VeriCombTest: Automated Test Case Generation Technique Using a Combination of Verification and Combinatorial Testing

Sangharatna Godboley[a]
Department of Computer Science and Engineering,
National Institute of Technology Warangal, Telangana, India

Keywords: Verification, Combinatorial Testing, Test Cases, Mutation Analysis.

Abstract: We propose VeriCombTest which is the combination of Verification and Combinatorial Testing. We experimented with 38 C-Programs from The RERS challenge repository. Verification (CBMC) produced 940 test cases and Combinatorial Testing (PICT) populated a total of 42053 test cases. The good point is that for 40 programs, PICT consumed only 2.6 Minutes to populate the test inputs, however, CBMC which is a Static Symbolic Executor consumed 546.99 Minutes to generate the test inputs. We performed mutation analysis for this work. VeriCombTest has 355 extra killed mutants as compared to the baseline. VeriCombTest is a fully automated tool.

1 INTRODUCTION

Software verification and testing are two important techniques in Software Development Life Cycle (SDLC) (Mall, 2018; Mathur and Wong, 1993). Both have their individual advantages and disadvantages. For example, verifiers are usually faster as compared to testers. The automated test case generation process is now becoming an essential approach in the software testing phase.

Automated test case generation is still a challenging task. There are several test case generation techniques proposed in past including Fuzzing, Bounded Model Checking, Dynamic Symbolic Execution, and Combinatorial Testing, etc. These techniques have shown good performances in efficient ways to date. But, each individual technique has some issues associated and none of them claims to be 100% accurate. Hence, there is always a scope for improvement. For verification, we have considered the state-of-art tool CBMC (Bounded Model Checker for C) and for the Combinatorial Testing, we considered PICT (Pairwise Independent Combinatorial Testing). The idea is straightforward, we generate test inputs for a C-Program using CBMC and supply them into PICT to populate new test inputs. PICT is lightweight because it doesn't execute the program but rather applies pairwise or n-way techniques on given test inputs to generate new useful test inputs.

[a] https://orcid.org/0000-0002-6169-6334

In Bounded Model Checking (BMC)(Clarke et al., 2003; Armando et al., 2006), a Boolean formula is checked for satisfiability using SAT solver. If the Boolean formula is satisfiable, a counterexample is extracted from the output of the SAT. On the other hand, if the formula is not satisfiable, the program can be unwound more to determine if a longer counterexample exists.

Combinatorial Testing (CT) (Nie and Leung, 2011) can identify crashes or failures by interactions of various parameters in the program with a covering set of test cases generated by existing techniques. It has been an active domain of research and practice in the last 2 to 3 decades.

Motivation of this work is to make stronger test cases so that the quality of the application can be assured. This is one of the principles of Software Quality Assurance. As literature shows that no single technique or tool is sufficient to cover all the corner cases while testing. Every tool comes with pros and cons. So, there is a need of combining the tools and test the application. Such combinations are most of the time beneficial. There are works such as Veritesting (Avgerinos et al., 2014), VeriAbs (Afzal et al., 2019), and VeriFuzz (Basak Chowdhury et al., 2019) which are based on the combinations of techniques. As we know the Bounded Model Checking (BMC) concept is a static symbolic execution. So dynamic behaviour in BMC is not covered. But, BMC-based tools are very lightweight and most of the time assure quality results. But, due to the static behaviour, there is always a scope for exploring the uncovered code or bugs. On the other hand, combinatorial testing proved to be the most powerful testing technique. But, it cannot be even started without a set of initial test cases. Since combinatorial testing is a black-box testing technique there is no need of providing the source code of the application. So just test cases are there to make the pairing and produce new test cases. This is the main objective of combinatorial testing. Now, to solve the disadvantages of both techniques, combining them would be a good solution. We consider coverage and mutation analysis to show the benefits.

In this work, we are taking advantage of both verification and testing methods. From the domain of verification, we found that BMC is a prominent approach and *CBMC* is a state-of-the-art tool to use. On the other hand, combinatorial testing is becoming a reasonable and economical technique to be used to populate new test cases. *PICT* is a popular tool in combinatorial testing domain. Since, the hybridization of the techniques and tools becoming one of the practices, hence, we consider both *CBMC* and *PICT* tools to combine in our work. Our main objective and contribution to this work are to improve the test suite so that we get benefits in terms of both coverage and errors (mutants). Hence, *VeriCombTest* technique is proposed and implemented. In our work, we have chosen two modes and defined them below:

Definition 1.1 (Mode1)**.** This Mode1 deals with the process of computing coverage and mutation results in wrt. test cases generated from *CBMC*. In this paper, any metrics suffixed with “1” is associated with Mode1.

Definition 1.2 (Mode2)**.** This Mode2 deals with the process of computing coverage and mutation results in wrt. test cases generated from *CBMC* and *PICT*. In this paper, any metrics suffixed with “2” is associated with Mode2.

The rest of the paper is organised as below: The proposed approach is explained in Section 2. An example is explained in Section 3. Section 4 shows the detailed experimental study of the work. Finally, we conclude the paper with future work in Section 5.

2 PROPOSED APPROACH: VeriCombTest

In this section we discuss our proposed approach in detail. We will discuss the framework of our approach.

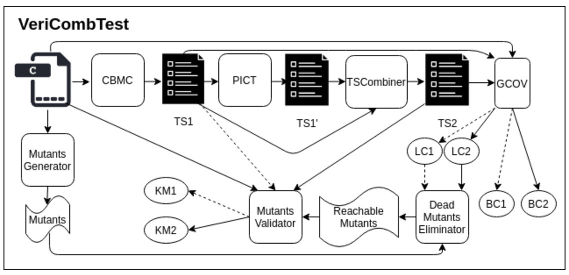

Figure 1: Framework for VeriCombTest.

Basically, *VeriCombTest* tool is the integration of seven components as shown in Fig. 1 and it is automatic. These seven components are a) CBMC, b) PICT, c) TSCombiner, d) GCOV, e) Mutants Generator, f) Dead Mutants Eliminator, and g) Mutants Validator. The main purpose of this tool is to escalate the test suite by quantity as well as quality. The quantity can be measured directly by counting test cases. However, to measure quality, coverage and mutation analysis are required to be performed. There is a reason to go for mutation analysis which is beyond coverage analysis. This we will explain in more detail in Section 4. For now, consider the coverage analysis and mutation analysis are the outputs of the automated *VeriCombTest* tool.

The flow in the framework starts by supplying a C-Program into *CBMC* as an input. The *CBMC* has two options to analyse a program. The first option is a bug finding and the second option is the coverage. We have used **coverage option** for our work. *CBMC* produces a detailed report from which we have extracted the test suite. In Fig. 1, it can be observed that *CBMC* produced *TS1*.

This *TS1* is supplied into *PICT*. The *PICT* tool has options for **n-way** testing. The basic one is **2-way** or **pairwise** testing. We have used this basic and default option in our work. We leave work on the higher order *n-way* testing for future analysis. The *PICT* tool produces combinatorial test cases. In Fig. 1, it can be observed that *PICT* produced *TS1’*.

Now, the *TSCombiner* component takes both *TS1* and *TS1’* and produces *TS2*. Please note that *TS1’* is generated from *PICT* tool, which is only mutating test values and doesn’t bother with logic in the program, consider it as a black-box tool. So, it is quite possible that *TS1’* alone could lead to low coverage. Also, as stated earlier *CBMC* is a verifier though it generates quality test cases, due to complex predicates and loop structures, there is a scope for improvement in the quality of test cases. Now, to utilise the advantages of *CBMC* and *PICT* mechanisms we combine both the test suites *TS1* and *TS1’*, so that at least we will achieve equal or more coverage. In Fig. 1, it can be observed that *TSCombiner* produced *TS2*.

Next, C-Program, *TS1* and *TS2* supplied into *GCOV* component. *GCOV* is a well-known test case validator that comes with GCC compiler. We replayed *TS1* and *TS2* with C-Program and produced Line Coverage for Mode 1 (LC1), Line Coverage for Mode 2 (LC2), Branch Coverage for Mode 1 (BC1), and Branch Coverage for Mode 2 BC2 as shown in Fig. 1.

Briefly speaking, Line Coverage and Branch Coverage information was not enough to show the quality of test cases generated by *VeriCombTest* tool (as mentioned earlier too the detailed reason will be explained in the experimental section). Hence, Mutation Analyser which is developed by us has been used. *Mutation Analyser* component comprises of three components, *Mutants Generator*, *Dead Mutants Eliminator*, and *Mutants Validator*. From Fig. 1, it can be observed that C-Program is supplied into *Mutants generator* to produce *Mutants* as output. Please note that we have taken five types of Mutants (M) in our work. These are 1. LOF (Logical Operator Faults), 2. AOF (Arithmetic Operator Faults), 3. ROF (Relational Operator Faults), 4. LNF (Literal Negation Faults) and 5. PNF (Predicate Negation Faults).

Now, after generating all mutants from the C-Program, it is quite possible that most of them might be unreachable. The reason is the static mechanism of mutant generation. We have implemented *Dead Mutants Eliminator* which takes all mutants and Line Coverage information (LC1 and LC2) as inputs and removes the **Dead Mutants** (in other words unreachable mutants based on line coverage). So, the remaining mutants are called *Reachable Mutants* which are the outputs of *Dead Mutants Eliminator* component.

Finally, the last component i.e. *Mutants Validator* accepts *C-Program*, *TS1*, *TS2*, and *Reachable Mutants* as inputs. This validator replays test cases with C-Program and all mutants and checks their outputs. If the outputs of a C-Program and a mutant are not the same then the mutant is called as ***Killed Mutant (KM)***, otherwise ***Alive Mutants***. This component produces Killed Mutant for Mode 1 (KM1) and Killed Mutant for Mode 2 (KM2) as outputs as shown in Fig.1.

3 WORKING EXAMPLE

In this section, we explain the flow with an example that we experimented.

We have selected *test27-B2.c* program from our experimented benchmark set. This program is of **2934** LOCs, **160** functions, **232** total predicates, and **1293** atomic conditions. It has a for loop with two iterations. Since the program is too big therefore *main()* has been shown in Listing 1. This program has a variable "*symb*" that is not concrete variable rather required test input value(s) to achieve code coverage. Also, it is to be noted that the "*symb*" variable is inside "*for*" loop, it means for each iteration the variable gets reset and takes new value. It considered as two different variables for second iteration.

The *test27-B2.c*[1] program analysed using *CBMC* with coverage mode and test suite have been generated as shown in Listing 2. Each line in Listing 2 is a test case. So, there is a total **5** test cases in *TS1*. Now, *TS1* required some formatting as per the requirement of *PICT* tool. The format PICT accepts the test cases is mentioned in Listing 3. It is to be noted that *CBMC*'s report considered "*symb*" variable two different times but the variable names mentioned was "*symb*" only. But, *PICT* required distinct variables in the test suite which need to be supplied. Hence, we renamed the variables for the purpose to utilise *PICT*. The renamed variable are "*_1symb*" and "*_2symb*" as shown in Listing 3.

Listing 1: main() of C-Program test27-B2.c.

```
int BOUND = 2;
.....................
calculate_output(int input) {
  THE PROGRAM IS TOO BIG
 }
.....................
int main() {
int symb;
printf("POINT: 467\n");
for (int FLAG=0;FLAG<BOUND;FLAG++){
printf("POINT: 468\n");
symb = nondet_int();
__CPROVER_input("symb",symb);
printf("POINT: 469\n");
if((symb != 8) && (symb != 9) && (symb != 7) &&
      (symb != 5) && (symb != 2) && (symb != 4) &&
      (symb != 10) && (symb != 6) && (symb != 3) &&
      (symb != 1)){
printf("POINT: 470\n");
 return -2;}
 calculate_output(symb);//Too Big function
}}
```

Listing 2: Test Suite (TS1) generated from CBMC.

```
TC1:   symb=8,          symb=8
TC2:   symb=4,          symb=4
TC3:   symb=134217732
TC4:   symb=8,          symb=6
TC5:   symb=8,          symb=2
```

Listing 3: Test Suite (TS_{PICT}) generated from CBMC and formatted as per PICT requirements.

```
_1symb:8, 4, 134217732, 8, 8
_2symb:8, 4, 6, 2
```

[1]Full Program is provided with Supplementary material

Table 1: Test Suite (TS1') generated from PICT.

TC	_1symb	_2symb	**TC**	_1symb	_2symb	**TC**	_1symb	_2symb
1	8	8	**8**	8	2	**15**	8	4
2	4	2	**9**	4	4	**16**	8	6
3	134217732	2	**10**	8	6	**17**	8	2
4	134217732	4	**11**	8	8	**18**	8	8
5	134217732	8	**12**	8	4	**19**	4	8
6	8	6	**13**	4	6	**20**	8	4
7	134217732	6	**14**	8	2	-	-	-

Next, TS$_{PICT}$ is supplied into *PICT* and generate *TS1'* as shown in Tabe 1. This test suite has **20** test cases generated from the combinatorial testing technique. Factually, *PICT* generates non-redundant test cases. But, as we can observe from Table 1 we do get some duplicate test cases. This is against the principles of *PICT* tool. We have identified the problems and we explain that the test values of "*_1symb*" variable in TS$_{PICT}$ has "**8**" which is repeated **3** times. But, if we see the context of these repetitions then we will observe that the combinations of both variables "*_1symb*" and "*_2symb*" achieve higher coverage which is reasonable, and they are absolutely not redundant for *CBMC*. But, this is also true that *PICT* is a black-box tool which doesn't bother for the logic of the program here. Hence, it is leading to duplication of test input values in test suite. In this paper, we haven't removed such test cases upfront by considering that *PICT* is super fast and duplication may not effect the execution time much. But, yes we are pretty aware that mutation time might be definitely affected by this issue. We leave the study of Optimization of test cases for the future work. Finally, we merge two test suites *TS1* and *TS1'* and form a new test suite *TS2*. For the *test27-B2.c* program *TS2* has a total of **25** test cases (to save space we have not shown a separate Listing for *TS2*).

To know whether the combination of test cases is actually useful or not, we used *GCOV* to get the coverage results. *GCOV* results Line coverage and Branch Coverage reports. The coverage reports by *GCOV* on *TS1* and *TS2* are shown in Listings 4 and 5. We expect that the coverage should be enhanced since the test cases have been increased. But, unfortunately, both the modes have the same Line coverage for the program *test27-B2.c* i.e. **18.74%**. This is reasonable because the *CBMC* is a well-established verifier which is popularly known for covering lines (good reachability in terms of bugs). This is not surprising that both modes are having same Line coverage. Secondly, we expect that the Branch Coverage will be enhanced. From Listings 4 and 5, it can be observed that Mode1 has **23.40%** Branch Coverage, however Mode2 has **23.90%**. So, there is little improvement of **0.50%**. This result shows that *CBMC* has not 100% optimal Branch Coverage. Hence, there is a scope for some improvements in the results.

Listing 4: GCOV report using Test Suite for Mode 1 (TS1).

```
Lines executed:18.74% of 2102
Branches executed:23.40% of 2389
Taken at least once:13.94% of 2389
Calls executed:31.76% of 636
```

Listing 5: GCOV report using Test Suite for Mode 2 TS2.

```
Lines executed:18.74% of 2102
Branches executed:23.90% of 2389
Taken at least once:14.23% of 2389
Calls executed:31.76% of 636
```

Listing 6: Mutants Generated.

```
Logical Operator Faults: 1175
Arithmetic Operator Faults: 7312
Relational Operator Faults: 6465
Literal Negation Faults: 1391
Predicate Negation Faults: 235
```

Listing 7: Mutants Statistics for Mode1.

```
============Mutation Score Report============
Total number of Alive Mutants =: 2934
Total number of Killed Mutants =: 1171
Total number of Reached Mutants =: 4105
Total number of Dead Mutants =: 12473
Total number of Total Mutants =: 16578
Mutation Score (Killed/Reached) =: 28.53%
============Report-Finish====================
```

Listing 8: Mutants Statistics for Mode2.

```
============Mutation Score Report============
Total number of Alive Mutants =: 2928
Total number of Killed Mutants =: 1177
Total number of Reached Mutants =: 4105
Total number of Dead Mutants =: 12473
Total number of Total Mutants =: 16578
Mutation Score (Killed/Reached) =: 28.67%
============Report-Finish====================
```

Listing 9: Time Statistics.

```
Time Analysis CBMC Report: 116.10 seconds
Time Analysis PICT Report: 0.002 seconds
Time Analysis GCOV Report: 0.676 seconds
Time Analysis MA Mode1 Report: 1060.71 seconds
Time Analysis MA Mode2 Report: 1295.83 seconds
```

Now, Listings 7 and 8 show the mutants statistics for Mode1 and Mode2 respectively. From both reports, we can observe that the total of mutants created was **16578**. Since we know these mutants are statically generated. So, we use line coverage information from *GCOV* to detect the dead mutants and eliminate them. We have observed that **12473** mutants were dead (unreachable), which means **4105** mutants were only reachable and useful to run. In Mode1, a total number of **1171** mutants got killed and **2934** mutants were alive as shown in Listing 7. However, in Mode2, a total number of **1177** mutants got killed and **2928** mutants were alive. So, a total of **6** extra mutants got killed in Mode2 in contrast to Mode1. The Reachable Mutation scores can be observed for Mode1 and Mode2 as **28.53%** and **28.67%** respectively. This is the strength of our proposed work. There is an improvement of **0.14%** for Mode2 as compared to Mode1.

Next, we also explain the mutation analysis for *test27-B2.c* program. First, we statically generate all possible mutants for *test27-B2.c* program. We have considered *5* types of faults. The created mutants with each category are mentioned in Listing 6.

Lastly, we report the time analysis for *test27-B2.c* program. At each main step, we compute execution times. We consider *CBMC*, *PICT*, *GCOV*, *MA Mode1* and *MA Mode2* steps as important to capture execution times. For this program, *CBMC* consumed **116** sec, *PICT* consumed **0.002** sec, and *GCOV* consumed **0.67** sec (it covers both modes). Mutation Analysis time for Mode1 has **1060** sec and for Mode2 it is **1295** sec. We can observe that *PICT*'s execution time is very less, almost negligible. But, the mutation analysis time for Mode2 which is actually having test cases generated from both *CBMC* and *PICT* tools. The extra mutation analysis time Mode2 required was **235.12** sec as compared to Mode1. This is the weakness of our work.

4 EXPERIMENTAL STUDY

In this section, we discuss the setup and benchmarks tested, and discuss on results.

4.1 The Set Up

We used an Intel Core i7-9700 CPU @ 3.00GHz × 8 Linux box (64-bit Ubuntu 16.04) with 64 GB RAM. All the input programs considered for our study are written in ANSI-C format. For result comparison, we consider *CBMC* as our baseline because it is a state-of-the-art tool. The programs and all the raw experimental details are provided in the supplementary artefacts (VeriCombTest-Artifacts, 2021).

4.2 Benchmarks Tested

Here, we describe the programs experimented with. In total, we have tested 38[2] programs taken from various open source repositories. In our study, we have considered programs from RERS(RERS, 2018). These programs are from the small and moderate size

Table 2: Test Cases and Coverage Results.

Sl	Program	TC1	TC1'	TC2	LC	BC1	BC2	BI
1	test21-B2	13	156	169	26.95	28.43	28.43	0.00
2	test21-B3	24	619	643	53.67	56.35	56.35	0.00
3	test21-B4	64	4324	4388	100.00	100.00	100.00	0.00
4	test21-B5	49	2716	2765	100.00	100.00	100.00	0.00
5	test22-B2	6	30	36	9.78	12.83	12.83	0.00
6	test22-B3	13	159	172	15.05	22.03	22.03	0.00
7	test22-B4	24	596	620	23.58	31.32	31.40	0.08
8	test22-B5	44	2195	2239	40.03	51.57	51.57	0.00
9	test23-B2	8	56	64	6.46	7.89	7.89	0.00
10	test23-B3	19	365	384	11.99	15.10	15.10	0.00
11	test29-B2	10	100	110	28.54	32.80	32.80	0.00
12	test29-B3	23	556	579	47.57	55.03	55.03	0.00
13	test29-B4	38	1508	1546	62.96	68.25	68.25	0.00
14	test29-B5	40	1773	1813	73.08	76.19	76.19	0.00
15	test26-B2	9	81	90	26.23	33.44	33.44	0.00
16	test26-B3	17	314	331	38.17	49.27	49.27	0.00
17	test26-B4	28	883	911	58.52	69.68	69.68	0.00
18	test26-B5	32	1209	1241	73.00	84.33	84.33	0.00
19	test31-B2	4	12	16	7.00	8.53	8.53	0.00
20	test31-B3	6	30	36	9.45	11.25	11.25	0.00
21	test31-B4	7	49	56	12.77	15.18	15.18	0.00
22	test31-B5	10	110	120	16.71	19.44	19.44	0.00
23	test32-B2	17	272	289	27.41	27.32	27.43	0.11
24	test32-B3	47	2265	2312	46.66	46.54	46.54	0.00
25	test32-B4	72	5471	5543	60.82	59.79	59.79	0.00
26	test32-B5	87	8282	8369	66.76	65.30	65.30	0.00
27	test24-B2	5	25	30	14.34	16.02	16.02	0.00
28	test24-B3	9	74	83	15.34	16.80	16.80	0.00
29	test24-B4	11	143	154	18.03	19.74	19.74	0.00
30	test24-B5	22	556	578	22.97	24.70	24.75	0.05
31	test28-B2	7	49	56	15.19	17.54	17.54	0.00
32	test28-B3	13	164	177	18.35	20.31	20.31	0.00
33	test28-B4	28	854	882	23.27	24.55	24.55	0.00
34	test28-B5	40	1829	1869	31.25	31.61	31.69	0.08
35	test27-B2	5	20	25	18.74	23.40	23.90	0.50
36	test27-B3	10	95	105	24.83	32.31	33.24	0.93
37	test27-B4	23	563	586	36.63	46.30	47.47	1.17
38	test27-B5	56	3550	3606	63.84	69.28	71.20	1.92

group and easy to hard categories. Note that due to the complex loop patterns, the number of Killed mutants can change at different bounds. Programs are originally unbounded. We instead set reasonable bounds as *2*, *3*, *4*, and *5*. Also in the programs identified by a suffix with "-B*", "*" indicates the bound used in the programs. The unbounded programs mean the programs have infinite loop bound without any exit criterion. Further, it is to be noted that, the size of coverable parts in the program will vary from one loop bound to another loop bound. Because for a higher loop bound more parts of the program will be coverable. Since the reachable and killable mutants are getting increased hence the mutation score will be higher. It is to be noted that the total number of mutants for lower or higher loop bounds is equal because it is computed statically.

4.3 Coverage Analysis

In this section, we discuss on the coverage results generated for test cases.

Table 2 shows test cases and coverage results. Columns 1 and 2 of Table 2 show the number and names of programs. Columns 3 to 5 show the test cases. Column 3 (TC1) is the test cases generated

[2]Actually we targeted to run 40 programs, but CBMC runs for *test23-B4* and *test23-B5* were consuming almost 30+ hrs so we discontinued testing them.

from *CBMC* and Column 4 (TC1') is the test cases generated from *PICT*. For 38 programs, the aggregate of the test cases for *CBMC* and *PICT* are **940** and **42053** respectively. We can observe that test cases generated by *PICT* (TC1') have extra test cases almost **44**× as compared to CBMC. Now, Column 5 (TC2) can be generated by using the equation "$TC2 = TC1 + TC1'$". It is to be noted that TC1 and TC2 show the test cases for Modes 1 and 2 respectively.

Next, TC1 replayed with C-Program to generate Line and Branch Coverage. Column 6 of Table 2 shows the LC (Line Coverage). It is to be noted that $LC = LC1 = LC2$, which means the Line coverage for both modes 1 and 2 are equal hence, we reported in a single column only. The reason is that the *CBMC* has a good statement reachability feature. Columns 7 and 8 show the Branch Coverage information. Column 7 (BC1) shows the Branch Coverage for Mode 1 and Column 8 (BC2) shows the Branch Coverage for Mode 2. Lastly, Column 9 shows the Branch Coverage Improvement (BI) which can be computed using the equation "$BI = BC2 - BC1$". From Table 2, it can be observed that for **8** (**21.05%** out of 38) programs (highlighted with green colours in Table 2), Mode 2 i.e. *VeriCombTest* has higher Branch Coverage as compared to *CBMC* i.e. Mode 1. Though the improvements in Branch Coverage for the programs are not much, the achieved results are sufficient to prove that the baseline *CBMC* alone has the scope of improvement by introducing the extra test cases.

Giving a remark, in this section before the experimentation our main expectation was to observe improvement in Line Coverage. But, after the observation, we got to know that Line Coverages for both modes are equal. Hence, we decided to compute Branch Coverage in a hope that we might get an improvement. After the experimentation, we have observed that we have some improvements which ignite our further analysis i.e. Mutation analysis. As we know that Branch Coverage report will just focus on the *true* and *false* branches of conditional statements such as "*if-statement*" which might not be enough to observe the actual benefits of *VeriCombTest* technique. Therefore next we explain mutation analysis for the programs.

4.4 Mutation Analysis

In this section we discuss Mutation Analysis in detail. Table 3 shows the results of Mutation Analysis for the considered benchmark programs. Column 1 of Table 3 shows the programs. Columns 2, 3, and 4 show the total number of Mutants (TM), total number of Dead Mutants (DM), and total number of Reachable Mutants (RM) respectively. RM can be computed using the equation "$RM = TM - DM$". Columns 5 and 6 show the total number of killed mutants (KM1 for Mode1 and KM2 for Mode2) after processing reachable mutants. For 38 programs there are a total **305474** mutants (TM). Out of which **198593** mutants are Dead Mutants (DM) and **106881** are Reachable Mutants (RM). It means **65%** of total mutants are dead (or unreachable) and it does not require running them using the test cases for both modes because they will be reported as **Alive Mutants** which means no test case can detect that mutant. But, traditionally the mutation scores were able to compute without the information of dead mutants[3]. The total

Table 3: Mutation Analysis.

Sl	TM	DM	RM	KM1	KM2	T1%	T2%	R1%	R2%	TI	RI	D1	D2
1	1302	950	352	287	288	22.04	22.12	81.53	81.82	0.08	0.28	59.49	59.70
2	1302	598	704	550	558	42.24	42.86	78.13	79.26	0.61	1.14	35.88	36.40
3	1302	0	1302	1033	1046	79.34	80.34	79.34	80.34	1.00	1.00	0.00	0.00
4	1302	0	1302	1025	1051	78.73	80.72	78.73	80.72	2.00	2.00	0.00	0.00
5	12858	11302	1556	676	677	5.26	5.27	43.44	43.51	0.01	0.06	38.19	38.24
6	12858	10225	2633	1205	1207	9.37	9.39	45.77	45.84	0.02	0.08	36.39	36.45
7	12858	9014	3844	2104	2114	16.36	16.44	54.73	54.99	0.08	0.26	38.37	38.55
8	12858	6494	6364	3904	3918	30.36	30.47	61.35	61.57	0.11	0.22	30.98	31.09
9	31455	28947	2508	1280	1280	4.07	4.07	51.04	51.04	0.00	0.00	46.97	46.97
10	31455	26641	4814	2842	2845	9.04	9.04	59.04	59.10	0.01	0.06	50.00	50.05
11	1314	904	410	294	296	22.37	22.53	71.71	72.20	0.15	0.49	49.33	49.67
12	1314	621	693	535	547	40.72	41.63	77.20	78.93	0.91	1.73	36.49	37.30
13	1314	437	877	703	710	53.50	54.03	80.16	80.96	0.53	0.80	26.66	26.92
14	1314	322	992	799	814	60.81	61.95	80.54	82.06	1.14	1.51	19.74	20.11
15	3219	2129	1090	544	546	16.90	16.96	49.91	50.09	0.06	0.18	33.01	33.13
16	3219	1739	1480	884	889	27.46	27.62	59.73	60.07	0.16	0.34	32.27	32.45
17	3219	1078	2141	1405	1418	43.65	44.05	65.62	66.23	0.40	0.61	21.98	22.18
18	3219	646	2573	1797	1816	55.82	56.42	69.84	70.58	0.59	0.74	14.02	14.16
19	3182	2922	260	181	181	5.69	5.69	69.62	69.62	0.00	0.00	63.93	63.93
20	3182	2845	337	242	242	7.61	7.61	71.81	71.81	0.00	0.00	64.20	64.20
21	3182	2713	469	274	275	8.61	8.64	58.42	58.64	0.03	0.21	49.81	49.99
22	3182	2582	600	384	385	12.07	12.10	64.00	64.17	0.03	0.17	51.93	52.07
23	5883	3087	2796	948	956	16.11	16.25	33.91	34.19	0.14	0.29	17.79	17.94
24	5883	2085	3798	1840	1850	31.28	31.45	48.45	48.71	0.17	0.26	17.17	17.26
25	5883	1362	4521	2500	2514	42.50	42.73	55.30	55.61	0.24	0.31	12.80	12.87
26	5883	1074	4809	2785	2801	47.34	47.61	57.91	58.24	0.27	0.33	10.57	10.63
27	7827	5391	2436	407	408	5.20	5.21	16.71	16.75	0.01	0.04	11.51	11.54
28	7827	5322	2505	533	534	6.81	6.82	21.28	21.32	0.01	0.04	14.47	14.49
29	7827	5121	2706	767	770	9.80	9.84	28.34	28.46	0.04	0.11	18.55	18.62
30	7827	4752	3075	1125	1140	14.37	14.56	36.59	37.07	0.19	0.49	22.21	22.51
31	8478	5835	2643	717	717	8.46	8.46	27.13	27.13	0.00	0.00	18.67	18.67
32	8478	5595	2883	1008	1014	11.89	11.96	34.96	35.17	0.07	0.21	23.07	23.21
33	8478	5229	3249	1379	1388	16.27	16.37	42.44	42.72	0.11	0.28	26.18	26.35
34	8478	4611	3867	1909	1948	22.52	22.98	49.37	50.37	0.46	1.01	26.85	27.40
35	16578	12473	4105	1171	1177	7.06	7.10	28.53	28.67	0.04	0.15	21.46	21.57
36	16578	11155	5423	2149	2169	12.96	13.08	39.63	40.00	0.12	0.37	26.66	26.91
37	16578	8650	7928	4187	4203	25.26	25.35	52.81	53.01	0.10	0.20	27.56	27.66
38	16578	3742	12836	8803	8839	53.10	53.32	68.58	68.86	0.22	0.28	15.48	15.54

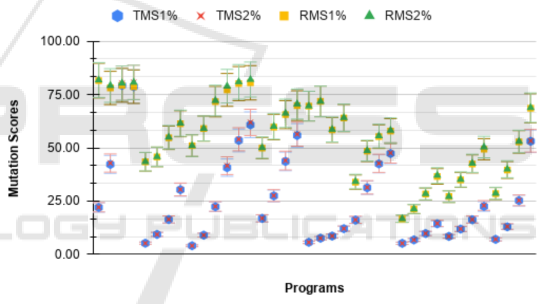

Figure 2: Improvement Analysis.

[3]To avoid the confusion, it is to be noted that "Dead Mutants" and "Killed Mutants" are two different elements. The "Dead Mutants" is nothing but the mutation or change made in the code which is actually dead or unreachable. However, "Killed Mutants" shows the detection of the mutants or change in the reachable code using test cases.

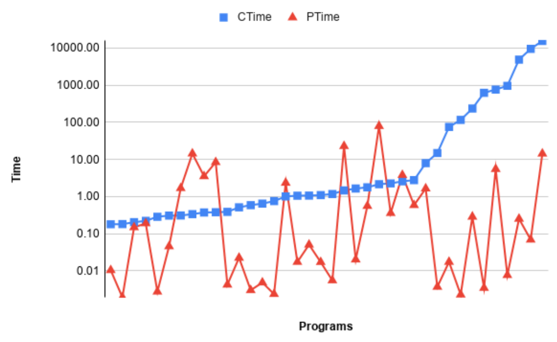

Figure 3: Time consumption by CBMC and PICT.

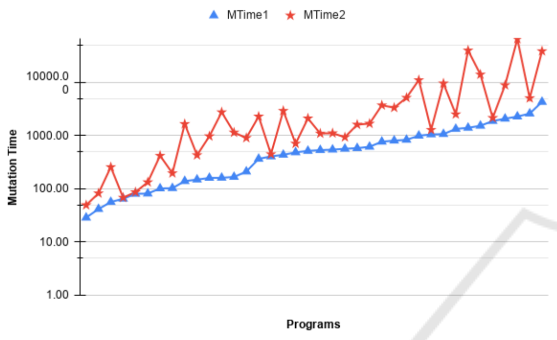

Figure 4: Time consumption in mutation analysis by Mode 1 and Mode 2.

killed mutants for Mode 1 is **55176** and for Mode 2 is **55531**. It can be observed that Mode 2 has killed **355** extra mutants as compared to Mode 1, this is the key result and strength that we were expecting from *VeriCombTest* approach.

Next, Columns 7 to 10 show the mutation scores. Column 7 shows the Traditional Mutation Score for Mode 1 (T1%) and can be computed using equation "$T1\% = \frac{KM1}{TM} X100$". Same way, Column 8 shows the Traditional Mutation Score for Mode 2 (T2%) and can be computed using equation "$T2\% = \frac{KM2}{TM} X100$". Column 9 shows the Reachable Mutation Score for Mode 1(R1%) and can be computed using equation "$R1\% = \frac{KM1}{RM} X100$". Same way, Column 10 shows the Reachable Mutation Score for Mode 2 (R2%) and can be computed using equation "$R2\% = \frac{KM2}{RM} X100$".

Columns 11 and 12 show the Traditional Mutation Improved (TI) and Reachable Mutation Improved (RI) results respectively. TI can be computed using equation "$TI = T2\% - T1\%$" and RI can be computed using equation "$RI = R2\% - R1\%$". From Columns 11 and 12 of Table 3, we can observe that in total **34** (**89.47%** of 38) programs have improvements (highlighted with green colours). The four programs *test23-B2*, *test31-B2*, *test31-B3*, and *test28-B2* have no improvements in both TI and RI. We can observe the programs, the loop bound inside the programs is 2 or 3. It means the structure of the program is less and a huge part is uncovered. As soon as the loop iterations get increased so the uncovered parts become coverable. On an average of 38 programs, the value for TI is **0.26%** and RI is **0.42%**. Fig. 2 shows the Improvement Analysis. The x-axis shows the programs whereas the y-axis shows the mutation score. We can observe that blue circles (T1%) and red crosses (T2%) are mostly below as compared to yellow squares (R1%) and green triangles (R2%). The *error* differences between blue circles (T1%) and red crosses (T2%) can be observed for 34 programs. Same way, it can be observed for yellow squares (R1%) and green triangles (R2%) for 34 programs.

A side work is done on mutation analysis to check that how much improvement has been achieved from the traditional to the proposed technique. Columns 13 and 14 of Table 3 show the Reachable Traditional Difference (D1 i.e. "$D1 = R1\% - T1\%$" for Mode1 and RTD2 i.e. "$D2 = R2\% - T2\%$" for Mode2).

For **36** (**94.73%** of 38) programs the improvement can be observed in D1 and D2. For two programs *test21-B4*, and *test21-B5* have no improvements in D1 and D2 (highlighted with orange colour). This is because, for these two programs, the total number of dead mutants is 0. On an average of 38 programs, the value of D1 is **29.22%** and D2 is **29.38%**. It shows that even though it is a traditional or newly proposed technique the elimination of Dead Mutants is important which can save a huge time of in mutation analysis. Finally, in this mutation analysis, we have observed that the *VeriCombTest* technique is beneficial, and we have provided the evidence with all the results.

4.5 Time Analysis

In this section, we discuss the time analysis. Except for a few programs mostly CBMC takes less execution time to analyse the programs. This is due to the abstract interpretation in CBMC. Also, the program structure is easy to verify by CBMC. Next, PTime is computed independently of the program. PICT requires a set of test cases and it produces another set of test cases after using a combinatorial testing approach. Therefore the execution time by PICT is less. In a comparison of Time for PICT (PTime) with Time for CBMC (CTime), it has been observed that **31** (**81.57%** out of 38) programs have less time-consuming as compared to CTime. For the rest 7 programs, it has been observed that the *CBMC* generated more test cases in less time and since the total number of test cases is more PICT consumed more time. The time comparison graph is shown in Fig. 3. The x-axis shows the programs and the y-axis shows the time in seconds (logarithmic format). We can observe the red line (PTime) is most of the time below the blue line (CTime). This shows that PTime is very less and hence *PICT* is beneficial to use in *VeriCombTest* framework. Time for GCoV (GTime) is the time consumed by the GCOV tool for both modes. The Aggregate of 38 programs, CTime, PTime, and GTime are **546.99** min, **2.67** min, **5.69** min respectively. Fig. 4 shows the graph for time consumption in mutation analysis by Mode1 and Mode2. The x-axis shows the programs and the y-axis shows the time in seconds (logarithmic format).

From the graph, it is very clear that the blue line is mostly below the red line. For very few programs the time difference is closer. In the Aggregate of 38 programs, the value of Mutation Analysis Time for Mode 1 (MTime1) is **496.07** min and Mutation Analysis Time for Mode 2 (MTime2) is **3960.17** min. The *VeriCombtest* i.e. Mode2 is clearly worse in mutation analysis time by **7.98**× as compared to baseline CBMC i.e. Mode1. Since the *PICT* produced more number of test cases that makes mutation time analysis worse. Improvement of the mutation time is the out of the scope of this paper. From this work, we would like to highlight that there is a scope for improvement for a verifier i.e. CBMC wrt. test cases.

5 CONCLUSION

From the literature survey we can observe that the works such as Veritesting (Avgerinos et al., 2014), Veriabs (Afzal et al., 2019; Darke et al., 2018), and VeriFuzz (Basak Chowdhury et al., 2019) used the common mechanism of combining different tools and techniques. Motivating by these works, we combine two techniques viz. Verification and Combinatorial Testing. For verification, we have considered *CBMC* and for Combinatorial Testing we considered PICT. We experimented with 38 C-Programs from The RERS challenge repository. For 38 programs PICT only consumed **2.6** Minutes to populate the test inputs, however, *CBMC* which is a Static Symbolic Executor consumed **546.99** Minutes to generate the test input. The *VeriCombTest* (Mode2) has **355** extra killed mutants as compared to baseline CBMC (Mode1). Finally, as discussed in coverage analysis section *VeriCombTest* has **21.05%** programs to achieve higher Branch Coverage and in mutation analysis *VeriCombTest* has **94.73%** programs to achieve higher mutation score, these show the strength of our proposed work.

REFERENCES

Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., and Venkatesh, R. (2019). Veriabs : Verification by abstraction and test generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1138–1141.

Armando, A., Mantovani, J., and Platania, L. (2006). Bounded model checking of software using SMT solvers instead of SAT solvers. In *13th SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 146–162. Springer.

Avgerinos, T., Rebert, A., Cha, S. K., and Brumley, D. (2014). Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1083–1094, New York, NY, USA. Association for Computing Machinery.

Basak Chowdhury, A., Medicherla, R. K., and R, V. (2019). Verifuzz: Program aware fuzzing. In Beyer, D., Huisman, M., Kordon, F., and Steffen, B., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 244–249, Cham. Springer International Publishing.

Clarke, E., Kroening, D., and Yorav, K. (2003). Behavioral consistency of C and Verilog programs using bounded model checking. In *40th DAC*, pages 368–371. ACM Press.

Darke, P., Prabhu, S., Chimdyalwar, B., Chauhan, A., Kumar, S., Basakchowdhury, A., Venkatesh, R., Datar, A., and Medicherla, R. K. (2018). Veriabs: Verification by abstraction and test generation. In Beyer, D. and Huisman, M., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 457–462, Cham. Springer International Publishing.

Mall, R. (2018). *Fundamentals of software engineering*. PHI Learning Pvt. Ltd.

Mathur, A. P. and Wong, W. E. (1993). Comparing the fault detection e ectiveness of mutation and data flow testing: An empirical study.

Nie, C. and Leung, H. (2011). A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2).

RERS (2018). RERS:. http://rers-challenge.org/.

VeriCombTest-Artifacts (2021). Raw experimnetal data. https://figshare.com/s/0d412445b4411208ae68.