A Method for Automated Unit Testing of C Programs

Duc-Anh Nguyen, Pham Ngoc Hung, Viet-Ha Nguyen
VNU University of Engineering and Technology
Email: {nguyenducanh, hungpn, hanv}@vnu.edu.vn

***Abstract*—This research proposes an automated test case generation method for C functions. In this method, the source code is transformed into a control flow graph corresponding to the given coverage criterion. After that, a list of feasible test paths are discovered by traversing the control flow graph using backtracking algorithm, symbolic execution, and Z3 solver. We also generate test cases for functions containing one loop or two-nested loop. A tool supporting the proposed method has been developed and applied to test on some C functions. The experimental results show the high coverage with the minimum number of test cases, the ability to improve the total time of the test case generation with a specified coverage criterion, and the increasing precision of checking the feasibility of test paths if comparing with the random technique. The experimental results display the potential usefulness of this tool for automated test case generation in practice.**

I. Introduction

The C language has been known as one of the most popular languages to develop embedded systems as well as system applications. Actually, because these systems require high reliability, it is necessary to perform a rigorous and strict testing phase. Up to now, the white-box testing is considered as a safe and effective approach in order to ensure the high reliability of systems because of the ability to discover many potential problems caused in the coding phase. The control flow testing technique (CFT) is a traditional form of the white-box testing and is used widely in the testing phase. In the current application industrial environment, the white-box testing needs to be performed automatically as much as possible to reduce the cost of time, expenditure, and human resources [1]. Especially, the expenditure can be up to 40 - 60% of the budget in the application development process. Therefore, the testing process will become more boring to testers. Automated test case generation is necessary when the projects not only do not have enough resources but also need to be evolved in regression testing.

CFT divides into two distinct directions: static direction and dynamic direction [1], [10]. In the dynamic direction, the test cases are generated by executing the source code many times in runtime environment [7]. This method deals with the complexity of the source code, easily gains a high coverage with a list of test cases, and solves the problem of infeasible test paths. However, this method has a lot of disadvantages beside many above advantages. Firstly, the number of test cases satisfying the coverage criterion might be huge or unknown which causes a lot of difficulties in management. Secondly, the total time of the test case generation can be long because the source code is compiled and run in the runtime environment repeatedly. The static direction is an effective solution to deal with many above disadvantages of the dynamic direction. This direction generates new test cases by parsing the source code instead of running source code many times [1]. With the source code that does not contain any loop, the number of test cases is minimum. In contrast, the number of test cases is specified with a smaller number in comparison with the dynamic direction. Moreover, the cost to generate the test cases of this direction is lower than that of the dynamic one because the test cases are generated without running the source code. Besides, the static direction typically reaches a high coverage. Furthermore, both of static and dynamic directions have difficulty in solving path constraints to compute input of test cases in real time. The traditional solution to this problem is using the random technique, but it leads to high cost.

This paper proposes a method to the automated test case generation for unit testing of C programs using the CFT technique in the static direction. Initially, the source code is analyzed to construct control flow graph (CFG) corresponding to the given coverage criterion. Next, we traverse the CFG to obtain all feasible test paths using the backtracking algorithm as follows. At each decision, a path from the initial vertex to this decision is checked whether it is feasible or not by SMT-Solver Z3 [11]. If this path is feasible, it means that all test paths contain the path may be feasible. All branches from the decision are therefore continued traversing. If it is not, the path is infeasible so that all test paths traverse this path are always infeasible. This result makes the traverse process from this decision terminate. After getting a list of feasible test paths, a minimum number of test paths satisfying the coverage criterion is selected from this set, and we parse the feasible test paths containing one loop or two-nested loop to generate test cases. Finally, we execute test cases in runtime environment to obtain the test report.

The structure of the paper is organized as follows. At first, Section II describes the process for control flow graph (CFG) generation from source code. The method of generating test cases from CFG is presented in Sect. III. Section IV shows an implemented tool for supporting the proposed method and experimental results. Section V discusses some related works. Finally, we conclude the paper in Sect. VI.

II. Control Flow Graph Generation

CFG is a directed graph visualizing logic structures of program simplify [8] and defined as follow.

Definition 1 (CFG): Given a function, a corresponding CFG is defined as a pair $G = (V, E)$, where $V = \{v_0, v_1, .., v_n\}$ is a list of vertices representing statements, $E = \{(v_i, v_j)|v_i, v_j \in V\} \subset V \times V$ is a list of edges. Each edge (v_i, v_j) implies the statement corresponding to v_j is executed after v_i.

With the statement/branch coverage criterion, each statement/branch is executed at least one time with a list of test cases. Therefore, vertices of CFG are corresponding to statements: decision, assignment statement, declaration statement, $return$, $break$, $continue$. Otherwise, the multiple condition coverage criterion requires all combinations of conditions in each decision are checked. It means that all decisions should be divided into single conditions in order to generate test cases more simplify.

Algorithm 1: CFG generation

input : f: source code; t: coverage criterion
output: $graph$: CFG
1: B = a list of blocks by dividing f
2: G = a graph by linking all blocks in B to each other
3: Update $graph$ by replacing f with G
4: **if** G contains $return/break/continue$ statements **then**
5: Update the destination of $return/break/continue$ pointers in $graph$
6: **end if**
7: **for** each block M in B **do**
8: **if** block M can be divided into smaller blocks **then**
9: $Generate\_CFG(M, t)$
10: **end if**
11: **end for**

Details of CFG generation algorithm are described in Algorithm 1. The input of the algorithm includes source code as a C function f and a coverage criterion t. The output is a CFG $graph$ satisfying the given coverage criterion. Before performing this algorithm, $graph$ is initialized as a global variable and contains only one vertex representing for the given source code. Initially, the source code f is divided into a list of blocks named B: $block_0$, $block_1$, ..., $block_{n-1}$, $block_n$ (line 1). In this case, the type of each block may be a statement, or a control block. Subsequently, a graph G describing the order execution of all above blocks is generated (line 2). CFG $graph$ is then updated by replacing the vertex f with the graph G (line 3). After that, if graph G contains vertices corresponding to $break/continue/return$ statements (line 4), CFG $graph$ continues to be updated by pointing these vertices to right destinations (line 5). Next, each block M of the list B is checked whether it can be continued dividing into smaller blocks (line 8). If it is, it means that block M does not satisfy the given coverage criterion. CFG $graph$ is then continued updating by parsing these smaller blocks. Otherwise, it means that B satisfies the given coverage criterion. The algorithm terminates when all blocks in $graph$ do not divide into smaller blocks any more.

III. Test Input Generation

In order to generate test inputs, a list of feasible test paths are discovered by traversing the given CFG. Path, test path are defined as follows.

Definition 2 (Path): Given a CFG $G = (V, E)$, a path is a sequence of vertices $\{v_0, v_1, .., v_k|(v_i, v_{i+1}) \in E, 0 < k < n\}$, where n is the number of vertices.

Definition 3 (Test path): Given a CFG $G = (V, E)$, a test path is a path $\{v_0, v_1, .., v_{n-1}|(v_i, v_{i+1}) \in E\}$, where v_0 and v_{n-1} are corresponding to the initial vertex and end vertex of the CFG.

The cost of process for test case generation from test paths is high due to the unexpected solving time of path constraints. In order to generate a list of test paths satisfying the coverage criterion, the backtracking algorithm is applied firstly to discover all feasible test paths. With the source code containing many loops or $if - else$ blocks, the total time for the feasible test paths generation might be terrible. To reduce the total time, our method proposes to apply an SMT solver named Z3 rather than the random technique. Furthermore, this research also proposes algorithms to evaluate the reliability of simple loop test paths and two-nested loop test paths.

A. Test Paths Generation

Algorithm 2: CFG traverse

input : v: the initial vertex of the CFG; $depth$: the maximum number of iterations for a loop; $path$: a global variable used to store a discovered test path
output: P: a list of feasible test paths
1: **if** v == $NULL$ or v is the end vertex **then**
2: Add $path$ to P
3: Save the test cases of $path$
4: **else if** the number occurrences of v in $path$ <=$depth$ **then**
5: Add v to the end of $path$
6: **if** (v is not a decision) or (v is decision and $path$ is feasible) **then**
7: **for** each adjacent vertex u to v **do**
8: $TraverseCFG(u, depth, path)$
9: **end for**
10: **end if**
11: Remove the latest vertex added in $path$ from it
12: **end if**

This research proposes Algorithm 2 to obtain feasible test paths. The input includes the initial vertex v, a maximum number of iterations for a loop $depth$, and a string $path$ used to store an obtained test path when traversing the CFG. The output is a list of feasible test paths P. Both P and $path$ are global variables and initialized to $\emptyset$. Initially, we check that if v is the end vertex or equivalent to $NULL$ (line 1). If it is, $path$ is then added to P (line 2) and its test case is collected (line 3). Otherwise, it means that v is traversed under $depth$ times (line 4), v is put at the end of $path$ (line 5). Subsequently, we check whether v is not a decision, or v is a decision and $path$ has a solution (line 6). It these conditions satisfy, all adjacent vertices u to v are then traversed (line 7, 8). If $path$ has no solution, the traverse process from this

decision terminates. Finally, $path$ removes the latest vertex added before in order to change the end branch to its negative branch (line 11). The algorithm terminates when feasible test paths satisfying the number of iterations for each loop under $depth$ times are discovered.

The value $depth$ should be chosen based on parameters: structure of loops (simple loop, nested loop, concatenated loop, or unstructured loop), structure of $if-else$ blocks, the number of loop blocks, and the number of $if-else$ blocks.

B. Test Path Generation for Loops

Each loop of a feasible test path is only executed a small number time due to the explosion of test paths when the maximum number of iterations for this loop is large enough. In fact, the block of loops may contain many potential errors so that it is necessary to check the reliability of these blocks. The paper focuses on how to generate test cases to ensure the quality of simple loop test paths and two-nested loop test paths that are defined as follows.

Definition 4 (Simple loop test path): Given a CFG $G = (V, E)$, a simple loop test path is a test path that has one loop, denoted as p:

$$p = \{v_0, v_1, .., v_k, .., v_k, .., v_{n-1} | (v_i, v_{i+1}) \in E\} \quad (1)$$

, where v_k is corresponding to a decision, v_0 and v_{n-1} are corresponding to the initial vertex and end vertex of CFG.

Definition 5 (Two-nested loop test path): Given a CFG $G = (V, E)$, a two-nested loop test path is a test path that has an inner loop and an outer loop, denoted as p:

$$p = \{v_0, .., v_k, .., v_m, .., v_m, .., v_k, .., v_{n-1} | (v_i, v_{i+1}) \in E\} \quad (2)$$

, where v_k and v_m is corresponding to the decision of the outer loop and the inner loop. v_0 and v_{n-1} are corresponding to the initial vertex and end vertex.

Test path generation for simple loops. Several test paths are generated by duplicating loop of a simple loop test path more than one time. In the case the maximum number of iterations for this loop is a concrete number n, 7 test paths are created corresponding to 0, 1, 2, a random number, n-1, n, and n+1 times. Otherwise, it means that the maximum number of iterations for this loop is not specified. In this case, 4 test paths are generated corresponding to 0, 1, 2, and a random number times.

Test path generation for two-nested loops. The key idea for testing a two-nested loop test path is destroying the structure of one loop at one time in order to obtain a simple loop test path. A set of test cases is then generated by analyzing the simple loop test path.

In order to test the inner loop, Algorithm 3 is applied as follows with the given two-nested loop test path $path$. At first, the vertices which come into and escapes the outer loop of the given two-nested loop test path are identified, named u and v respectively (line 1, 2). Next, the iteration variable of the outer loop $iterationVar$ is determined by parsing the outer loop (line 3). In this step, a variable is identified as an iteration variable when it exists in the condition of the outer loop, and its value is increased or decreased. Subsequently, we assume the outer loop had been executed under a specified number times. Therefore, an assignment statement of $iterationVar$ to an available value is saved in $newVertex$ (line 4). To destroy the loop structure of the outer loop, $output$ is then generated by removing v from $path$ (line 5). After that, $newVertex$ is inserted into $path$ before u in order to set the iteration variable of the outer loop to a new value (line 6). The algorithm terminates and returns a simple loop test path $output$.

Algorithm 3: Get test path for testing inner loop

input : $path$: a two-nested loop test path
output: $output$: a simple loop test path
1: u = the vertex which comes into the outer loop
2: v = the vertex which escapes the outer loop
3: $iterationVar$ = the iteration variable of the outer loop
4: $newVertex$= the statement to assign $iterationVar$ to an available value
5: Remove v from $path$
6: $output$ = insert $newVertex$ before u in $path$
7: **return** $output$

To check the reliability of the outer loop, the vertex which escapes the inner loop is removed from the given two-nested loop test path. The two-nested loop test path then becomes a simple loop test path.

C. Test Case Generation

1) Path Constraints Generation: Path constraints are generated by parsing a path using the symbolic execution technique and defined as follow.

Definition 6 (Path constraints): Path constraints are composed of logic expressions, denoted as PC.

$$PC = c_0 \wedge c_2 \wedge ... \wedge c_{n-1} \quad (3)$$

, where n is the number decisions of the given test path, c_i denotes one constraint corresponding to a decision in the given test path.

Algorithm 4: Path constraints generation

input : $path$: a path
output: PC: a list of path constraints
1: V = a list of vertices in $path$
2: $varList$ = $\emptyset$
3: PC = $\emptyset$
4: **for** each vertex v in set V **do**
5: Simplifies the statement corresponding to v
6: **if** v is construction statement **then**
7: Add new variable to table $varList$
8: **else if** v is assignment statement **then**
9: Update the assigned variable in table $varList$
10: **else**
11: Add vertex v to set PC
12: **end if**
13: **end for**
14: **return** PC

Algorithm 4 is applied in order to obtain path constraints PC from a path $path$. Initially, a list of vertices V in $path$ are identified (line 1). Next, table of variables $varList$ and PC are initialized to $\emptyset$ (line 2, 3). In the for block, the statement corresponding to v is simplified as much as possible (line 5). Depending on the type of this statement, there then happen three following cases. If the type of this statement is construction, a new variable is created and added to the table $varList$ (line 7). In this case, the default value of the new variable is the upper case name of itself when this variable is not initialized. In case 2, the type of this statement is assignment, the value of the assigned variable is then updated in table $varList$ (line 9). Otherwise, it means that the type of this statement is condition. In this case, PC add this statement at the end of the list (line 11). The algorithm terminates when all vertices are handled.

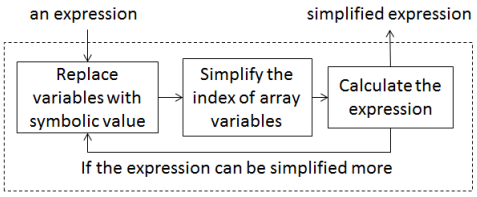

Fig. 1: The simplification process for an expression.

Figure 1 describes the simplification process for a statement in line 5 of Algorithm 4. The simplified expressions are index of arrays, the right side of assignment statements, and conditions. The details of this phase are described as follows. At first, all variables are replaced with its symbolic values. Next, the index of array variables is calculated to obtain the concrete values. Then we check whether the value of the expression is a concrete number or cannot be simplified any more. If it does not, the simplification process is continued and otherwise algorithm terminates.

2) Test Input Generation: Figure 2 describes the process for converting a path constraints to a corresponding SMT-LIB expression. At first, each logic expression in the given path constraints, which is referred to infix expression, is altered to a postfix expression. Next, the postfix expression is handled to obtain a corresponding expression tree. Finally, the expression tree is traversed to generate an SMT-LIB expression.

For example, with the logic expression $!(!(a > 0 \&\& a < 10))$. The process for converting a logic expression to SMT-LIB expression is described as follows. Firstly, the infix expression is converted to the postfix expression $a\ 0\ >\ a\ 10\ <\ \&\ !\ !$. Next, the analysis of the postfix expression is conducted to get the expression tree. Finally, the expression tree is traversed in order parent, left child, right child to obtain a SMT-LIB expression $(not(not(and(>\ a\ 0)(<\ a\ 10))))$.

IV. Implementation

We implement the proposed method in a tool called CFT4CUnit [12]. In order to show the effectiveness of the proposed method, CFT4CUnit is compared with two tools: PathCrawler and another which is described in [3]. The tool which is implemented in [3] is not available, so we had implemented it based on author idea and called it ATGC.

Table I shows comparisons in details about the average execution time for deriving all feasible test paths of 20 executions, and the number of feasible test paths between CFT4CUnit and ATGC. The input coverage criterion is branch coverage. ATGC uses the random technique to solve the path constraints with two parameters including the domain of test cases in [-10 .. 10] and 15000 iterations. In contrast, CFT4CUnit obtains the test cases by applying the powerful SMT-Solver Z3. The *time out* case happens when the execution time is over 400 seconds. With all examples without any loop, the maximum number of iterations for each loop *depth* is represented by zero. The input of CFT4CUnit and ATGC includes a C function (containing integer variables, float variables and arrays), a coverage criterion and a maximum number of iterations for each loop. The output is a list of test cases satisfying the coverage criterion.

In examples $Tritype$, $grade$, $uninit\_var$ ($depth$ = 2, 3), and GCD ($depth$ = 1..3), the number of feasible test paths in CFT4CUnit is greater than that of ATGC. Using random technique, 20 executions result in different number of feasible test paths. In contrast, the number of feasible test paths in CFT4CUnit is always identically and exactly in all of 20 executions. The feasibility of some test paths is evaluated incorrectly with ATGC.

Simple examples foo and $ComplexIndex$ only contain $if - else$ blocks without any loop. In these examples, the number of possible test paths is small and the complexity of path constraints is simple. As a result, the statistics in Table I does not show differences between two tools clearly. However, in example $Tritype$, which only contains a lot of $if - else$ blocks, shows the gap in time clearly: an approximation to 1 seconds in CFT4CUnit and 16 seconds in ATGC. Many other examples such as GCD, $Average$, $SelectionSort$, $uninit\_var$ show the effectiveness of our proposed method when $depth$ is large enough. The execution time of ATGC is greater than CFT4CUnit many times, even *time out* in two examples: GCD and $uninit\_var$ with $depth$ = 4.

Table II lists five C programs that have been used in the experiments. The first example and the final example do not contain any loops. The second example contains a simple loop and the passing parameters consist of an array variable. The third example includes a simple loop. The fourth example contains the index of array variable as an expression. In all cases, the $depth$ of CFT4CUnit is 1. With PathCrawler, the size of array variables are in [1..10] and the value of variables are in [-10..10]. In the first case, both CFT4CUnit and PathCrawler reach 100% code coverage with 5 test cases. In the case second and case third, CFT4CUnit and PathCrawler reach the same branch coverage, however, our tool generates less test cases than PathCrawler very much. In the case fourth, CFTCUnit obtains 100% code coverage with 2 test cases while PathCrawler only reachs 75% code coverage. In the case five, PathCrawler fails to generate test cases white CFT4CUnit reaches 100% code coverage. The reason is PathCrawler fails

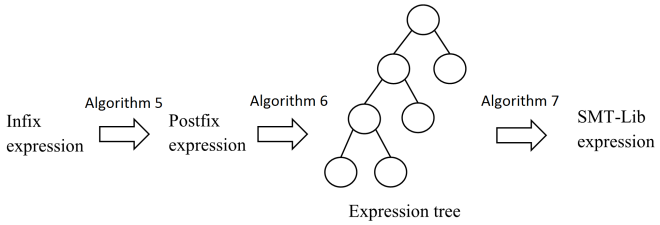

Fig. 2: The process for analyzing the logic expression to SMT-LIB format.

TABLE I: Comparison of CFT4CUnit with ATGC

Input			CFT4CUnit		ATGC	
Example	depth	total test paths	time	feasible test paths	time	feasible test paths
Tritype	0	18	<1s	7	16s	6
ComplexIndex	0	4	<1s	4	<1s	4
foo	0	3	<1s	3	<1s	3
Grade	0	6	<1s	6	<1s	5
GCD	1	20	1s	16	61s	6
	2	36	2s	32	109s	10
	3	68	2s	64	211s	18
	4	132	4s	128	time out	-
Average	1	6	< 1s	3	8s	3
	2	14	<1s	7	22s	7
	3	30	1s	15	63s	15
	4	62	2s	31	138s	31
SelectionSort	1	4	<1s	3	<1s	3
	2	13	$< 1s$	3	15s	3
	3	40	1s	3	51s	3
	4	121	4s	8	145s	8
uninit_var	1	23	1s	3	8s	3
	2	83	2s	14	75s	13
	3	275	13s	22	261s	20
	4	875	27s	25	time out	-

TABLE II: Comparison of CFT4CUnit with PathCrawler

Input	PathCrawler		CFT4CUnit	
	test case	coverage	test case	coverage
Grade	5	100%	5	100%
Average	192	83.33%	3	83.33%
GCD	201	100%	7	100%
ComplexIndex	4	75%	2	100%
Foo	-	-	3	100%

to generate the first test input, so the test case generation can not continue.

V. Related Works

There are many works that have been recently proposed for automated test case generation, by several authors. Focusing only on the most recent and closest ones, we can refer to [1], [3], [4], [9].

Nicky Williams and et al. introduce a tool named PathCrawler to generate test cases using a solver which is developed in CEA LIST [1]. The solver is based not on an SAT, linear or SMT solver but on the finite domain constraint solver named COLIBRI. COLIBRI is implemented in constraint logic programming (CLP). COLIBRI can treat non-linear arithmetic and provides specialized constraints for modular integer. PathCrawler continues to develop COLIBRI to treat bit operations, dynamic allocation, arrays with variable dimensions, and array accesses using variable index values arithmetic and floating-point arithmetic.

Sangeeta Tanwer and et al. proposed a process for generating test cases using random technique in [3]. It is a traditional technique to generate a list of test cases automatically. To generate test cases effectively, the range of values variable should be small. Moreover, the number of random generation is enough large to treat special test paths. However, the total time of this technique is high and it is difficult to specify the feasibility of test paths.

Zheng Wang proposed an algorithm to generate test cases in [4]. The proposed algorithm can treat pointer variables, primitive variables, structures, and arrays. To handle pointer variables, there are two tasks. The first one is to construct proper memory storages to which these pointers point. The second is to generate the value stored in that memory storages. In the case the program contains several pointers pointing to the same variable, a relation matrix of pointers on the base of the path constraints is build. Structures and arrays are divided into primitive variables. This method can deal with in the case which the index of the array is a variable exciting but not completely due to the explosion of array variables.

Leonardo de Moura and et al. have published a paper about Z3 solver in [9]. Z3 is a new SMT solver from Microsoft Research and based on SAT Solver. Many problems in software verification and software analysis can be solved with Z3. To solve path constraints with Z3, the path constraints must be converted to SMT-LIB format. This work can be conducted through API or our proposed algorithm. Z3 supports arithmetic, fixed-size bit-vectors, extensional arrays, data types, uninterpreted functions, and quantifiers.

Not only Z3, some other SMT solvers such as SMTInterpol [13], MathSAT [14], CVC4 [15], and Yices [16] can generate test cases from path constraints effectively. All of these SMT solvers accept SMT-LIB format.

VI. Conclusion

We have presented a method for generating test cases for C functions using a static direction. Our method can treat integer variables, float variables, and arrays. At first, CFG is generated with inputs including source code as a C function, a coverage criterion, and a maximum number of iterations for a loop. Feasible test paths are then explored by traversing the generated CFG using the backtracking algorithm, symbolic execution and an SMT-Solver named Z3. At each decision in the traverse process, we check whether the path from the initial vertex to this decision is feasible or not by applying three required steps as follows. Initially, the path is analyzed to generate corresponding path constraints by symbolic execution. Subsequently, the path constraints are transformed into SMT-LIB expression. Finally, SMT solver Z3 solves this SMT-LIB expression to generate test cases. If the SMT-Lib expression has solution, we continue traversing the children of the decision. If it does not, the traverse process from this decision is terminates because all test paths includes this path are always infeasible. After getting all feasible test paths, the test paths satisfying the given coverage criterion are chosen as minimum as possible. With test paths have one loop or two-nested loop, the reliability of each loop is checked by performing the testing loop independently to each other.

The best advantages of the proposed method are high coverage and lower cost in comparison with the dynamic direction. The total time to discover all feasible test paths is improved by using SMT solver Z3. The effectiveness of Z3 is shown clearly in experimental results. Because of specifying the feasible test paths, it is easy to choose minimum feasible test paths to satisfy the coverage criterion. Our method is applied not only for C function but also for others programming languages. Moreover, test cases can be discovered by using any SMT solvers, which accept SMT-LIB format. Besides these advantages, the proposed method still has some limitations. The implemented tool has been tested with small and medium size C functions with primitive types. In addition, our method cannot deal with pointers, structures, string, etc.

Currently, we are applying this tool for more complex functions to demonstrate the effectiveness of the proposed method. At the same time, we are going to improve the implemented tool supports for other variable types: pointers, structures, string, etc. by combining with the dynamic direction. Besides, we continue finding out solutions to generate expected output automatically. Moreover, we are extending this method to treat with program level rather than function level. A version for generating test cases for Java programming language is developing.

By analyzing the structure of path constraints, its type can be QF_LRA, QF_LIA, etc. The ranks of the well-known SMT-Solvers for each type are updated annually . It means that, depending on the type of path constraints we can choose a suitable SMT-Solver to generate test cases with the lowest cost. Therefore, the problem of choosing suitable SMT solvers is another issue and need to be conducted additional researches. In this research, we completely choose another SMT solver, which is accepted SMT-LIB format, in order to obtain test cases. In this research, we completely choose another SMT solver, which is accepted SMT-LIB format, in order to generate test cases. The problem of choosing a suitable SMT solver is another issue and need to be conducted additional researches.

Moreover, some information about how variables are defined and used can be discovered without generating Data Flow Graph by parsing the CFG. In future, our tool generates test cases for not only satisfying three commonly coverage criteria, but also checking the reliability of using variables.

Acknowledgments

This work is supported by the project no. QG.16.31 granted by Vietnam National University, Hanoi (VNU).

References

[1] Nicky Williams, Bruno Marre, Patricia Mouy, Muriel. Roger, PathCrawler: automatic generation of path tests by combining static and dynamic analysis, in: EDCC-5, 2005
[2] R. A. DeMillo, A. J. Offutt: Constraint-based automatic test data generation, in: IEEE Transactions on Software Engineering, 17(9), pp. 900-910, 1991
[3] Sangeeta Tanwer, Dr. Dharmender Kumar, Automatic testcase Generation of C Program Using CFG, in: IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 4, No 8, July 2010
[4] Zheng Wang, Test Data Generation for Derived Types in C Program, in: Third IEEE International Symposium on Theoretical Aspects of Software Engineering, 2009
[5] Arthur H. Watson, Thomas J. McCabe, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, in: NIST Special Publication 500-235
[6] J. C. King, Symbolic execution and program testing, in: Commun. ACM, vol. 19, no. 7, 1976, pp. 385-394
[7] M.Prasanna, S.N. Sivanandam, R.Venkatesan, R.Sundarrajan, A survey on automatic test case generation, in: Academic Open Internet Journal, Vol. 5, 2005
[8] Robert Gold, Control flow graph and code coverage, in: Int. J. Appl. Math. Comput. Sci., Vol. 20, No. 4, 2010, pp. 739-749
[9] Leonardo de Moura, Nikolaj Bjrner, Z3: An Efficient SMT Solver, in: Proceeding TACAS'08/ETAPS'08 Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, pp. 337-340
[10] Sai Zhang, David Saff, Yingyi Bu, Michael D. Ernst, Combined Static and Dynamic Automated Test Generation, in: ISSTA'11, pp. 353-363
[11] https://z3.codeplex.com/releases
[12] http://coltech.vnu.edu.vn/~hungpn/CFT4CUnit/
[13] http://ultimate.informatik.uni-freiburg.de/smtinterpol/
[14] http://mathsat.fbk.eu/
[15] http://cvc4.cs.nyu.edu/web/
[16] http://yices.csl.sri.com/