Optimal MC/DC Test Case Generation

Joxan Jaffar
National University of Singapore
joxan@comp.nus.edu.sg

Sangharatna Godboley
National University of Singapore
sanghara@comp.nus.edu.sg

Rasool Maghareh
National University of Singapore
rasool@comp.nus.edu.sg

***Abstract*—We present a new method for automated test case generation based on symbolic execution and a custom process of interpolation. The method first identifies program execution paths in order to define a corresponding set of test inputs. It then annotates the program with assertions so as to identify feasible and infeasible cases, the former of which are processed to produce the desired test inputs. The main contribution is that performing symbolic execution using a custom form of interpolation significantly prunes the search space. Our main result is that the set of Modified Condition/Decision Coverage (MC/DC) test cases we produce is *optimal*.**

***Index Terms*—Software Testing, Code Coverage, MC/DC, Symbolic Execution**

I. Introduction

In testing, MC/DC [1] requires only a linear number of tests cases with respect to the number of atomic conditions (AC) in a program, in contrast to the combinatorially explosive Multiple Condition Coverage (MCC). In industry, it is mandatory to produce MC/DC report for Level A category software (critical-safety application in avionics domain) [1]. Now even though MC/DC requires just a small number of test inputs, generating these automatically remains a challenging task.

Our method, which is based on Dynamic Symbolic Execution (DSE), addresses this problem. DSE is a promising approach to identify all the bugs present in a program. In this approach, a symbolic execution engine traverses the symbolic execution tree (SET) of a program by path enumeration. This has an advantage over traditional testing tools because the generation of the SET is done in a systematic and controllable way. In particular, if the SET is fully traversed, then we have full knowledge about the behaviour of the program. For example, we could prove an assertion, or discover a bug. Since the SET is often exponential in the size of the program, full exploration is unlikely. The key property we take away from DSE is that upon exploring one path, it is possible to generate a concrete instance of the start state, i.e. an (input) test case, which would execute along this path.

The main contribution of our method is to employ *pruning DSE with a specialized interpolation*. The method is *optimal* in the sense that it generates test cases which produce *the highest possible MC/DC score*. In previous methods, eg. [2], [3], optimality is only possible with very small programs.

II. Proposed Framework

Our method of DSE comprises three main components: A Sequence Generator (SG), a Resource Annotator (RA), and DSE. The SG generates a collection of boolean assignments, a "sequence", to the atomic boolean expressions appearing in predicates in the (C) program. This collection, in accordance with standard MC/DC methodology, characterizes a corresponding collection of test inputs which gives an MC/DC score, represented as a percentage. If there is a test input for each one of these sequences, then the MC/DC score will be 100%. Note importantly, that such a perfect score is not always possible: some sequences are *infeasible*, that is, there is no test input which would display the sequence in real execution. Importantly, our SG generates a *sufficient* collection of sequences. This means that getting test inputs for the *feasible subset* of these sequences will lead to the *optimal* MC/DC score. Our SG obeys *short-circuit evaluation*.

Consider an example predicate and its four ACs: `(a<0 || ((b<0 || c<0) && d<0))`. Our SG generates six short-circuited sequences S_1 to S_6 where 1, 2, 0 represent *True*, *False*, and *Don't Care* values for each of the ACs: $\{1000\}$, $\{2101\}$, $\{2102\}$, $\{2211\}$, $\{2212\}$ and $\{2220\}$.

Next, the RA *annotates* the original program so that DSE will either discover an execution path which corresponds to each given sequence, or report that there is no such path. Because of this complete determination of a sequence, i.e. a test input or a proof of infeasibility, our method is *optimal* and produces the highest possible MC/DC score.

The RA first adds a ghost resource variable (κ), initialized to zero, before the predicate. Next, for each sequence s, it annotates the LLVM IR of the (C) program. For each truth value in the input sequence, if not exists, an update to κ is added. The update values for *True*, *False*, and *Don't Care* would be $\kappa = \kappa * 10 + 1$, $\kappa = \kappa * 10 + 2$ and $\kappa = \kappa * 10$. We will demonstrate this more in the following example.

Consider the control flow graph (CFG) of the predicate in Fig. 1 where ⟨T⟩ and ⟨F⟩ represent the true and false branches and ⟨postDom⟩ is the post dominator basic block of the predicate. A set of increment basic blocks (presented in red color in Fig. 1) are added to the LLVM IR such that by taking the respective path of a sequence the value of κ would be equal to the aforementioned value of the sequence at ⟨postDom⟩. Next, an assert instruction is added after ⟨postDom⟩ where each value represents one of the sequences. This program will violate the assertion for all the sequences.

Finally, we address the *path explosion* problem of DSE. Here we borrow and adapt recent technology. Pruning DSEs such as TRACER [4] utilize *abstraction learning* [5]. This method more popularly is known as "lazy annotations" [6], [7]. The core feature in abstraction learning is *subsumption* of

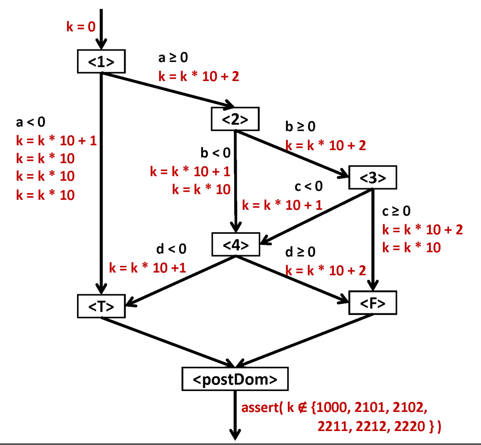

Figure 1: Annotated CFG of the Example Predicate

Table I
ANALYSIS RESULTS FOR BASELINE AND OUR METHOD

Programs	LOC	#P/ #AC/ #S	BASELINE		OUR METHOD	
			#T	#F/#I/ #U	#T	#F/ #I/ #U
tcas	301	10/51/86	4.9	79/7/0	6	79/7/0
zodiac	47	12/48/ 84	1.2	58/26/0	1.1	58/26/0
nsichneu	3919	126/625/878	257	769/109/0	1752	769/109/0
pals3	378	3/41/79	10.9	61/18/0	3.15	61/18/0
p28-s	2480	399/1425/1826	3347	465/1361/0	1487	465/1361/0
psyco1	437	10/99/190	∞	7/0/183	47	79/111/0
pals2	680	6/101/196	∞	16/0/180	3150	181/15/0
p10	637	136/992/1591	∞	282/0/1309	1597	564/1027/0
p23	3670	290/2946/3268	∞	68/0/3200	869	379/2889/0
p27	2182	216/1410/1645	∞	151/0/1494	285	410/1235/0
p31	1027	175/513/711	∞	2/0/709	106	109/602/0
psyco2	630	50/274/419	∞	15/0/404	6781	96/323/0
psyco3	627	50/275/421	∞	16/0/405	6667	97/324/0
psyco4	624	50/274/419	∞	17/0/402	6635	96/323/0
pals1	656	6/95/184	∞	30/0/154	∞	37/0/147
p28-l	2480	399/1425/1826	∞	242/0/1584	5981	502/1324/0

paths whose traversals are deemed to no longer be necessary due to similarity with already-traversed paths. The technique first employs *interpolation* to abstract the already-traversed paths in the SET such that more paths later traversed can be subsumed (pruned) in the SET. Our framework employs TRACER with a modified form of interpolation customized for this problem and scales to realistic programs.

III. DISCUSSION ON RESULTS

We used an Intel Core i7-6700 3.40 GHz Linux Box with 32GB RAM. The programs are from different benchmarks SV-COMP verification tasks [8], [9], RERS [10], Embedded Systems [11] and academic programs. We compared our method against a baseline method which used only a standard DSE system, KLEE [12] (timeout 3600 sec).

See Table I. Columns 1 and 2 show the benchmark name and size (LOC). The next column shows a triple: number of predicates (#P), number of atomic conditions (#AC) and total number of sequences (#S.). The rest of the columns present the result from the BASELINE method and ours. #T presents the analysis time, and a triple: #F, #I, #U where #F is the number of feasible sequences found within timeout, #I is the number of (provably) infeasible sequences, and #U denotes the remainder (denoting those sequences not determined within timeout).

The first group contains 5 programs where both our method and the BASELINE method fully traverse the SET. The only interesting observation is that our method is faster on 3 of 5 programs. For the other 2, the throughput speed of KLEE shines through because: (a) `tcas` has a small SET (it is loop free) and (b) `nsichneu` contains large number of infeasible paths. As a result, non-pruning DSE like KLEE with good path sensitivity has a chance to fully explore their SET rapidly.

The second group contains 6 programs where only our method has full traversal. This means that our method is *complete*, either producing a test input or proving no such input exists, for each sequence. Thus, the set of generated test cases for the first and second group is *optimal*. In contrast, the BASELINE approach is incomplete and hence it can only produce a conservative estimate of the test inputs.

The third group contains 5 programs. These are the largest amongst the benchmark programs. Neither our method nor BASELINE algorithm are able to fully traverse their SET within timeout. Here we do not know if our method is optimal. In a next experiment we extended the timeout to 7200 seconds for this group and reran both methods. This time our method, unlike BASELINE approach, is able to fully analyze 4 out of 5 programs. Overall, from our analysis of experiments, we can infer the importance of employing pruning DSE for finding more test cases and yielding to higher MC/DC score.

We note that finding test cases for sequences in general is as hard as the reachability problem. Although KLEE like DSE in general have shown good results in terms of code coverage but they might perform poorly in solving the reachability problem. For example, in our experiments both KLEE and our method were able to reach high instruction coverage for most programs (100% instruction coverage for all the programs from RERS [10]), however, our method has been able to find in average 3X more feasible sequences in group two programs.

REFERENCES

[1] K. Hayhurst *et al.*, “A practical tutorial on modified condition/decision coverage,” NASA Technical Memorandum, May 2001., Tech. Rep.
[2] J. Kauttio, “Mc/dc based test selection for dynamic symbolic execution,” G2 Pro gradu, diplomityö, 2013.
[3] M. W. Whalen *et al.*, “A flexible and non-intrusive approach for computing complex structural coverage metrics,” in *ICSE*, 2015.
[4] J. Jaffar *et al.*, “Tracer: A symbolic execution tool for verification,” in *CAV*, 2012.
[5] J. Jaffar, A. E. Santosa, and R. Voicu, “An interpolation method for clp traversal,” in *CP*. Springer, 2009, pp. 454–469.
[6] K. L. McMillan, “Lazy annotation for program testing and verification,” in *CAV*, 2010.
[7] ——, “Lazy annotation revisited,” in *CAV*, 2014.
[8] “SV-COMP Psyco Benchmarks ,” Dec. 2017. [Online]. Available: https://github.com/sosy-lab/sv-benchmarks/tree/master/c/psyco
[9] “SV-COMP Pals Benchmarks,” Nov. 2017. [Online]. Available: https://github.com/sosy-lab/sv-benchmarks/tree/master/c/seq-mthreaded
[10] “RERS Challenge,” 2018. [Online]. Available: http://rers-challenge.org/
[11] “Mälardalen WCET Research Group Benchmarks,” 2006. [Online]. Available: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[12] C. Cadar *et al.*, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.” in *OSDI*, vol. 8, 2008.