Toward Optimal MC/DC Test Case Generation

Sangharatna GODBOLEY†, Joxan JAFFAR*, Rasool MAGHAREH*, Arpita DUTTA*

*National University of Singapore, Singapore
  {joxan,arpita}@comp.nus.edu.sg
*Huawei Canada Research Centre, Canada
  rasool.maghareh@huawei.com
†National Institute of Technology Warangal, India
  sanghu@nitw.ac.in

KLEE Workshop - September 2022
Accepted Contribution: Technical Track


Accepted Contribution: Poster Track


Artifact Available

Badges obtained: Available, Functional, and Reusable

Website: https://tracer-x.github.io/
Github: https://github.com/tracer-x/
What is MC/DC?

Modified Condition/Decision Coverage (MC/DC)
MC/DC is the second strongest coverage criterion for unit testing. It requires linear number of test cases wrt. the number of atomic conditions present in the program. MC/DC requires all the following requirements [1]:

- Each entry and exit point should get invoked.
- Each predicate takes both possible truth values.
- Each atomic condition (AC) in a predicate takes both possible truth values.
- Each AC in a predicate shown as independent.

Why MC/DC?
According to RTCA standard of DO-178B/C[1], it is mandatory to achieve MC/DC for Level A certificate of safety critical application.
1. Introduction
2. Survey
3. Proposed Idea
4. Experimental Evaluation
5. Conclusion
Figure: Which Domain?

Figure: Which automatic tools?
Figure: Automatic vs Manual!

Figure: Which strategy?
Outline

1 Introduction
2 Survey
3 Proposed Idea
4 Experimental Evaluation
5 Conclusion
Proposed Idea

Figure: The Overall Architecture of our Framework
An Example Predicate: if ((b<0 || c<0) && d < 0 ) { ... }

SG generates five short-circuited sequences\{S_1 : 101\}, \{S_2 : 102\}, \{S_3 : 211\}, \{S_4 : 212\}, and \{S_5 : 220\} where 1, 2, 0 represent True, False, and Don’t Care for the atomic conditions.

Table: Short-circuit Truth Table for Predicate

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td>T</td>
<td>X</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>S_2</td>
<td>T</td>
<td>X</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>S_3</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>S_4</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>S_5</td>
<td>F</td>
<td>F</td>
<td>X</td>
<td>F</td>
</tr>
</tbody>
</table>
Figure: Annotated CFG for the Predicate

### Table: Paths from sequences

<table>
<thead>
<tr>
<th>Id</th>
<th>Seq</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>101</td>
<td>1 → 3 → T</td>
</tr>
<tr>
<td>S₂</td>
<td>102</td>
<td>1 → 3 → F</td>
</tr>
<tr>
<td>S₃</td>
<td>211</td>
<td>1 → 2 → 3 → T</td>
</tr>
<tr>
<td>S₄</td>
<td>212</td>
<td>1 → 2 → 3 → F</td>
</tr>
<tr>
<td>S₅</td>
<td>220</td>
<td>1 → 2 → F</td>
</tr>
</tbody>
</table>
Algorithm 1 Test Case Generation Using DSE

**Input:** Upd_LLVM_IR,

**Output:** Test_Suite

1. Test_Suite $\leftarrow \emptyset$
2. errorPaths $\leftarrow$ Run_DSE (Upd_LLVM_IR)
3. for each errorPath in errorPaths do
4.    Input_Values $\leftarrow$ extractInputValues(errorPath)
5.    Test_Suite $\leftarrow$ Test_Suite + Input_Values
6. end for
Figure: Exploration of Symbolic Execution Tree in Non-pruning DSE vs. Pruning DSE
Consider the program:

```c
x = 0;
if (b1) x += 12;
if (b2) x += 15;
assert(x != 28);
```

**Figure:** SET of the program
Algorithm 2 Custom Interpolation

1: function PRE(Annotation , ChildInt)
2:     if BASE_BUG THEN RETURN \((\kappa \notin SeqVals - \{Val_\kappa\})\)
3:     END IF
4:     if BASE_NO_BUG THEN RETURN \((\kappa \notin SeqVals)\)
5:     END IF
6:     CHILD\_INT \equiv (\kappa \notin Set)
7:     ParentSet \leftarrow \{\}
8:     FOR EACH \(s\) IN Set DO
9:         ParentSet \leftarrow ParentSet + PRE\_COND(s , Annotation)
10:     END FOR
11:     ParentSet \leftarrow REMOVE\_NON\_INTEGRALS(ParentSet)
12:     RETURN \((\kappa \notin ParentSet)\)
13: END FUNCTION
14: FUNCTION JOIN(PATH\_INT\_1 , PATH\_INT\_2)
15:     PATH\_INT\_1 \equiv (\kappa \notin Set\_1)
16:     PATH\_INT\_2 \equiv (\kappa \notin Set\_2)
17:     RETURN \((\kappa \notin Set\_1 \cup Set\_2)\)
18: END FUNCTION
Example

```c
if (a < 0) b = 3;

if ((b < 0 || c < 0) && d < 0) {...}
```

Figure: The Main Example Program
Custom Interpolation

Figure: The Execution Tree of the Main Example Program
1 Introduction
2 Survey
3 Proposed Idea
4 Experimental Evaluation
5 Conclusion
Experimental Setup

Used Setup:
- We experimented on Intel Core i7-6700 3.40 GHz Linux Box with 32GB RAM, and a timeout of 3600 seconds.
- The raw experimental results can be accessed at [28].

Experimental Evaluation:
1. Main Experiment
   - Our Method (CUSTOM) v/s CBMC
2. Supplementary Experiment
   - No Interpolation (KLEE) and Standard Interpolation v/s CUSTOM Interpolation

Used Data set:

<table>
<thead>
<tr>
<th>Type</th>
<th>psyco</th>
<th>RERS(12-20)</th>
<th>RERS(19-Industry)</th>
<th>zodiac</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>14</td>
<td>181</td>
<td>14</td>
<td>1</td>
<td>210</td>
</tr>
</tbody>
</table>

Table: Programs Experimented.
KLEE v/s Standard Interpolation v/s CUSTOM Interpolation

No Interpolation (KLEE) v/s Standard Interpolation (TracerX)
- Forward Symbolic Execution to find feasible paths (Similar to KLEE).
- Intermediate execution states preserved (Unlike KLEE).
- Half interpolants are generated by backward tracking and Full interpolants generated by merging half interpolants.
- Full interpolants used for subsumption at similar program points.

Standard Interpolation (TracerX) v/s CUSTOM Interpolation (Paper’s Contribution)
- CUSTOM is designed to discover the MC/DC sequences and generate test cases for those sequences unlike TracerX which is used only in case of safety.
- Symbolic execution typically stops the path on witnessing a bug. In contrast, CUSTOM modifies the interpolant and continue the path.
- In CUSTOM, we generate a weakest precondition (WP) interpolant on the ghost variable (κ) alongside the standard interpolant on the rest of the variables.
Main Experiment Results

Groups
1. Both CUSTOM and CBMC terminate.
2. CUSTOM terminates, but CBMC does not terminate.
3. CBMC terminates, but CUSTOM does not terminate.
4. Neither of the tools terminate.

Table: Experimented Programs

<table>
<thead>
<tr>
<th>Groups</th>
<th>Group1</th>
<th>Group2</th>
<th>Group3</th>
<th>Group4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Programs</td>
<td>91</td>
<td>71</td>
<td>5</td>
<td>43</td>
<td>210</td>
</tr>
</tbody>
</table>

Table: Main Results (Total 710.4K sequences)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Proved Sequences (Feasible + Infeasible)</th>
<th>UnProven Sequences</th>
<th>Optimal Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBMC</td>
<td>104.7K</td>
<td>605.7K</td>
<td>96/210 (45.71%)</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>531.2K</td>
<td>179.2K</td>
<td>162/210 (77.14%)</td>
</tr>
</tbody>
</table>
**Main Experiment Results**

**Figure:** Scatter chart for MC/DC Proved Sequences

**Figure:** Scatter chart for MC/DC UnProven Sequences
Figure: Comparison of Execution Time in No Interpolation (KLEE) [2], Standard Interpolation [3] vs. CUSTOM Interpolation
Symbolic Execution (SE) is designed to perform either of these two:

1. Bug Finding
2. Program Verification

In contrast, we used it to discover MC/DC sequences and generate MC/DC specific test cases.

- Our CUSTOM interpolation technique is clever enough to prune the sub trees which contain already discovered MC/DC sequences.
- Our algorithm, if it terminates, generates an optimal set of MC/DC test cases.
Outline

1 Introduction
2 Survey
3 Proposed Idea
4 Experimental Evaluation
5 Conclusion
We have surveyed and found that in industrial practice, automatic MC/DC test generation is woefully inadequate and most practitioners rely on manual effort.

Our algorithm, if terminates, generates an optimal set of MC/DC test cases.

We compared CUSTOM against CBMC, the only practical method available which address large programs.

A comprehensive experimental evaluation shows our implementation to perform at a higher level.
References


RERS: http://kers-challenge.org/, Jun, 2018

Artifact Workbook for CUSTOM-Interpolation, https://doi.org/10.6084/m9.figshare.13650242.v1