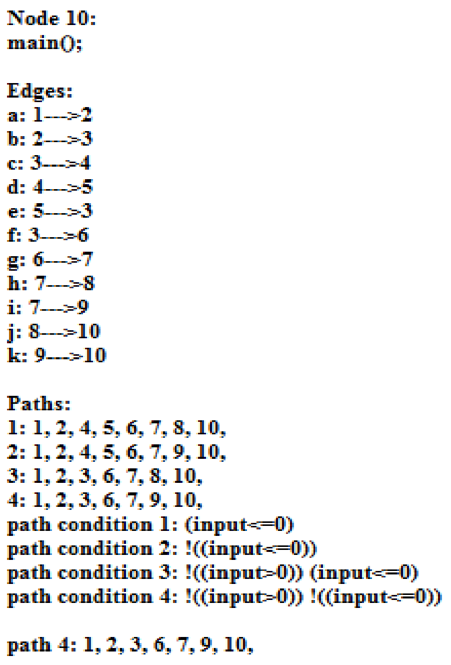
The fourth sample contains a while statement, actually, this loop structure needs more work to be maintained well, because the same condition will be used more than once in the paths that contains one or more loops of the node that contains that condition, which indeed will cause the result to be incorrect, because the condition will be used one time as is, then, when the false outcome of the condition is traversed, the condition will be negated, so, our tool will consider any path that contains such this case as infeasible, which really not correct result. So, the loop structures will be maintained in the future works. Figure 5 depicts this situation.



```
c: 3--->4
d: 4--->5
e: 5--->3
f: 3--->6
g: 6--->7
h: 7--->8
i: 7--->9
j: 8--->10
k: 9--->10

Paths:
1: 1, 2, 3, 4, 5, 3, 6, 7, 8, 10,
2: 1, 2, 3, 4, 5, 3, 6, 7, 9, 10,
3: 1, 2, 3, 6, 7, 8, 10,
4: 1, 2, 3, 6, 7, 9, 10,
path condition 1: (input>0) !((input>0)) (input<=0)
path condition 2: (input>0) !((input>0)) !((input<=0))
path condition 3: !((input>0)) (input<=0)
path condition 4: !((input>0)) !((input<=0))

path 1: 1, 2, 3, 4, 5, 3, 6, 7, 8, 10,

path 2: 1, 2, 3, 4, 5, 3, 6, 7, 9, 10,
```

Figure 5: Loop demonstration

So, in order to minimize the incorrect results and sub correct results, we will maintain in this project only the paths that contain no entrance in the loop. Figure 6 depicts this partial correlation situation.



```
Node 10:
main();

Edges:
a: 1--->2
b: 2--->3
c: 3--->4
d: 4--->5
e: 5--->3
f: 3--->6
g: 6--->7
h: 7--->8
i: 7--->9
j: 8--->10
k: 9--->10

Paths:
1: 1, 2, 4, 5, 6, 7, 8, 10,
2: 1, 2, 4, 5, 6, 7, 9, 10,
3: 1, 2, 3, 6, 7, 8, 10,
4: 1, 2, 3, 6, 7, 9, 10,
path condition 1: (input<=0)
path condition 2: !((input<=0))
path condition 3: !((input>0)) (input<=0)
path condition 4: !((input>0)) !((input<=0))

path 4: 1, 2, 3, 6, 7, 9, 10,
```

Figure 6: Loop exclusion assumption

As we can notice from Table 1, our tool can statically detect efficiently all the infeasible paths that may be caused by the existence of the logically inconsistent predicates (the clauses that are related by && operation) and the conditional correlation except in the case of while-loops, our tool can just detect the infeasible paths that do not traverse through the loop.

5. CONCLUSION AND FUTURE WORK

This paper focuses on addressing some of the main reasons for the infeasible paths, and detecting automatically and statically two types of infeasible paths; the paths that contain the logically inconsistent predicates (which is a sort of dead codes) and the paths that may be caused by the correlated conditional statements. The first type is caused by the existence of contradicting clauses that are related by the conditional operator &&. The second type may be caused by the correlation between two predicates with respect to a certain variable which is def-clear between them.

Experimental results showed that our proposed approach can detect effectively the first type of infeasible paths and effectively the second type in PUT except the PUT that contains while loop structures; it can correctly detect only the infeasible paths through the paths that do not enter the loop.

There are many ideas that can be treated and maintained in the future work; the first one is to perform auto generation of the GraphXML file that describes the CFG and DFG along PUT. Auto generation of such file can save effort, and time of the generation process, and guarantees the building of a precise GraphXML file. Another issue that is related to the GraphXML file is the creation of the nodes, edges and paths; for example, if PUT has a huge amount of paths, it will be very tiring and tedious work, in addition to the possibility that the manual generation of such GraphXML file may result in fatal errors in the creation of the paths.

The partial comparisons that were conducted by consistancyCheck() and checkForEquivalency() will be treated also in the future works.

Finally, the application tool results contain the infeasible path number and the number of its elements of nodes. In order to make the output more useful, the application tool will be modified in the future works to be able to print out PUT lines and highlighting the lines that are contained in the infeasible path, and explaining the piece of code that is responsible for the infeasibility of the path.

6. REFERENCES

[1] Ngo, M.N. and H.B.K. Tan, Heuristics-based infeasible path detection for dynamic test data generation. Information and Software Technology, 2008. 50(7): p. 641-655.

[2] Gong, D. and X. Yao, Automatic detection of infeasible paths in software testing. Software, IET, 2010. 4(5): p. 361-370.

[3] Delahaye, M., B. Botella, and A. Gotlieb. Explanation-based generalization of infeasible path. in Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. 2010. IEEE.

[4] Ngo, M.N. and H.B.K. Tan. Detecting large number of infeasible paths through recognizing their patterns. 2007. ACM.

[5] Suhendra, V., et al. Efficient detection and exploitation of infeasible paths for software timing analysis. in Proceedings of the 43rd annual Design Automation Conference. 2006. ACM.

[6] Jang, S., et al. A Study of Advanced Hybrid Execution Using Reverse Traversal. in Information Management, Innovation Management and Industrial Engineering (ICIII), 2011 International Conference on. 2011. IEEE.

[7] Arlt, S. and M. Schäf. Joogie: infeasible code detection for java. in Computer Aided Verification. 2012. Springer.

[8] Chen, C., et al. A new method of test data generation for branch coverage in software testing based on EPDG and Genetic Algorithm. in Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on. 2009. IEEE.

[9] Kalaji, A.S., R.M. Hierons, and S. Swift. Generating feasible transition paths for testing from an extended finite state machine (EFSM). in Software Testing Verification and Validation, 2009. ICST'09. International Conference on. 2009. IEEE.

[10] Blanco, R., J. Tuya, and B. Adenso-Díaz, Automated test data generation using a scatter search approach. Information and Software Technology, 2009. 51(4): p. 708-720.

[11] Sofokleous, A.A. and A.S. Andreou, Automatic, evolutionary test data generation for dynamic software testing. Journal of Systems and Software, 2008. 81(11): p. 1883-1898.

[12] Agrawal, H. Dominators, super blocks, and program coverage. in Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1994. ACM.

[13] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In FMCO, volume 4111 of Lecture Notes in Computer Science, pages 364–387. Springer, 2005.