A Search Strategy Guided By Uncovered Branches For Concolic Testing*

Qixing Dong
School of Computer Science and Technology, University of Science and Technology of China.
Hefei, Anhui, PR China.
dqx@mail.ustc.edu.cn

Jun Yan
Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences.
Beijing, PR China.
yanjun@otcaix.iscas.ac.cn

Jian Zhang
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences.
Beijing, PR China.
zj@ios.ac.cn

Fanping Zeng
School of Computer Science and Technology, University of Science and Technology of China.
Hefei, Anhui, PR China.
billzeng@ustc.edu.cn

***Abstract*—Test generation plays an important role in software testing. Concolic testing runs concrete executions on programs simultaneously with symbolic executions. It faces the same problem as symbolic execution when generating test inputs, namely the combinatorial explosion of the path space. Most of the existing approaches can search only a fraction of the path space. Therefore, how to cover as many branches as possible with a few test inputs becomes an important research issue. Upon this issue, the paper proposes a heuristic search strategy based on inter-procedural control flow graph. The proposed strategy calculates the approximate mathematic expectation of uncovered branches. Then this expectation guides the search process to select the optimal branch to form a path condition. According to the path condition, the test input that may cover a large number of uncovered branches is generated. Experiments on three classes of test benchmarks show that, compared to the other strategies, the proposed strategy can significantly reduce the number of test inputs for covering reachable branches, and achieve high branch coverage quickly.**

Keywords-concolic testing; dynamic symbolic execution; test generation; uncovered branch expectation

I. INTRODUCTION

Software testing is a key step in the software development cycle. Test generation [1] plays an important role in software testing. Concolic testing [2-4] which is also called dynamic symbolic execution[5, 6] or directed random testing [3], is a new technique for automated test generation. It runs concrete executions on programs with symbolic execution [4]. Since there may be many or even infinite paths in programs under test, concolic testing suffers from the problem of the path space explosion [7]. This problem makes concolic testing hard to scale to large programs. Therefore, it becomes an interesting research issue to achieve high branch coverage quickly while exploring only a small fraction of the path space [4, 7-10].

Many search strategies for concolic testing have been proposed to explore the path space [4, 9-11]. Among them, the primary search strategies are depth-first-search and breadth-first-search strategy that search the paths one by one. They are impractical due to the explosion of the path space. PathCrawler [8] employs depth-first-search by limiting the number of loop iterations. Burnim and Sen [4] propose a search strategy that calculates the distance to uncovered branches and selects branches with shorter distance preferentially. Dong et al. [9] propose a strategy that makes use of Control Flow Graph (CFG), and branch residual is used to guide the process of test path generation. In order to generate the path that could cover the largest number of uncovered branches, more than one test input should be generated in this strategy.

In order to achieve high branch coverage with fewer test inputs, a heuristic search strategy based on Inter-procedural Control Flow Graph (ICFG) is proposed in this paper. It selects the execution path according to the mathematic expectation of uncovered branches that can be covered by the path. The larger the expectation is, the more likely the path is selected. The expectation of uncovered branches is evaluated approximately by assigning weights to branches. Experimental results show that the proposed strategy can reduce 93%, 91%, 88% of test inputs in the best case respectively, compared to the random-selection strategy, the strategy in which the branch with the shortest distance is searched first [4] and the strategy which selects branches with more "branch residual" preferentially [9].

The rest of the paper is organized as follows: The next section presents the key idea and related definitions of the proposed strategy. Section III describes the algorithm for calculating the branch weight. Section IV reports the experiments and Section V concludes.

II. UNCOVERED BRANCHES GUIDED SEARCH STRATEGY

ICFG is a directed graph which consists of nodes and edges. A node in ICFG represents a sequence of consecutive statements and an edge represents a branch in the program. A function may be called more than once, so branches in the function will appear multiple times in ICFG. For ease of assigning branch weights, the multiple occurrences of the same branch are treated as different branches (a map from branches in ICFG to the real branches in programs is kept to make sure the branch coverage is counted accurately). ICFG is somewhat different from CFG. The control flow in ICFG can move among functions, while in CFG can only move in the same function. For example, the function *Abort* will terminate a program. In ICFG its subsequent node is the end node of ICFG, while in CFG it is the last node of the function that includes it. The ICFG knows the relationship between the branches in different functions, so the information in ICFG about the program's structure is more accurate. ICFG is adopted to represent the program's static structure in the proposed strategy.

High branch coverage will be achieved with fewer test inputs, if the path including the maximum number of

* This work was supported by Natural Science Foundation of Anhui Province (Grant No. 11040606M131), and National Natural Science Foundation of China (Grant No. 91118007 and 61003026).

uncovered branches is selected preferentially to generate test inputs. According to ICFG, the path containing the maximum number of uncovered branches is easy to find. However, in the context of concolic testing, it is not easy to generate inputs that execute along such a path [2]. The proposed search strategy in this paper tries to generate such test inputs preferentially. In order to explain this strategy, some concepts are introduced.

Definition 1. If there is an execution path, in which the branch B_j appears after the branch B_i, then B_j is a subsequent branch of B_i. If there is no branch between B_i and B_j along the path, then B_j is an Immediate Subsequent Branch (ISB) of B_i.

Definition 2. A Prefix Path (PP) of the branch B_i is a path from the program's entrance to B_i. A Prefix Path Condition (PPC) of B_i is the path condition corresponding to a PP of B_i. A Prefix Test Input (PTI) of B_i is a test input generated based on a PPC of B_i. Note that a branch may have more than one PP. For each PP, correspondingly, there is a PPC and a set of PTIs.

A PP may be a partial path. If a test input is generated based on a partial path condition, then its execution path is uncertain. It may go along one of the candidate paths. Thus, the number of branches that the test input can cover is unknown.

Definition 3. Suppose the branch B_i has a PP that contains branches $B_1,\cdots, B_{i-1}, B_i$, and PTI_i is a PTI corresponding to the PP of B_i. The Branch Expectation (BE) of B_i is the mathematical expectation of all the branches covered by PTI_i except $B_1,\cdots, B_{i-1}$. The Uncovered Branch Expectation (UBE) of B_i is the mathematical expectation of all the uncovered branches covered by PTI_i except $B_1,\cdots, B_{i-1}$.

The test input that can execute through the branch with the largest UBE is likely to cover the largest number of uncovered branches. However, the UBE is difficult to calculate accurately. First, the size of the input space for every path is different, and the probability of the execution of a path cannot be calculated without any analytical technologies. Second, the sampling distribution of the generated test inputs is uncertain. Therefore, we suppose that the probability of that PTI_i executes along each path through B_i is the same. Then the approximate UBE can be calculated. Note that all the UBEs in the following sections are approximate. We define the weights of edges in ICFG, so that the weight of a branch approximates to its UBE. The edges are weighted according to the rules below. Note that W_{avg} stands for the average weight of all the ISBs of a branch and α is a positive number less than 1.

1) A branch without any ISBs: if it is uncovered, its weight is 1. Otherwise 0;

2) An uncovered branch with some ISBs: its weight is W_{avg} +1;

*3) A covered branch with some ISBs: if all the weights of its ISBs are the same, its weight is α*W_{avg}. Otherwise its weight is W_{avg}.*

According to these rules, we have the following theorem.

Theorem 1. Suppose that the value of α is 1, then the above-defined branch weight is equal to the UBE of the branch.

Proof. (Inductive Reasoning) Suppose the branch B has no ISB. If it has been covered, then the PTI of B cannot cover any uncovered branches after B. Otherwise, it can cover an uncovered branch. So the UBE is either 0 or 1. According to rule 1, the weight is equal to the UBE for such branches.

Suppose the branch B has at least one ISB, and all the weights of B's ISBs are the same. Assume PP' is a PP of B, and the branches in PP' are denoted as $B_1, B_2,\cdots, B_{i-1}, B$. When providing a PTI corresponding to PP' as a test input, the probability of covering each ISB of B is the same. Therefore the average weight of B's ISBs is equal to the mathematical expectation of all the uncovered branches that could be covered by the PTI except $B_1, B_2,\cdots, B_{i-1}, B$. Note that the difference between W_{avg} and B's UBE is whether B is taken into account when calculating their value. If B is covered, then it is not part of the uncovered branches that could be covered by PTI. Thus, B makes no contribution to its UBE. So W_{avg} is equal to B's UBE. If B is uncovered, then PTI will certainly cover B. It means that B contributes 1 to its UBE. So W_{avg}+1 is equal to B's UBE. Therefore, the weight is equal to the UBE. QED.

In fact, α is set to be less than 1, so that the branch closer to uncovered branches is assigned a larger weight when some branches' UBEs are the same. The closer a branch is to an uncovered branch, the higher the probability that the branch's PTI covers the uncovered branch is. There is always more than one branch whose distance to the uncovered branches is the shortest. It is easy to prove the following theorem: the branch with the maximum weight is one of the branches whose distances to uncovered branches are the shortest.

The smaller α is, the smaller the weight of the branch far from uncovered branches is; the larger α is, the closer the weight is to the UBE. The number α is set to 0.75 in practice.

III. The Algorithm for Calculating Weights

The algorithm for calculating branch weight is shown in Fig. 1. It adopts a memo to record the already-calculated weights, and retrieves from the memo when the already-calculated weights are needed. The time complexity of the memorized algorithm is $O(n)$, where n represents the number of branches. When there is a cycle in the ICFG, the algorithm may come to a deadlock, namely the branch weight should be known previously while calculating the weight of a branch in the cycle. Due to this, a wait queue is used to record the branches to be calculated. If a branch in the wait queue is to be calculated again, the algorithm returns 0 or 1 directly based on whether or not the branch is covered.

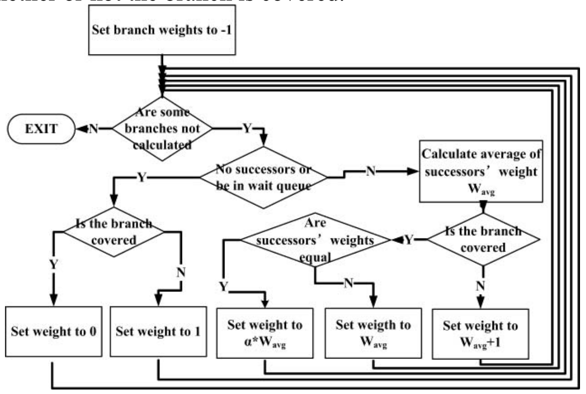
Set branch weights to -1
EXIT
N
Are some branches not calculated
Y
No successors or be in wait queue
Y
N
Calculate average of successors' weight W_{avg}
Is the branch covered
Y
N
Are successors' weights equal
Y
N
Is the branch covered
Y
N
Set weight to 0
Set weight to 1
Set weight to α*W_{avg}
Set weigth to W_{avg}
Set weight to W_{avg}+1

Figure 1. The flow chart of the alogrithm for calculating branch weight.

The PPs of the branch with the optimal weight may be infeasible and the PPCs may be unsatisfied. Once such a branch is found, its weight is set to 0. And then the search process selects the branch in sequence from the candidate queue where candidates are sorted by their weights.

If the program under test contains loops, the branch inside a loop will repeatedly appear in the execution path. Since the branch's weight is fixed, it will occupy large and continuous space of the candidate queue, so that the branches with smaller weights cannot be put in the candidate queue. In addition, if there are no solutions for the PPCs of the repeating branch, the search process will repeat trying to solve the PPCs. So the repeating branch has to be penalized: find the branches that represent the repeating branch in the candidate queue and number them from 1 randomly. Then assign the weight of such a branch with the quotient resulting from that its original weight divides its number. Thus, there are only a few of branches corresponding to the repeating branch in the candidate queue. And these branches scatter over the candidate queue. The penalization is independent of the algorithm in Fig. 1. It will not affect other branches' weights.

IV. Experiments and Evaluation

The proposed strategy is implemented based on an open source concolic testing tool named CREST[1]. In order to evaluate the efficiency of the proposed strategy (Expectation for short), experimental comparisons between Expectation and other strategies are conducted. The compared strategies include bounded depth-first-search, random-selection search, the CFG-Directed search upon the shortest branch distance proposed by Burnim and Sen [4] and the branch residual guided search based on CFG proposed by Dong et al. [9]. These strategies are DFS, Random, Shortest, BR for short respectively. DFS is the primary and classic search strategy. In the random-selection strategy, branches are randomly selected without any heuristics. Shortest and BR both have similarity with Expectation, namely all of them make use of the program's static structure. The subjects consist of three parts: (1) the benchmarks released with CREST; (2) siemens benchmark suite[2]; (3) *grep* 2.2.

TABLE I. The Number Of Test Inputs Required To Cover All Reachable Branches For Subjects In part 1 (Average over 10 runs).

Subject	Code line/ Number of branches	DFS	Random	Shortest	BR	Expectation
cfg_search_test	20 / 6	2	2	2	2	2
cfg_test	52 / 12	7	32.2	36.2	19.1	10.8
function	14 / 2	2	2	2	2	2
math	14 / 2	2	2	2	2	2
simple	13 / 2	2	2	2	2	2
uniform_test	19 / 8	5	19.2	5	16	5

The results of some subjects in part 1 (*concrete_return*, *structure_return*, *structure_test*) are not reported because the test inputs cannot affect their execution path, or their structures become sequential after compiling optimization. As shown in Table I, the programs in part 1 have simple structures and a little number of branches, so that DFS can cover all branches with a little number of test inputs. Since CFG cannot represent the entire control flow of a program, Shortest cannot take branches inside the nested function into account when calculating branch distance. Therefore, the resulting branch distance is shorter than the real. For BR, it also suffers from such a problem. Furthermore, in order to obtain the path that contains the maximum number of uncovered branches, BR has to generate many test inputs. For *cfg_test*, Expectation reduces 66.5%, 70.1%, 43.5% of test inputs respectively, compared to Random, Shortest, BR.

When testing *print_token*, *print_token2* and *tot_info* in part 2, CREST cannot construct an appropriate composition of characters and digits to generate a valid test input. Therefore the execution paths of test inputs generated by different strategies are the same. As a result, the experiments performed on these subjects are worth nothing. Experimental results of the remaining four subjects in part 2 (Fig. 2 - Fig. 5) suggest that Expectation is better than Shortest, Random and BR. Expectation achieves the same or higher branch coverage than other strategies when the last test input is generated. Table II shows the specific data comparison.

Fig. 2 - Fig. 5 suggest that Expectation reduces the number of test inputs required to cover branches and accelerates the speed of covering branches. Compared to Expectation, the curves of Shortest and Random are flatter. Because they just try to generate test inputs that can cover an uncovered branch, instead of the maximum number of uncovered branches. BR tries to find the path that can cover the maximum number of uncovered branches. Its search process begins from the program's main entry point, then selects the branch with the most "residual" one by one. In order to update the execution path, concrete and symbolic executions are run again and again [9]. As a result, when the best path is found, many test inputs have been generated. Furthermore, BR records the paths that have been explored. So its space cost is too high. This makes it fail to scale to large programs.

When performing experiments on *grep* 2.2, as shown in Fig. 6, Shortest has a better result overall than Expectation. In fact, Expectation covers more branches than other strategies at the beginning of the test-generation process. Since CREST adopts the local search algorithm [4], the best candidate is selected while the others are discarded. However, if the best candidate is infeasible, CREST has to restart. When using Expectation, CREST restarts after every 312.5 test inputs are generated. Fig. 7 which is partial enlargement of Fig. 6 indicates that Expectation covers branches faster than other strategies during the first iteration of the search. The path space explored by Expectation during different iterations is similar. In such situations, the path space involved with the second best candidate branches is hard to explore. So few uncovered branches can be covered during the latter iterations. Expectation will do better, when being integrated with a global search algorithm.

V. Conclusion and Future Work

A heuristic search strategy for concolic testing is proposed in this paper. It generates fewer test inputs, but could cover the same number of branches by weighing every branch based on the approximate mathematic expectation of uncovered branches. The experiments indicate that combining the strategy with other heuristics may lead to an improvement. We will do more work about the combination in the future.

1 crest-automatic test generation tool for C, http://code.google.com/p/crest/
2 Aristotle Analysis System -- Siemens Programs, HR Variants, http://www.cc.gatech.edu/aristotle/Tools/subjects/

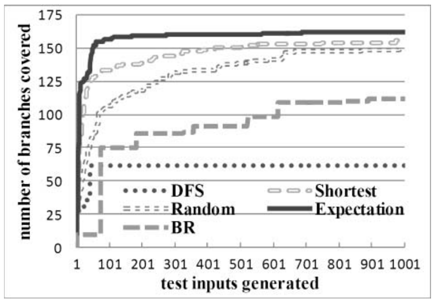

Figure 2. Experimental results on replace.

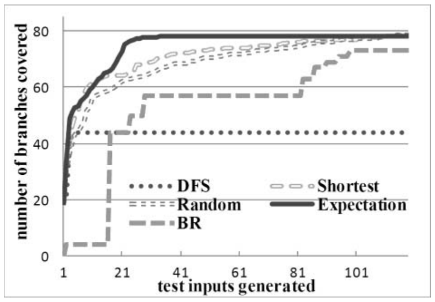

Figure 4. Experimental results on schedule2.

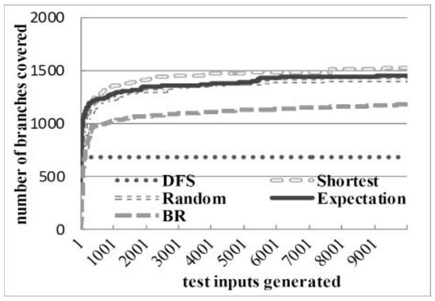

Figure 6. Experimental results on grep.

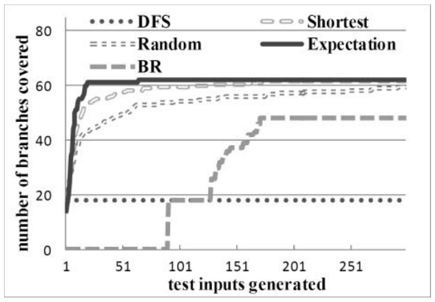

Figure 3. Experimental results on schedule.

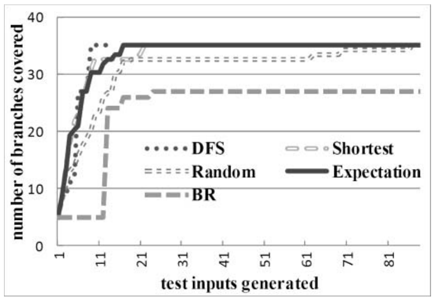

Figure 5. Experimental results on tcas.

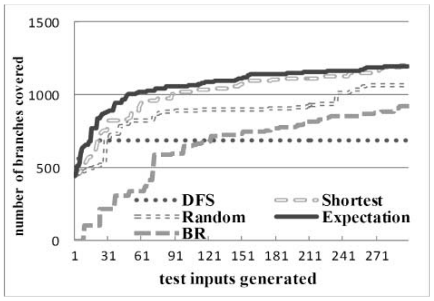

Figure 7. The first 300 test inputs for grep.

TABLE II. EXPERIMENTAL RESULTS ON SIEMENS, DFS IS NOT SHOWN FOR ITS POOR PERFORMANCE (AVERAGE OVER 10 RUNS)

Subject	Test inputs required to cover reachable branches				The percent of branches that Expectation covers more than others, when Expectation covers all reachable branches			The percent of test inputs that Expectation generates less than other strategies to cover all reachable branches		
	Expectation	*Shortest*	*Random*	*BR*	*Shortest*	*Random*	*BR*	*Shortest*	*Random*	*BR*
replace	391	4509	4957	3265	8.1%	19.9%	78.0%	91.3%	92.0%	88.0%
schedule	65	291	903	441	6.9%	19.2%	40.9%	77.7%	92.8%	85.2%
schedule2	34	116	112	206	10.5%	18.3%	36.8%	70.7%	69.6%	83.5%
tcas	17	22	87	124	7.3%	9.0%	45.8%	22.7%	70.0%	86.7%

REFERENCES

[1] J. Edvardsson, "A survey on automatic test data generation," Proc. the 2nd Conference on Computer Science and Engineering in Linkoping (ECSEL 99), Otc. 1999, pp. 21-28.

[2] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," Proc. the 10th European Software Engineering Conference held jointly with 13th ACM Symp. Foundations of Software Engineering (ESEC/FSE 05), ACM Press, Sept. 2005, pp. 263-272, doi:10.1145/1081706.1081750.

[3] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," ACM SIGPLAN Notices, vol. 40, Jun. 2005, pp. 213-223, doi:10.1145/1064978.1065036.

[4] J. Burnim, and K. Sen, "Heuristics for scalable dynamic test generation," Proc. the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 08), ACM Press, Sept. 2008, pp. 15-19, doi:10.1109/ase.2008.69.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," ACM Transactions on Information and Systems Security, vol. 12, Dec. 2008, pp. 10(1)-10(38), doi:10.1145/1455518.1455522.

[6] N. Tillmann, and J. de Halleux, "Pex-white box test generation for .NET," Proc. Tests and Proofs (TAP 08), Springer, Apr. 2008, pp. 134-153, doi:10.1007/978-3-540-79124-9_10.

[7] P. Godefroid, "Compositional dynamic test generation," Proc. the 34th annual ACM Symp. Principles of Programming Languages (POPL 2007), ACM Press, Jan. 2007, pp. 47-54, doi:10.1145/1190216.1190226.

[8] N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: Automatic generation of path tests by combining static and dynamic analysis," Proc. Dependable Computing (EDCC 05), Springer, Apr. 2005, pp. 281-292, doi:10.1007/11408901_21.

[9] Y. Dong, M. Lin, K. Yu, Y. Zhou, and Y. Chen, "Achieving high branch coverage with fewer paths," Proc. IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW), IEEE Press, Jul. 2011, pp. 155-160, doi:10.1109/COMPSACW.2011.35.

[10] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Strategies for scalable symbolic execution-driven test generation for programs," Science China Information Sciences, vol. 54, Sept. 2011, pp. 1797-1812, doi:10.1007/s11432-011-4368-7.

[11] X. Tao, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," Proc. the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 09), IEEE Press, Jul. 2009, pp. 359-368, doi:10.1109/DSN.2009.5270315.