MCDC-Star: A White-Box Based Automated Test Generation for High MC/DC Coverage

Linghuan Hu and W. Eric Wong*
Department of Computer Science
University of Texas at Dallas, USA
{linghuan.hu, ewong}@utdallas.edu

D. R. Kuhn and R. N. Kacker
National Institute of Standards and Technology, USA
{kuhn, raghu.kacker}@nist.gov

***Abstract*—The US Federal Aviation Administration requires complete modified condition/decision coverage (MC/DC) for the most critical (level A) software. Complete MC/DC is a gold standard for thoroughness of testing. However, it is challenging to generate test cases to achieve high MC/DC as it requires testers to manually conduct complex control flow analysis. In this paper, we propose MCDC-Star, a white-box based automated test case generation technique for achieving high MC/DC coverage criterion using greedy-based symbolic execution. By analyzing the control-flow of the subject program, MCDC-Star generates test cases that can improve the MC/DC efficiently and effectively. An experiment using three industrial programs was conducted, and the results show its high effectiveness and efficiency.**

Keywords—test generation, MC/DC, symbolic execution, software testing

I. Introduction

To effectively and efficiently detect bugs in software, the Federal Aviation Administration requires software venders to use modified condition/decision coverage (MC/DC), which was first defined in DO-178B [1] and updated in DO-178C [2], to ensure level A (most critical) software is adequately tested. Such software has stringent requirements regarding safety because its anomalous behavior could result in catastrophic consequences, including property damage and human casualties. Study [3] has shown that MC/DC can help testers detect more bugs, but it also significantly increases the cost of testing, up to seven times the cost of other developmental tasks. This is because to achieve high MC/DC, testers are required to manually conduct complex control flow analysis to generate the test input values, which can be extremely difficult.

Recently, both black-box based techniques, such as search-based test generation, combinatorial testing, and test generation using genetic algorithms, and white-box based techniques, such as test generation using symbolic execution (also referring to concolic execution and dynamic symbolic execution), have been proposed. In general, black-box based techniques are easy to use and can generate test cases at a low cost. However, it is very unlikely that black-box techniques will deliver outstanding performance, as some specific execution paths required by MC/DC can be only executed using one or several specific input values. In addition, the effectiveness of black-box techniques relies on the testers' knowledge to reduce the search domain, which can negatively impact their effectiveness in real-world settings.

To address this challenge, we propose MCDC-Star, which automatically generates test cases to achieve high MC/DC coverage using greedy-based symbolic execution. An experiment using three industrial programs is conducted to evaluate the efficiency of the proposed approach and the effectiveness of the test cases generated by MCDC-Star against the test cases generated by the random method. The experiment results show that MCDC-Star outperforms the random method. Currently, MCDC-Star only works for *C* programs.

* Corresponding author: Prof. W. Eric Wong

The rest of the paper is organized as follows: Section II introduces the necessary background. In Section III, the details of MCDC-Star and several running examples are presented to help understand the approach. Experiment setup and the results of the case studies are presented in Section IV. Section V addresses threats to the validity of our approach. Section VI describes the related work. The conclusions and future work are presented in Section VII.

II. Background

In this section, we give a brief introduction to the four major topics that are necessary for understanding MCDC-Star: 1) MC/DC definition and practical interpretation; 2) MC/DC measurement; 3) program instrumentation; and 4) test generation using Symbolic Execution. Hereafter, *MC/DC coverage* and *MC/DC* will be used interchangeably. In addition, *test case* and *test input values* will also be used interchangeably in the remainder of the paper.

A. MC/DC Definition and Practical Interpretation

MC/DC is defined in DO-178C as shown below:

1) *Every point of entry and exit in the program has been invoked at least once;*
2) *Every condition in a decision in the program has taken on all possible outcomes at least once;*
3) *Each condition has been shown to affect that decision outcome independently;*
4) *A condition is shown to affect a decision's outcome independently by (1) varying just that condition while holding fixed all other possible conditions or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.*

$$if\,(A \;\&\&\; B \;\&\&\; C \;\&\&\; D) \qquad (1)$$

Consider the above decision (1) with four conditions *A, B, C,* and *D*. A condition combination *c* is a set of evaluation results of the conditions of that decision. For example, (*T, T, T, T*) can be a condition combination for decision (1). An independence pair [4] (*ip*) is a pair of condition combinations that shows the

978-1-5386-9266-0/18/$31.00 ©2018 IEEE
DOI 10.1109/DSA.2018.00027

independent effect of a condition. For example, ip_1:(c_1, c_2), ip_2:(c_1, c_3), ip_3:(c_1, c_4), and ip_4:(c_1, c_5) as shown in TABLE 1 are the independence pairs for conditions *A*, *B*, *C*, and *D*, respectively. If a set of independence pairs can show the independence effects of all the conditions, this set is called an adequate set (*AS*). Therefore, TABLE 1 shows an adequate set AS_1 for decision (1).

To cover decision (1) with respect to (w.r.t.) MC/DC, we can execute the program using any test input values that can exercise an *AS* of decision (1). Notice that a decision might have multiple different adequate sets. For example, the c_2 of the ip_1 can be replaced by c_6. This is allowed by the definition of MC/DC, as the *false* value of condition *D* does not affect the decision outcome, which is also called masking MC/DC interpretation [5].

TABLE 1. AN ADEQUATE SET AS_1 FOR DECISION (1)

	A	***B***	***C***	***D***	***Decision Outcome***
c_1	*T*	*T*	*T*	*T*	*T*
c_2	*F*	*T*	*T*	*T*	*F*
c_3	*T*	*F*	*T*	*T*	*F*
c_4	*T*	*T*	*F*	*T*	*F*
c_5	*T*	*T*	*T*	*F*	*F*
c_6	*F*	*T*	*T*	*F*	*F*

We also include the short-circuit evaluation [6] in the MC/DC measurement, since most of today's programming languages evaluate decisions with short-circuit evaluation. Short-circuit evaluation is an evaluation strategy where not every condition in a decision needs to be evaluated to determine the decision outcome.

For example, consider the decision (1) and c_6:(*F, T, T, F*). Without the short-circuit evaluation, each condition needs to be evaluated to determine the outcome of the decision. With short-circuit evaluation, however, only condition *A* needs to be evaluated as it is sufficient to determine the outcome of the decision to be *false* without evaluating the rest of the conditions. As a result, TABLE 2 shows a simplified AS_2 of decision (1), where *N/A* means that the condition is not evaluated due to the short-circuit evaluation, and it can be either *true* or *false*. The independence pairs of AS_2 are, ip_4:(c_1, c_5), ip_5:(c_1, c_7), ip_6:(c_1, c_8), and ip_7:(c_1, c_9).

TABLE 2. A SIMPLIFIED AS_2 OF DECISION (1)

	A	***B***	***C***	***D***	***Decision Outcome***
c_1	*T*	*T*	*T*	*T*	*T*
c_5	*T*	*T*	*T*	*F*	*F*
c_7	*F*	*N/A*	*N/A*	*N/A*	*F*
c_8	*T*	*F*	*N/A*	*N/A*	*F*
c_9	*T*	*T*	*F*	*N/A*	*F*

B. MC/DC Measurements

To apply MC/DC in real-world settings, it is important to measure the percentage of MC/DC since not every decision can be adequality tested w.r.t MC/DC. The adequate set (AS)-based MC/DC measurement is adopted in some tools [7],[8] that are widely used in the industry. For a decision with an adequate set *AS*, this approach measures its MC/DC using the following equation:

$$\text{MC/DC} = \frac{\text{no. of exercised condition combinations in } AS}{\text{no. of all condition combinations in } AS}$$

The problem of the AS-based approach is that it might overestimate or underestimate the contributions of some condition combinations. For example, the condition combination c_1 included in ip_4, ip_5, ip_6, and ip_7 shows the independent effects of four conditions *A*, *B*, *C*, and *D* of decision (1), where c_7, c_8, c_9, and c_5 only show the independent effect of *A*, *B*, *C*, and *D*, respectively. Therefore, c_1 contributes more than each of c_7, c_8, c_9, and c_5. However, the AS-based approach will measure the MC/DC of c_1 as 20%, which significantly underestimates its contribution. Similarly, the AS-based approach will measure the MC/DC of c_7, c_8, c_9, or c_5 each as 20%, which overestimates their contributions.

To address this issue, we propose the branch independent effect (BIE)-based MC/DC approach. The BIE-based approach measures MC/DC by directly checking whether the independent effects of the branches of each condition are covered. More specifically, it treats a condition as two individual branches (a *true* branch and a *false* branch), where each branch has its own independent effect on the whole decision.

Under the short-circuit evaluation, for each branch, if it is executed and its result is not masked by other conditions, it must show the independent effect of the decision outcome. Identifying the branches that show the independent effects can be done using the abstract syntax tree (*AST*) of that decision with the following steps: 1) from the condition node (the node that represents a complete condition) to the root node, each node will be labeled with *true* or *false* according to its evaluation result; 2) starting from the root of *AST*, use recursive Algorithm 1 to identify the conditions that do not show the independent effects; and 3) the conditions that are executed and not masked show the independent effects.

```
Algorithm 1
FindMaskedAndNotExecuted(AST)
1  if AST.operator is && then
2    if AST.leftChild.value is false then
3      NotExecuted(AST.rightChild)
4      FindMaskedAndNotExecuted (AST.leftChild)
5    else
6      if AST.rightChild.value is false then
7        Mask(AST.leftChild)
8        FindMaskedAndNotExecuted (AST.rightChild)
9      else
10       FindMaskedAndNotExecuted (AST.leftChild)
11       FindMaskedAndNotExecuted (AST.rightChild)
12 if AST.operator is || then
13   if AST.leftChild.value is true then
14     NotExecuted (AST.rightChild)
15     FindMaskedAndNotExecuted(AST.leftChild)
16   else
17     if rightChild.value is true then
18       Mask (AST.leftChild)
19       FindMaskedAndNotExecuted(AST.rightChild)
20     else
21       FindMaskedAndNotExecuted (AST.leftChild)
22       FindMaskedAndNotExecuted (AST.rightChild)
```

Figure 1. Algorithm to identify masked and not-executed condition(s)

$$if\,((A \mid\mid B)\ \&\&\ (C \mid\mid D)) \tag{2}$$

For example, Figure 2 is the syntax tree of decision (2) as shown above, which is exercised by condition combination: (*T, T, F, T*).

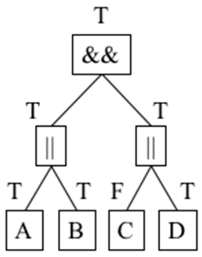


Figure 2. The syntax tree of decision (2)

By running Algorithm 1, condition *B* is not executed because of condition *A*, and condition *C* is masked by condition *D*. As a result, the *true* branch of condition *A* and the *true* branch of condition *D* independently affect the decision outcome. In this case, we say the *true* branch of condition *A* and the *true* branch of condition *D* are covered.

Using the BIE-based approach, the MC/DC of a decision is measured using the following equation:

$$\text{MC/DC} = \frac{\text{no. of covered branches}}{\text{total no. of branches}}$$

The following decision (3) is another example that demonstrates why the BIE-based approach is more accurate. Consider the decision (3) and the AS_3 shown in TABLE 3,

$$if\,(A\ \&\&\ B\ \&\&\ C) \tag{3}$$

TABLE 3. *AN ADEQUATE SET* AS_3 OF DECISION (3)

	A	***B***	***C***	***Decision Outcome***
c_{10}	*T*	*T*	*T*	*T*
c_{11}	*F*	*N/A*	*N/A*	*F*
c_{12}	*T*	*F*	*N/A*	*F*
c_{13}	*T*	*T*	*F*	*F*

The AS-based approach measures the MC/DC of c_{10} as 25%, which underestimates its contribution. The BIE-based approach, however, measures the MC/DC of c_{10} as 50% because it shows the independent effects of the *true* branches of conditions *A*, *B*, and *C*. Similarly, the AS-based approach overestimates the MC/DC contributions of c_{11}, c_{12}, and c_{13} by reporting 25% for each of them, while the BIE-based approach reports 16.67% for each of them. Therefore, the BIE-based approach measures MC/DC more accurately than the AS-based approach.

C. Program Instrumentation

Automated MC/DC measurement is very important for MCDC-Star. To measure MC/DC for a program, we need the boolean value of each condition when its decision is executed. Unfortunately, we are not aware of any existing tools that satisfy our needs. Therefore, we designed our MCDC-Star to automatically perform program instrumentation w.r.t. MC/DC. MCDC-Star uses *instrumentation variables* to save the value of each condition when it is evaluated. Consider the following decision (4) with three conditions and two boolean operators.

$$if\,((a == 3 \mid\mid foo(b))\ \&\&\ c > 200) \tag{4}$$

MCDC-Star constructs the syntax tree of decision (4), adds instrumentation variables *iv*, and converts it to decision (5) as shown below.

$$if\,((iv1 = (a==3) \mid\mid iv2 = (foo(b)))\ \&\&\ iv3 = (c>200)) \tag{5}$$

In addition to the instrumentation variables, we also need to save these values of the instrumentation variables to trace files by adding two *saving functions* for each decision. For an *if* statement, MCDC-Star adds one *saving function* before the first statement of the *true* branch and another before the first statement of the *false* branch. A similar process is also applied to *for*, *while*, and *do-while* decisions. Figure 3 shows the code segment before and after the instrumentation.

```
if (a > b && c < a)
   max = a;
else
   max = b;

for (a; a > y; a++)
   foo(x)

while (b < z)
   foo(x)

do
{
   foo(x)
} while (c < z);
```

(a) Before instrumentation

```
if (iv1 = (a > b) && iv2 = (c < a))
{
   save (iv1, iv2)
   max = a;
}
else
{
   save (iv1, iv2)
   max = b;
}

for (a; iv1 = (a > y); a++)
{
   save(iv1)
   foo(x)
}
save(iv1)

while (b < z)
{
   save(iv1)
   foo(x)
}
save(iv1)

int first = 1;
do
{
   if (first != 1)
   {
      save(iv1);
      first = 0;
   }
```

```
    foo(x)
} while (iv1 = (c < z));
save(iv1)
```

(b) After instrumentation

Figure 3. MC/DC Instrumentation

D. Test Generation Using Symbolic Execution

The symbolic execution was first introduced in the study by J. C. King [9], which proposes a new method of program analysis and test generation. In general, when a program is executed, the symbolic executor can substitute some variables of the program with symbolic variables. The symbolic variable not only stores the value but also maintains the symbolic expression of that variable, which is updated along with the program execution. During the program execution, the symbolic executor will construct a *path constraint* (*pc*), which is a boolean expression, at each decision point, e.g., a condition in a decision (*if*, *for*, *while*, etc.). The *pc* is constructed using the symbolic expressions of the symbolic variables that are related to this decision point. When the program execution finishes, a boolean formula *PC* of the execution path is constructed by connecting all pc_i using *and* operator (e.g., $PC = pc_1 \wedge pc_2 \wedge \ldots \wedge pc_n$). Next, a constraint solver tries to solve this *PC* to generate the corresponding input values. If the generation is successful, we then obtain a set of input values for this specific execution path. If it is not successful, then this path is considered an infeasible path. Notice that the unsuccessful generation for a certain path does not guarantee the infeasibility of this path, as some pc_i of *PC* are neither too difficult to be solved nor supported by the solver.

Figure 4 shows a *C* function that determines whether a given character is in the alphabet or not.

```
1.  int check_alphabet (char c)
2.  {
3.      if (( c >= 'a' && c <= 'z' ) || ( c >= 'A' && c <= 'Z' ))
4.          return 1;
5.      else
6.          return 0;
7.  }
```

Figure 4. A function that checks whether a given character is in the alphabet

Figure 5 shows the code structure of the function *check_alphabet* in assembly code view. The *if* decision shown in line 3 of Figure 4 is represented by the code blocks 2, 3, 4, and 5. TABLE 4 shows the seven possible execution paths of *check_alphabet*, their corresponding *PC*, and a possible generated input for each *PC*. A traditional symbolic executor usually explorers all the paths of the program by negating every *pc* using a DFS or BFS-based algorithm, e.g, SAGE [10]. However, exploring every path using symbolic execution in real-world settings might not be feasible, as it suffers from the path explosion problem [11].

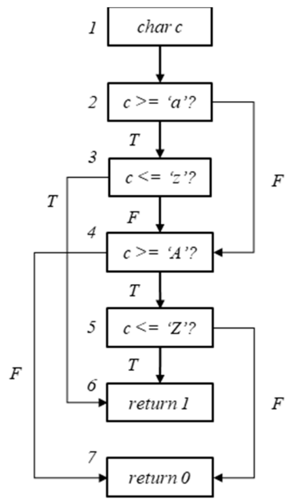


Figure 5. *check_alphabet* function in assembly code view

TABLE 4. SEVEN POSSIBLE PATHS, THEIR CORRESPONDING *PC*, AND A POSSIBLE INPUT

Execution Path		***PC***	***Input***
p_1	*1, 2, 3, 6*	$c >= \text{'a'} \wedge c <= \text{'z'}$	*'b'*
p_2	*1, 2, 4, 5, 6*	$\neg (c >= \text{'a'}) \wedge c >= \text{'A'} \wedge c <= \text{'Z'}$	*'B'*
p_3	*1, 2, 3, 4, 5, 6*	$c >= \text{'a'} \wedge \neg (c <= \text{'z'}) \wedge c >= \text{'A'} \wedge c <= \text{'Z'}$	*N/A*
p_4	*1, 2, 4, 7*	$\neg (c >= \text{'a'}) \wedge \neg (c >= \text{'A'})$	*'#'*
p_5	*1, 2, 3, 4, 7*	$c >= \text{'a'} \wedge \neg (c <= \text{'z'}) \wedge \neg (c >= \text{'A'})$	*N/A*
p_6	*1, 2, 4, 5, 7*	$\neg (c >= \text{'a'}) \wedge c >= \text{'A'} \wedge \neg (c <= \text{'Z'})$	*']'*
p_7	*1, 2, 3, 4, 5, 7*	$c >= \text{'a'} \wedge \neg (c <= \text{'z'}) \wedge c >= \text{'A'} \wedge \neg (c <= \text{'Z'})$	*'}'*

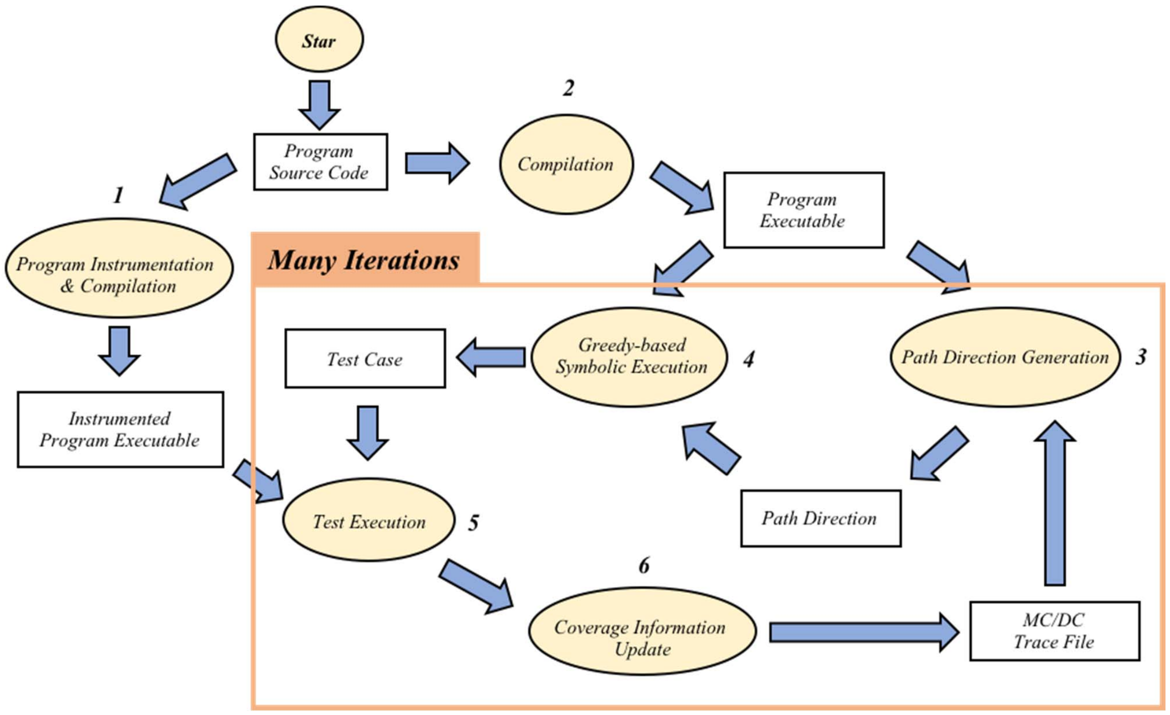
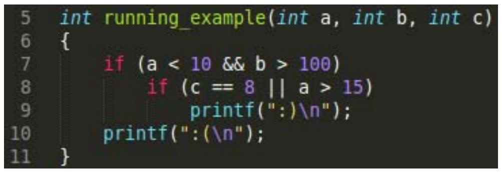

Star
Program Source Code
2
Compilation
Program Executable
1
Program Instrumentation & Compilation
Many Iterations
Test Case
Greedy-based Symbolic Execution
4
Path Direction Generation
3
Instrumented Program Executable
Test Execution
5
Path Direction
6
Coverage Information Update
MC/DC Trace File

Figure 6. An overview of MCDC-Star

III. Our Technique

In this section, we will explain the details of the proposed MCDC-Star technique and give several running examples along with descriptions to help understand how it works. Figure 6 shows an overview of how MCDC-Star generates test cases for a *C* program in six steps.

The overall test generation process can be described as follows. *Step 1* is to instrument the subject program and compile it to get the instrumented executable. The instrumented executable will be used in *Step 5* for the test execution later. We also compile the subject program without instrumentation to obtain its executable in *Step 2*. In *Step 3*, the executable and the coverage information are then used to generate a desired path direction, which will be used to guide the symbolic execution in *Step 4*. A greedy-based symbolic execution will be performed in *Step 4* to try to generate the test input values. Once the test input values are successfully generated, we execute against the instrumented executable generated in *Step 1* and update the MC/DC coverage information. *Steps 3*, *4*, *5*, and *6* are conducted repeatedly until no more test input values can be successfully generated. We now explain the details of each step using a sample *C* program *running_example* as shown in Figure 7. Notice that, although for demonstration purposes the *running_example* does not contain any loops, all the analysis presented in the following sections can be applied to *for, while,* and *do-while* in a similar manner.

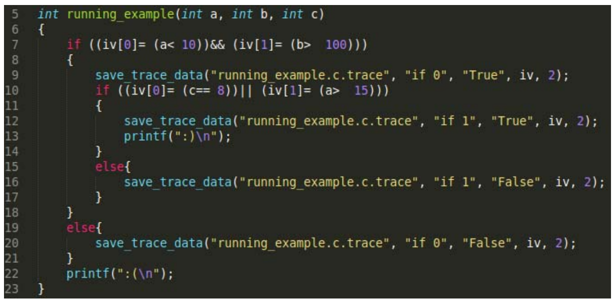


```c
int running_example(int a, int b, int c)
{
    if (a < 10 && b > 100)
        if (c == 8 || a > 15)
            printf(":)\n");
    printf(":(\n");
}
```

Figure 7. The source code of *running_example*

A. Step 1: Program Instrumentation and Compilation

MCDC-Star needs to keep track of the MC/DC coverage information of each decision of the program to generate the test input values that can improve the MC/DC effectively. To do this, we instrument the subject program and compile it using *GCC*, the gnu compiler [12], to generate the instrumented executable, which will be used in *Step 5*. Figure 8 shows the source code of the *running_example* after the instrumentation.



```c
int running_example(int a, int b, int c)
{
    if ((iv[0]= (a< 10))&& (iv[1]= (b>  100)))
    {
        save_trace_data("running_example.c.trace", "if 0", "True", iv, 2);
        if ((iv[0]= (c== 8))|| (iv[1]= (a>  15)))
        {
            save_trace_data("running_example.c.trace", "if 1", "True", iv, 2);
            printf(":)\n");
        }
        else{
            save_trace_data("running_example.c.trace", "if 1", "False", iv, 2);
        }
    }
    else{
        save_trace_data("running_example.c.trace", "if 0", "False", iv, 2);
    }
    printf(":(\n");
}
```

Figure 8. Source code of *running_example* after the instrumentation

B. Step 2: Compilation

In addition to the instrumented executable generated in *Step 1*, we also compile the subject program using *GCC* with its debug option "-g" enabled. The debug option is required as it will generate additional debug information, which will be analyzed to generate the path direction in *Step 3*. The generated executable will also be used for symbolic execution later on in *Step 4*.

C. Step 3: Path Direction Generation

The key of generating test cases that can effectively improve the MC/DC of the subject program using symbolic execution is to identify the specific path that can improve the MC/DC effectively. Suppose a subject program has in total n execution paths, $P = \{p_1, p_2, \ldots, p_n\}$. Theoretically, we can generate and execute test input values using symbolic execution to improve MC/DC of a subject program in the following optimal way: a) prioritize these execution paths w.r.t. their MC/DC improvements in a descending order; b) run the symbolic executor to generate the input values of execution path with the highest rank (suppose that the symbolic executor can generate input values for all execution paths); c) execute the generated input values; and then repeat a), b), and c) until the MC/DC cannot be further improved. However, due to memory and computation limitations, this is not feasible, as there might be an overwhelmingly large number of execution paths, which can make identifying all execution paths of a program and the path prioritization impossible. Furthermore, in real-world settings, this approach might be useless if its computation cost (w.r.t. time or computation power) is unacceptable.

To overcome this challenge, MCDC-Star conducts the control-flow analysis to identify the path directions of each decision and lets the symbolic executor only generate input values for those directions that can improve MC/DC effectively. We will elaborate on the path direction generation process using *running_example*, as shown in Figure 7.

For a program, we need to construct the path direction $PD = \{i = 1 \ldots n, k = 1 \ldots m \mid pd_{i,k}\}$, where i represents the ith decision in the program and k represents the ith decision's kth condition combination. Each $pd_{i,k} = (c_{i,k}, mcdc_{i,k}, aes_{i,k})$ is a three tuple, where $c_{i,k}$, $mcdc_{i,k}$, and $aes_{i,k}$ represent the ith decision's kth condition combination, the corresponding MC/DC improvement, and the assembly code execution sequence, respectively. We will now present the details on how $pd_{i,k}$ is constructed.

First, $c_{i,k}$ and $mcdc_{i,k}$ can be generated using the BIE-based MC/DC measurement approach. *running_example* has the following two *if* decisions (6) and (7) at lines 7 and 8.

$$if\,(a < 10 \;\&\&\; b > 100) \qquad (6)$$

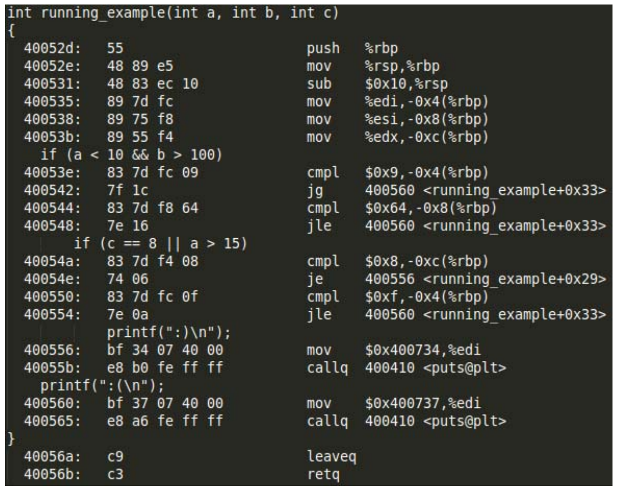
$$if\,(c == 8 \;||\; a > 15) \qquad (7)$$

Initially, *running_example is not executed by any input values*; therefore, the MC/DC of each decision in *running_example* is 0.0%. MCDC-Star computes each condition combination's MC/DC improvement for each decision to obtain $mcdc_{i,k}$, as shown in TABLE 5.

TABLE 5. CONDITION COMBINATIONS AND THEIR MC/DC IMPROVEMENTS

Decision 1: if (a < 10 && b > 100)					
Condition Combination			*MC/DC Improvement*	*Will Cover*	*Covered Branches*
$c_{1,1}$	*T*	*T*	*50.0%*	*1st: True 2nd: True*	*N/A*
$c_{1,2}$	*T*	*F*	*25.0%*	*2nd: False*	*N/A*
$c_{1,3}$	*F*	*N/A*	*25.0%*	*1rd: False*	*N/A*
Decision 2: if (c == 8 || a > 15)					
Condition Combination			*MC/DC Improvement*	*Will Cover*	*N/A*
$c_{2,1}$	*T*	*N/A*	*25.0%*	*1st: True*	*N/A*
$c_{2,2}$	*F*	*T*	*25.0%*	*2nd: True*	*N/A*
$c_{2,3}$	*F*	*F*	*50.0%*	*1st: False 2nd: False*	*N/A*

Next, $aes_{i,k}$ can be generated by analyzing the assembly code, as shown in Figure 9. The assembly code is generated by *objdump* [13] from the executable that we obtained from *Step 2*.



```
int running_example(int a, int b, int c)
{
  40052d:   55                      push   %rbp
  40052e:   48 89 e5                mov    %rsp,%rbp
  400531:   48 83 ec 10             sub    $0x10,%rsp
  400535:   89 7d fc                mov    %edi,-0x4(%rbp)
  400538:   89 75 f8                mov    %esi,-0x8(%rbp)
  40053b:   89 55 f4                mov    %edx,-0xc(%rbp)
    if (a < 10 && b > 100)
  40053e:   83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
  400542:   7f 1c                   jg     400560 <running_example+0x33>
  400544:   83 7d f8 64             cmpl   $0x64,-0x8(%rbp)
  400548:   7e 16                   jle    400560 <running_example+0x33>
        if (c == 8 || a > 15)
  40054a:   83 7d f4 08             cmpl   $0x8,-0xc(%rbp)
  40054e:   74 06                   je     400556 <running_example+0x29>
  400550:   83 7d fc 0f             cmpl   $0xf,-0x4(%rbp)
  400554:   7e 0a                   jle    400560 <running_example+0x33>
            printf(":)\n");
  400556:   bf 34 07 40 00          mov    $0x400734,%edi
  40055b:   e8 b0 fe ff ff          callq  400410 <puts@plt>
    printf(":(\n");
  400560:   bf 37 07 40 00          mov    $0x400737,%edi
  400565:   e8 a6 fe ff ff          callq  400410 <puts@plt>
}
  40056a:   c9                      leaveq
  40056b:   c3                      retq
```

Figure 9. Assembly code of *running_example*

The generated assembly code contains the detailed assembly execution information, where each condition of a decision can be mapped to a jump instruction. For each decision, we only need the jump instructions and the address of the first instruction of the *true* and *false* branch of the decision. By parsing this assembly code, we can obtain the mapping shown in TABLE 6.

TABLE 6. EACH CONDITION AND ITS CORRESPONDING JUMP INSTRUCTIONS

Branch Address	*Assembly Code*	
Decision 1: if (a < 10 && b > 100)		
a < 10	*400542*	*jg 400560*
b > 100	*400548*	*jle 400560*

First Instruction of True Branch	*40054a*	*Not Interested*
First Instruction of False Branch	*400560*	*Not Interested*
Decision 2: if (c == 8 || a > 15)		
c == 8	*40054e:*	*je 400556*
c > 15	*400554:*	*jle 400560*
First Instruction of True Branch	*400556*	*Not Interested*
First Instruction of False Branch	*400560*	*Not Interested*

A jump instruction, such as *jg* or *jle*, has a destination address, which represents the next instruction's address if the jump instruction's condition is satisfied. If the jump instruction's condition is not satisfied, the execution will simply move to its next instruction. Although whether a jump instruction's condition can be satisfied or not depends on the actual execution result, a jump instruction's condition will be satisfied if the result of its corresponding condition can short-circuit other conditions (except for the final jump instruction). Hence, the execution sequence that is represented by the instruction address can be constructed using this rule. For the final jump instruction, if the decision outcome is *true*, we append the address of the *true* branch's first instruction to the execution sequence, and vice versa.

For example, if both conditions of the *Decision 1* shown in TABLE 6 are *true*, the conditions of both jump instructions will not be satisfied because no condition can short-circuit other conditions. Therefore, the execution sequence will be (*40053e* → *400542* → *400544* → *400548*), where *NI* represents the instructions in which we are not interested. Since the decision outcome is *true*, the address of the first instruction of the true branch *40054a* is appended to the execution sequence. As a result, the execution sequence will be (*40053e* → *400542* → *400544* → *400548*→ *40054a*).

If the first condition is *false*, it will short-circuit the second condition. Therefore, the condition of the jump instruction of the first condition at *400542* will be satisfied. Hence, the execution sequence will be (*40053e* → *400542* → *400560*). Because the decision outcome is *false* and *400560* is already the address of the first instruction of the decision's *false* branch, there is no need to append *400560* at the end of the execution sequence.

By conducting this analysis, MCDC-Star maps $c_{i,k}$ to the corresponding execution sequence $ase_{i,k}$. At this point, we now obtain the path directions, $PD = \{i = 1 \ldots n,\ k = 1 \ldots m \mid pd_{i,k} = (c_{i,k},\ mcdc_{i,k},\ aes_{i,k})\}$ of each decision. TABLE 7 shows the *PD* of *running_example.*

TABLE 7. CONSTRUCTED *PD* OF *RUNNING_EXAMPLE*

Condition Combination			*MC/DC Improvement*		*Assembly Code Execution Sequence*	
Decision 1: if (a < 10 && b > 100)						
$c_{1,1}$	*T*	*T*	$mcdc_{1,1}$	*50.0%*	$aes_{1,1}$	*40053e* → *400542* → *400544* → *400548* → *40054a*
$c_{1,2}$	*F*	*N/A*	$mcdc_{1,2}$	*25.0%*	$aes_{1,2}$	*40053e* → *400542* → *400560*
$c_{1,3}$	*T*	*F*	$mcdc_{1,3}$	*25.0%*	$aes_{1,3}$	*40053e* → *400542* → *400544* → *400548* → *400560*
Decision 2: if (c == 8 || a > 15)						
$c_{2,1}$	*T*	*N/A*	$mcdc_{2,1}$	*25.0%*	$aes_{2,1}$	*40054a* → *40054e* → *400556*
$c_{2,2}$	*F*	*T*	$mcdc_{2,2}$	*25.0%*	$aes_{2,2}$	*40054a* → *40054e* → *400550* → *400554* → *400556*
$c_{2,3}$	*F*	*F*	$mcdc_{2,3}$	*50.0%*	$aes_{2,3}$	*40054a* → *40054e* → *400550* → *400554* → *400560*

D. Steps 4, 5, and 6

Once we obtain the *PD* of the subject program from *Step 3*, we now can generate the test input values for the subject program. Before we start the symbolic execution, MCDC-Star uses greedy strategy to create an assembly execution sequence set AES_j, where *j* represents the *j*th symbolic execution. AES_j consists of $aes_{i,k}$ from each $pd_{i,k}$ that has the highest $mcdc_{i,k}$. For example, the AES_1 of *running_example* = $\{aes_{1,1},\ aes_{2,3}\}$ = {(*40053e* → *400542* → *400544* → *400548* → *40054a*), (*40054a* → *40054e* → *400550* → *400554* → *400560*)}. In the actual symbolic execution, the AES_j will be used to guide the symbolic executor to explore the path directions that have the highest MC/DC improvement of each direction.

For the symbolic execution, MCDC-Star uses *Triton* [14], a highly customizable dynamic symbolic executor to generate input values. Notice that different symbolic executors can be easily integrated with MCDC-Star.

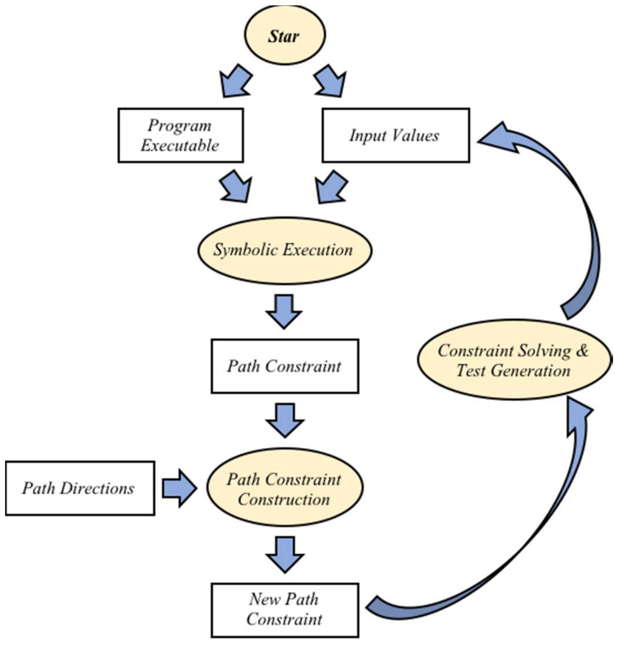


Figure 10. Overview of *Step 4*

Figure 10 shows the overview of *Step 4*. First, *Triton* starts the symbolic execution using the program executable and initial input values (initial input values can be either fixed or randomly generated). When the symbolic execution is completed, *Triton* analyzes the execution path to generate its corresponding path constraint $PC_i = \{i = 1...n, k = 1...m \mid pc_{i,k} = (da_{i,k}, be_{i,k}, bne_{i,k})\}$, where *i* represents the *ith* iteration of the current symbolic execution and $da_{i,k}$, $be_{i,k}$, and $bne_{i,k}$ represent the address of the *k*th decision point, the address of the corresponding branch that was executed after the $da_{i,k}$, and the branch that was not executed, respectively. For a $pc_{i,k}$, if the $be_{i,k}$ is not the desired execution direction, we can swap the $be_{i,k}$ with $bne_{i,k}$ so that we can use solver to generate test input values that will execute $bne_{i,k}$.

For example, in the first symbolic execution of *running_example*, if test input values (*a = 0, b = 0, c = 8*) are used, the collected PC_1 will be $\{pc_{1,1}(400542, 400544, 400560)$, $pc_{1,2}(400548, 400560, 40054a)\}$. As we mentioned previously, we want to generate the input values that can execute the execution sequences with the highest MC/DC improvements, which is indicated by the $AES_1 = \{aes_{1,1}(40053e \rightarrow 400542 \rightarrow 400544 \rightarrow 400548 \rightarrow 40054a)$, $aes_{2,3}(40054a \rightarrow 40054e \rightarrow 400550 \rightarrow 400554 \rightarrow 400560)\}$. Therefore, we check PC_1 with AES_1 to see whether each *be* is our desired execution direction or not. In this example, $pc_{1,1}$ indicates that the actual execution after *400548* went to the instruction at *400560*, which does not match the execution sequence $aes_{1,1}$ of AES_1. Hence, we modify the $pc_{1,2}$ to $pc_{1,2}$' (*400548, 40054a, 400560*), and then we obtain our new $PC_1' = \{pc_{1,1}(400542, 400544, 400560), pc_{1,2}'(400548, 40054a, 400560)\}$.

Next, *Triton* uses constraint solver to solve the PC_1' and gets generated input values (*a = 5, b = 150, c = 8*). We then use the input values (*a = 5, b = 150, c = 8*) to continue the next iteration of the symbolic execution, which will return a $PC_2 = \{pc_{2,1}(400542, 400544, 400560)$, $pc_{2,2}(400548, 40054a, 400560)$, $pc_{2,3}(40054e, 400556, 400550)\}$. By checking the PC_2 with AES_1 again, we can construct a new $PC_2' = \{pc_{2,1}(400542, 400544, 400560)$, $pc_{2,2}(400548, 40054a, 400560)$, $pc_{2,3}'(40054e, 400550, 400556)\}$. A new input (*a = 5, b = 150, c = 0*) can be solved and generated by the constraint solver.

At this point, if we execute the input values (*a = 5, b = 150, c = 0*) and check the obtained PC_3, we will observe that no more execution sequences can be solved. Hence, we can stop *Step 4* and move on to *Step 5* to conduct the actual test execution.

In *Step 5*, we execute the generated input values (*a = 5, b = 150, c = 0*) and then update the MC/DC of *running_example*. We achieve 50% MC/DC in *Step 6* (four covered independent effects divided by eight independent effects in total), as $c_{1,1}$ and $c_{2,3}$ are covered. Once *Step 6* is complete, we then return to *Step 3* to obtain a new *PD*, followed by *Steps 4, 5*, 6, and so on. *Steps 3, 4, 5,* and *6* will be conducted repeatedly until the MC/DC cannot be further improved.

Although the major steps of MCDC-Star have been presented, there are two issues we need to overcome to ensure the test input values can be generated successfully. The first issue is that the symbolic execution using the presented greedy strategy might not be able to reach some decisions. For example, consider that we have finished the first three symbolic execution of *running_example*, and we are about to start the fourth symbolic execution. After the first three executions, we already covered all the condition combinations of *decision 1* and the $c_{2,3}$ of *decision 2*. The AES_4 will be $\{aes_{2,1}(40054a \rightarrow 40054e \rightarrow 400556)\}$, which contains no information for *decision 1*. Because *decision 2* is in the *true* branch of *decision 1*, the initial input values cannot take the execution path to the *true* branch of *decision 1*. Therefore, *decision 2* will not be reached.

To address this issue, MCDC-Star will check the current *PC* to determine whether we have reached any decision points that are in the *AES*. If the result is *false* and *AES* is not *null*, we will combine the current *AES* with every previous *AES* and start the symbolic execution again until we can reach some decision points in the *AES*. For example, if the initial input values of the fourth symbolic execution are (*a = 0, b = 0, c = 8*), the PC_4 will be $\{pc_{4,1}(400542, 400544, 400560)$, $pc_{4,2}(400548, 400560, 40054a)\}$. MCDC-Star detects that no decision points in AES_4 are in PC_4; therefore, it then combines the AES_1 with AES_4 to generate $AES_4' = (aes_{1,1}(40053e \rightarrow 400542 \rightarrow 400544 \rightarrow 400548 \rightarrow 40054a)$, $aes_{2,1}(40054a \rightarrow 40054e \rightarrow 400556))$. As a result, the symbolic execution will reach *decision 2 and generate* the corresponding input values.

Another issue is that if a condition combination has the highest MC/DC improvements but cannot be solved due to constraint conflicts, the symbolic execution might end up with an infinite loop. For example, consider that we have finished the first four symbolic executions of *running_example* and are about to start the fifth symbolic execution. MCDC-Star will pick $c_{2,2}$ to construct AES_5. The AES_5 is obviously not solvable as it requires *decision 1* and the second condition of *decision 2*, as shown below, to be *true* at the same time.

$$a < 10 \; \&\& \; b > 100 \; \&\& \; a > 15$$

In this simple example, we know that the constraint conflict is due to $a < 10$ and $a > 15$. However, in real-world settings, it is very difficult to analyze these constraint conflicts, as they require tremendous data-flow analysis. However, if we do not address the constraint conflict issue, the MC/DC might not be improved effectively, or we can even end up in a dead loop, as MCDC-Star will select this $c_{2,2}$ again. To address this, MCDC-Star will try to solve and generate the input values using the intermediate constructed *PC* whenever a swap of *be* and *bne* is conducted. If an intermediate *PC* cannot be solved, we compare this intermediate *PC* with its previous version to identify the last changed $pc_{i,k}$. Then, we trace back to its corresponding condition combination that cannot be solved and add it to a blacklist, so it will be ignored in future symbolic executions.

IV. Case Studies

A. Subject Programs

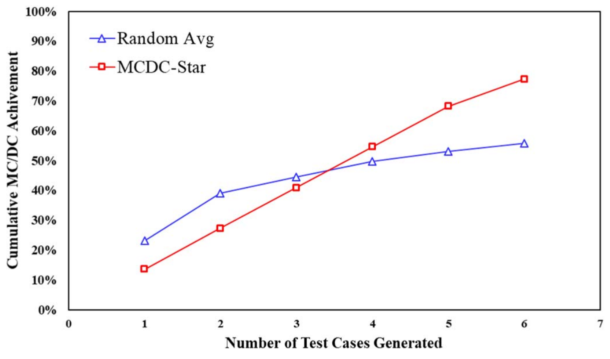
We conducted our case studies using three industrial programs. The first program is *utf8toutf16* [15], which converts a utf8 encoding to a utf16 encoding. The second program is *tcas* [16], a well-known traffic collision avoidance system. The third program, Photo Editing Line (*PEL*) [17], is a constraint verification module of a customization program for a photo editing software. TABLE 8 shows the line of code and number of decisions of each subject program.

TABLE 8. INFORMATION OF THE THREE SUBJECT PROGRAMS

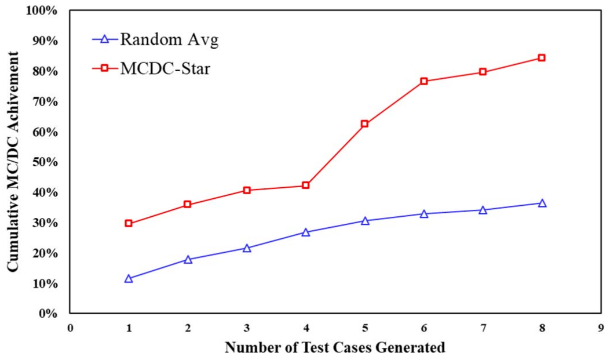
	Subject Programs		
	utf8toutf16	*tcas*	*PEL*
LOC	*70*	*197*	*188*
No. of Decisions	*7*	*15*	*14*

B. *Experiment Setup*

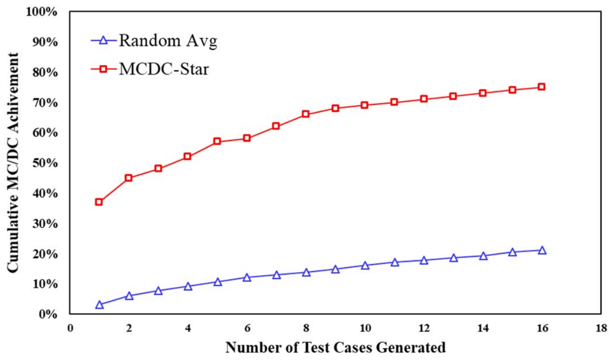
We first evaluate the effectiveness of MCDC-Star by measuring the cumulative MC/DC achievement after executing the *i*th test case generated by MCDC-Star versus a random method. For each subject program, if MCDC-Star generates *n* test cases (the generated test cases by MCDC-Star are deterministic), we also stop the random method after executing *n*th test cases to conduct a fair comparison. In addition, to remove any potential bias, we independently run the random method 50 times and compare their average results with MCDC-Star.



(a) *utf8toutf16*



(b) *tcas*



(c) *PEL*

Figure 11. Comparison between the random method (average score) and MCDC-Star w.r.t. MC/DC achievement

TABLE 9. THE CUMULATIVE MC/DC COVERAGE ACHIEVEMENTS OF RANDOM METHOD AND MCDC-STAR AFTER EXECUTING EACH GENERATED TEST CASE

	utf8toutf16		*tcas*		*PEL*	
	Random	MCDC-Star	Random	MCDC-Star	Random	MCDC-Star
t_1	23.27%	13.64%	11.50%	29.69%	3.16%	37.00%
t_2	39.00%	27.27%	17.97%	35.94%	6.18%	45.00%
t_3	44.45%	40.91%	21.72%	40.62%	7.74%	48.00%
t_4	49.63%	54.55%	26.88%	42.19%	9.30%	52.00%
t_5	53.18%	68.18%	30.56%	62.50%	10.80%	57.00%
t_6	55.91%	77.27%	32.84%	76.56%	12.22%	58.00%
t_7	-	-	34.28%	79.69%	13.12%	62.00%
t_8	-	-	36.56%	84.38%	13.88%	66.00%
t_9	-	-	-	-	14.86%	68.00%
t_{10}	-	-	-	-	16.10%	69.00%
t_{11}	-	-	-	-	17.16%	70.00%
t_{12}	-	-	-	-	17.88%	71.00%
t_{13}	-	-	-	-	18.62%	72.00%
t_{14}	-	-	-	-	19.34%	73.00%
t_{15}	-	-	-	-	20.52%	74.00%
t_{16}	-	-	-	-	21.18%	75.00%

TABLE 10. THE COMPARISON BETWEEN THE HIGHEST MC/DC ACHIEVEMENTS OF MCDC-STAR AND THE MC/DC ACHIEVEMENTS OF RANDOM METHOD AFTER EXECUTING 100 TEST CASES

utf8toutf16				*tcas*				*PEL*			
Random		MCDC-Star		Random		MCDC-Star		Random		MCDC-Star	
t_{100}	76.00%	t_6	77.27%	t_{100}	71.78%	t_8	84.38%	t_{100}	46.86%	t_{16}	75.00%

C. *Experiment Results*

Figure 11 shows the comparison between the random method and MCDC-Star, and TABLE 9 shows the detailed cumulative MC/DC coverage achievements of the random method and MCDC-Star after executing each test case. For *tcas* and *PEL*, MCDC-Star achieved higher cumulative MC/DC than the random method after executing every generated test case. In *utf8toutf16*, the random method achieved higher cumulative MC/DC in the first three test cases and was then outperformed by MCDC-Star after executing the fourth test case. We investigated the reason why the random method outperformed MCDC-Star until the fourth execution. The reason is that *utf8toutf16* has several condition combinations that have the highest MC/DC improvement for that decision, but it will end the test execution after that, so other decisions cannot be executed. This is a typical local optimization versus global optimization problem. On average, MCDC-Star outperforms the random method by 34.35%.

In addition, since MCDC-Star achieves higher cumulative coverage than the random method in each subject program after executing the last test case generated by MCDC-Star, we continue the test generation of the random method to examine whether it can outperform MCDC-Star after executing up to 100 test cases. The results presented in TABLE 10 show that MCDC-Star still outperforms the random method even after executing 100 test cases.

Due to tool limitations, we did not conduct a comprehensive evaluation on the efficiency of w.r.t., the time required for generating each test case. One reason for this is that one of our required tools (*pintool*) will cause compatibility issues when using an Intel Core Processor that is newer than the fourth generation, and we have not yet found a solution to address this. As a result, the desktop we used for the experiment has an Intel i7-4790 processor, which is outdated and cannot reflect the real efficiency of using a state-of-the-art processor. Another reason is that MCDC-Star is a technique that automatically generates test cases by guiding symbolic execution using the greedy strategy. Different symbolic executors can be easily integrated with MCDC-Star. Therefore, the efficiency of MCDC-Star should not be simply evaluated by the execution time of one symbolic executor.

Although we did not conduct a comprehensive efficiency analysis, MCDC-Star generates test cases very fast. The time of generating each test case is less than 1 minute for both *tcas* and *PEL* and about 2 to 4 minutes per test case for *utf8toutf16*. In the future, we will conduct a fair evaluation of its efficiency using different symbolic executors.

V. THREATS TO VALIDITY

We evaluate the effectiveness of the proposed MCDC-Star by comparing the cumulative MC/DC achievement after executing the *i*th test case generated by MCDC-Star and the random method. The bias of the random method can be an external validity to our study. We reduce this threat by independently running the random method 50 times and using the average results for the comparison. Another external validity is whether our evaluation results can reflect the actual effectiveness of applying MCDC-Star to other programs. We mitigate this threat by using three distinct kinds of subject programs that are different from each other. We are confident that MCDC-Star can be still effective on other programs. We did not use programs such as *CalDate* or *Triangle* [18] as they are not industrial programs.

VI. RELATED WORK

Due to the outstanding bug detection effectiveness but high testing cost of MC/DC, many techniques have been proposed to improve it through the automatic generation of test cases. Some of the techniques, however, still require human effort in their test generation. For example, the model-based techniques, such as [19]-[21], require testers to design the input or system model before the test generation. Other search-based techniques, such as [22] and [23], use genetic algorithms to search input values from the input domain.

Recently, more advanced symbolic execution-based (also referring to dynamic symbolic execution and concolic execution) techniques have been proposed. In the beginning, by solving each path constraint collected using the DFS or BFS-based algorithms, e.g., SAGE, test cases can be generated to improve MC/DC. However, this is not effective, and it is time consuming since it is essentially an exhaustive-based approach. More effective code transformation-based symbolic execution techniques, such as [24] and [25], have been proposed to generate test cases for achieving structural code coverage criterion, such as MC/DC. However, the code transformation-based strategy can change the program's behavior, which limits its practicality. Su et al. [26] propose a coverage-driven test data generation technique that only solves the *PC* of the path that contains the uncovered targets (could be branch, decision, etc. with respect to the different coverage criteria). Wu et al. [18] propose a similar greedy-based symbolic execution but with 1) inaccurate AS-based MC/DC measurement; 2) manual instrumentation for MC/DC measurement that limits its practicality; and 3) no path constraint conflict solution as presented in Section III.D.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we first investigate the issue of the widely-used AS-based MC/DC measurement approach and propose the BIE-based MC/DC measurement approach. Then, we propose MCDC-Star, which provides fully automated:

1) MC/DC instrumentation;
2) MC/DC measurement;
3) Test case generation using greedy-based symbolic execution.

An experiment is conducted using three industrial programs. The results show that MCDC-Star outperforms the random method by 34.35% on average with respect to cumulative MC/DC after executing each generated test case. Additionally, the MCDC-Star still outperforms the random method even after the random method generates and executes 100 test cases. In the future, we will further enhance the constraint conflict solving ability and revise our greedy strategy. Disclaimer: Any mention of commercial products in this paper is for information only; it does not imply recommendation or endorsement by the national institute of standards and technology (NIST).

References

[1] RTCA, DO-178B: Software considerations in airborne systems and equipment certification. Washington, RTCA, Inc., December 1992

[2] RTCA, DO-178C: Software considerations in airborne systems and equipment certification. Washington, RTCA, Inc., December 2011

[3] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate, "Testing or formal verification: DO-178C alternatives and industrial experience," IEEE software, 30(3), 50-57, 2013

[4] A Practical Tutorial On Modified Condition/Decision Coverage

[5] John J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Tech. Rep. DOT/FAA/AR-01/18, Federal Aviation Administration, US-Department of Transportation, Washington, DC, April 2001

[6] K. Louden, Programming languages: principles and practices. Cengage Learning, January 2011

[7] Is 100% Code Coverage Enough?, https://www.hitex.com/fileadmin/documents/tools/dynamic/tessy/WP-TESSY-Is-100-Percent-Code-Coverage-Enough.pdf, accessed July 2018

[8] Telelogic Logiscope TestChecker - Getting Started Version 6.5, ftp://public.dhe.ibm.com/software/rationalsdp/documentation/archive/Logiscope/version_6-5/TestGS.pdf, June 2016

[9] J. C. King, "Symbolic execution and program testing," ACM Communication, 19:385–394, July 1976

[10] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, pp. 20, January 2012

[11] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," Journal of Systems and Software, vol. 86, no. 8, pp. 1978-2001, 2013

[12] GCC, the GNU Compiler Collection, https://gcc.gnu.org/, accessed July 2018

[13] GNU Binutils, https://www.gnu.org/software/binutils/, accessed July 2018

[14] Triton - A DBA Framework, https://triton.quarkslab.com/, accessed July 2018

[15] Zint Barcode Generator, https://github.com/zint/zint, accessed July 2018

[16] Software-artifact Infrastructure Repository, https://sir.unl.edu/portal/bios/tcas.php, accessed July 2018

[17] X. Li, W. E. Wong, R. Gao, L. Hu, and S. Hosono, "Genetic Algorithm-based Test Generation for Software Product Line with the Integration of Fault Localization Techniques," *Empirical Software Engineering*, Vol. 23, No. 1, pp 1-51, 2018

[18] T. Wu, J. Yan, and J. Zhang, "Automatic Test Data Generation for Unit Testing to Achieve MC/DC Criterion," in *Proceedings of IEEE Eighth International Conference on Software Security and Reliability (SERE)*, pp. 118-126, San Francisco, USA, June 2014

[19] S. Rayadurgam and M.P.E. Heimdahl, "Generating MC/DC Adequate Test Sequences Through Model Checking," *SEW*, Vol. 3, pp. 91-26, 2003

[20] M.P.E. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao, "Auto-generating test sequences using model checkers: A case study," in *Proceedings of International Workshop on Formal Approaches to Software Testing,* pp. 42-59, Berlin, Germany, October 2003

[21] D. Li, L. Hu, R. Gao, W. E. Wong, D. R. Kuhn, and R. N. Kacker, "Improving MC/DC and Fault Detection Strength Using Combinatorial Testing," in *Proceedings of IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C),* pp. 297-303, Prague, Czech Republic, July 2017

[22] Z. Awedikian, K. Ayari, and G. Antoniol, "MC/DC automatic test input data generation," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation,* pp. 1657-1664, Montreal, Canada, July 2009

[23] A. El-Serafy, G. El-Sayed, C. Salama, and A. Wahba, "Enhanced genetic algorithm for MC/DC test data generation," in *Proceedings of IEEE 2015 International Symposium on innovations in Intelligent Systems and Applications (INISTA)*, pp. 1-8, September 2015

[24] R. Pandita, T. Xie, N. Tillmann, and J. De Halleux, "Guided test generation for coverage criteria," in *Proceedings of IEEE Iternational Conference on Software Maintenance (ICSM)*, pp. 1-10, Timisoara, Romania, September 2010

[25] S. Godboley, A. Dutta, D. P. Mohapatra, and R. Mall, "Making a concolic tester achieve increased MC/DC," *Innovations in Systems and Software Engineering,* vol. 12, no. 4, pp. 319-332, 2016

[26] T. Su, G. Pu, B. Fang, J. He, J. Yan, S. Jiang, and J. Zhao, "Automated coverage-driven test data generation using dynamic symbolic execution," in *Proceedings of IEEE Software Security and Reliability (SERE)*, pp. 98-107, San Francisco, USA, June 2014