2014 Eighth International Conference on Software Security and Reliability

Automatic Test Data Generation for Unit Testing to Achieve MC/DC Criterion [†]

Tianyong Wu[1,2,4], Jun Yan[1,2,3] and Jian Zhang[3]
1. Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
2. State Key Laboratory of Rail Traffic Control and Safety, Beijing Jiaotong University
3. State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
4. University of Chinese Academy of Sciences
Email: {wutianyong11, yanjun}@otcaix.iscas.ac.cn, zj@ios.ac.cn

***Abstract*—Modified Condition/Decision Coverage (MC/DC) became widely used in software testing, especially in safety-critical domain. However, existing testing tools often aim at achieving statement or branch coverage and do not support test generation for MC/DC. In this paper, we propose a novel test generation method to find appropriate test data for MC/DC. Specifically, we first extract paths from the target program and then find appropriate test data to trigger these paths. In the path extraction process, we propose a greedy strategy to determine the next selected branch. The evaluation results show that our method can actually generate test data quickly and the coverage increases a lot (up to 37.5%) compared with existing approaches.**

***Keywords*-unit testing, test data generation, MC/DC**

I. Introduction

Software testing is one of the most critical activities in the software development process and it is also the primary technique to ensure high software quality. However, thorough testing is often infeasible because of the infinite execution spaces of the software. Therefore, many structural criteria were proposed to alleviate this problem. Since manually written test suites may be insufficient to achieve high coverage and it is often time-consuming, automated test generation has been proposed to reduce human efforts in testing.

For safety-critical software systems that involve complex logical decisions, the basic criteria (like branch coverage) seem to be inadequate because they only consider the structure of the programs. Suppose there is a boolean expression denoted by $(A \wedge B)$ in a decision of a program, where A and B indicate two conditions. For branch coverage, we just need to generate test data such that the value of $(A \wedge B)$ is assigned $true$ and $false$ at least once. The assignments $(true, true)$ and $(false, true)$ for A and B can satisfy this requirement. However, it is easy to notice that the situation where the value of B is assigned $false$ has not been tested. To take logical expressions into account, many complex coverage criteria have been proposed, including condition coverage (CC), decision coverage (DC), condition/decision coverage (C/DC) and multiple condition coverage (MCC). They are all called "logical coverage criteria". For more details, see [20]. However, CC, DC and C/DC now are considered inadequate for testing software embedded with complex logical expressions too, while the number of test cases generated for MCC grows exponentially with the number of conditions in a logical expression.

As a trade-off between test adequacy and cost, Modified Condition/Decision Coverage (MC/DC) was raised by J. J. Chilenski and S. P. Miller in 1994 [6]. Currently, there are many versions of MC/DC, such as weak MC/DC, strong MC/DC. These two versions of MC/DC both aim at generating test data to cover each independent value of conditions in the program. And strong MC/DC also requires only one condition's value changes at one time. It can be shown that strong MC/DC subsumes CC, DC, C/DC and weak MC/DC while the number of test cases does not grow too much (at most twice as many as the number of conditions for each branch). Since we need only a few test cases to satisfy this coverage criterion, this test methodology is very important in software testing. In addition, recent experimental results [7] show that strong MC/DC has better fault detection capability than all of the criteria mentioned above, except for MCC. Therefore, we only discuss strong MC/DC (MC/DC for short) in this paper.

In the early period, the test generation approaches for MC/DC [12] only considered a single decision, and these methods seem to be difficult to be applied to produce test data for a real program automatically. In the recent decade, there are several tools based on symbolic execution [3], [4], [10], [14] that can generate test data automatically. And these approaches all target at statement or branch coverage for test generation. However, MC/DC requires specific combination values for each condition in the predicate decisions, not always resulting in execution of a new branch in the program under test. Therefore, existing test generation approaches can not generate test inputs to achieve high MC/DC. To address this issue, some researchers [9], [13] proposed a method to transform the source code to assist existing test generation approaches that target at statement or branch coverage to achieve MC/DC. But their approaches focus on the test generation for weak

[†]This work is supported by National Natural Science Foundation of China (under grant No. 91118007), National Key Basic Research Program of China (under grant 2014CB340701) and the State Key Laboratory of Rail Traffic Control and Safety (Contract No. RCS2010K006), Beijing Jiaotong University.

MC/DC, which is not sufficient for MC/DC.

In this paper, we propose an approach to generate test data to achieve MC/DC. Specifically, in this method, we first extract paths from the target programs and then find appropriate test data to trigger these paths. In the path extraction process, we propose a greedy strategy to determine the next selected branch, which can increase coverage as fast as possible.

This paper makes the following contributions:

- A novel test data generation method with a greedy strategy to achieve MC/DC.
- A prototype tool based on our test generation framework.
- An evaluation of our method with several benchmarks and real programs. The evaluation results show the coverage of test data generated by our method increases a lot, compared with existing test generation approaches.

The rest of this paper is organized as follows. The next section will give a precise definition of MC/DC criterion and several related concepts used later. The test data generation method will be presented in Section III. And we give an outline of our prototype tool and show our experimental results in Section IV. In Section V, related works are discussed. Finally we give the conclusion and discuss the possible research in the future.

II. Background

In this section, we provide several basic concepts, mainly about MC/DC.

A. Notations and terminology

MC/DC focuses on the logical expressions in the branch statements of a program. A logical expression always consists of some atomic relational expressions (or boolean variables) and zero or more boolean operators, like $not(\neg), and(\wedge), or(\vee),\ xor(\oplus)$. The followings are several definitions about a logical expression.

Definition 1: A **condition** is a primitive logical expression, that is to say, it cannot be broken down into several simpler boolean expressions. A **decision** is a logical expression composed of conditions with zero or more boolean operators.

Definition 2: A **boolean skeleton** for a logical expression is a pure boolean expression (not containing arithmetic and relational operators), which is constructed by replacing each relational expression in this logical expression with a distinct boolean variable.

Combining Definition 1 with Definition 2, we can get that each decision corresponds to a boolean skeleton and each condition can be seen as a boolean variable in the boolean skeleton.

The program evaluates all of conditions in each decision certain boolean outcomes, including true (T) and false (F), on the basis of the test data.

Definition 3: A **condition vector** for a decision is a group of boolean outcomes of all conditions in this decision under some test data.

In fact, a condition vector for a decision can be seen as an assignment to the boolean skeleton for this decision.

B. MC/DC

Hayhurst et al. [12] gave the notions of MC/DC and the formal definitions are shown as follows.

Let $D(C_1, C_2, \ldots, C_n)$ denote a decision, where C_i ($1 \leq i \leq n$) indicates a condition. Let $BS(c_1, c_2, \ldots, c_n)$ be the boolean skeleton for decision D, where c_i ($1 \leq i \leq n$) indicates a boolean variable. Assume that tv_1 = $\langle v_{11}, v_{12}, \ldots, v_{1n}\rangle$ and tv_2 = $\langle v_{21}, v_{22}, \ldots, v_{2n}\rangle$ indicate two condition vectors for D, where v_{11}, …, v_{1n}, v_{21}, …, v_{2n} can be T or F. We define a series of functions $f_i(tv_1, tv_2)$ and the range of f_i is $\{T, F\}$ ($1 \leq i \leq n$). If $v_{1j} = v_{2j}$ ($1 \leq j \leq n\ and\ j \neq i$) and $v_{1i} \oplus v_{2i} = T$, then $f_i(tv_1, tv_2) = T$, otherwise, $f_i(tv_1, tv_2) = F$. The actual meaning of f_i is whether two condition vectors are different only in the ith component.

Definition 4: Two condition vectors tv_1 and tv_2 **MC/DC cover** condition C_i ($1 \leq i \leq n$), if and only if, $f_i(tv_1, tv_2)$ and $BS(v_{11}, v_{12}, \ldots, v_{1n}) \oplus BS(v_{21}, v_{22}, \ldots, v_{2n})$ both hold. In addition, we call these two vectors a **MC/DC pair** for condition C_i.

By the way, if $BS(v_{11}, v_{12}, \ldots, v_{1n}) \oplus BS(v_{21}, v_{22}, \ldots, v_{2n})$ and $v_{1i} \oplus v_{2i}$ both hold, then tv_1 and tv_2 **Weak MC/DC Cover** condition C_i.

Example 1. Consider the following C code. It has three input variables (x, y, z) and a local variable w. Let test data t_1 denote ($x = F, y = T, z = T$), t_2 denote ($x = T, y = T, z = F$), t_3 denote ($x = F, y = T, z = F$), t_4 denote ($x = T, y = F, z = T$) and t_5 denote ($x = T, y = F, z = F$). The outcomes of the conditions in decision $x \wedge (y \vee z)$ under these test data are shown in Table I.

```
bool x, y, z;
int example() {
  int w;
  if(x && (y || z))
    w = 0;
  else w = 1;
  if(w == 0) return 1;
  else return 0;
}
```

Fig. 1: A simple example of C code

TABLE I: Outcomes of conditions and decisions

expression	t_1	t_2	t_3	t_4	t_5
x	F	T	F	T	T
y	T	T	T	F	F
z	T	F	F	T	F
$x \wedge (y \vee z)$	F	T	F	T	F

Let us see the second and third cases, whose condition vectors are $\langle T, T, F\rangle$ and $\langle F, T, F\rangle$, respectively. We can easily get that $f_1(\langle T, T, F\rangle, \langle F, T, F\rangle) = T$ and $(T \wedge (T \vee F)) \oplus (F \wedge (T \vee F)) = T$, so these two condition vectors can be a MC/DC pair for condition x. Similarly, the condition vectors under t_2 and t_5 can be a MC/DC pair for condition y and the condition vectors under t_4 and t_5 can be a MC/DC pair for condition z. Note that the condition vectors under t_1 and t_2 can not make up a MC/DC pair for condition x, since

$f_1(\langle F,T,T\rangle, \langle T,T,F\rangle)$ = F. And the condition vectors under t_1 and t_2 can Weak MC/DC Cover condition x. Therefore, it indicates that the test data set, which satisfies weak MC/DC criterion, may not satisfy MC/DC.

III. Our Test Generation Method

In this section, we present a method to generate test data for MC/DC. Generally, in our test generation method, we first need to translate the target program into a Control Flow Graph (CFG). A CFG may be treated as a directed graph that consists of nodes and directed edges between two nodes and each node represents a linear sequence of assignment statements. Then a program path can also be defined as in graph theory, which is presented as a sequence of nodes or edges in the CFG. However, this definition of program paths does not consider condition vectors for each decision of edges in the path, so we propose a new definition as follows.

***Definition** 5:* A **condition level test path (CLTP)** is a sequence of condition vectors and there exists a program path such that each condition vector of this sequence corresponds to an assignment of boolean skeleton for the decision in relevant position of this path.

Example 2. Continue with the code in Example 1. The condition vectors for decisions $x \wedge (y \vee z)$ and $w == 0$ under test data t_1 are $\langle F,T,T\rangle$ and $\langle F\rangle$ (because $w = 1$). So $(\langle F,T,T\rangle, \langle F\rangle)$ is a CLTP corresponding to test data t_1.

A. Overview of our approach

The main idea of our approach can be divided into the following steps. In the first step, we extract several paths (CLTPs) from the source code. Meanwhile, we determine the feasibility of CLTPs and produce the corresponding test data for the feasible and complete CLTPs. Finally, we reduce the path set without decreasing the coverage so that the cost of testing can be lower. In the first step, since there are often an infinite number of paths in a program which has loops, testing all of these paths is almost impossible. And we expect to find the minimal test data set to satisfy MC/DC as fast as possible. Hence, we propose a greedy strategy to guide this path extracting process.

Figure 2 gives the architecture of our method. It first takes a program as input and parses it into a CFG. Then our method extracts the branch statements in each block of CFG, and identifies the decisions and conditions of each branch statement. With such information, our method can extract CLTPs to MC/DC cover conditions. In the path extraction process, we use a Pseudo-Boolean Optimization (PBO) solver (See section III-C) to assist our method to find valuable CLTPs quickly and reduce the cost. And we invoke a SMT solver to generate test data for each feasible and complete CLTP.

B. Feasible CLTP Extraction

Since a CFG is defined as in graph theory, we can use graph searching algorithms to extract CLTPs for a program, such as Depth-first search (DFS) and Breadth-first search (BFS). In our method, we choose DFS as the main searching method,

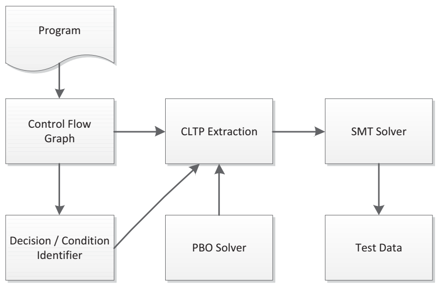

Fig. 2: The architecture of our approach

since the memory space required by BFS is often proportional to the searching space, which is huge or infinite.

In the search process, we start from the entry node of a CFG, and then continuously extend partial CLTP p to a new CLTP with condition vectors. When there are more than one edge starting from the tail node of p and the decisions of these edges have multiple conditions, the assignments (condition vectors) for these decisions also need to be taken into account. These are important issues when searching massive space and it determines how quickly the algorithm can find the test data we need. To solve this problem, we propose a greedy strategy in the next subsection.

Since our goal is to generate test data with these CLTPs, the feasibility of paths is required. Therefore, when we find a partial CLTP in the CFG, we need to check whether this path is feasible or not. If the CLTP is not feasible, we will stop extending this path and discard it because a complete CLTP from entry node to exit node which is prefixed with it must be infeasible. Subsection III-D will introduce how to check the feasibility of a CLTP.

Algorithm 1 shows our method for extracting feasible CLTPs in detail. The input is a CFG G. $\mathcal{P}$ indicates the set of selected CLTPs, p represents the current CLTP that is being searched and st is a stack for DFS. $\mathcal{C}$ denotes the set of all conditions in G, while $\mathcal{C}_{cov}$ indicates the set of conditions that have been MC/DC covered by $\mathcal{P}$. Initially, $\mathcal{P}$, p, st and $\mathcal{C}_{cov}$ are empty, and $\mathcal{C}$ is initialized by all conditions of G. Function $tailnode$ obtains the last node of path p according to the condition vectors in it, and if p has no condition vector, then this function returns the entry node of G. Line 8-14 is the core code of the whole algorithm and it is used to extend the CLTP p to a new CLTP. Unlike the traditional search-based algorithms, we consider the condition vector (outcomes of conditions) of each decision in the path extending process, instead of only considering the outcomes of decisions. Since there are many possible values of the condition vector for a decision, we use the function $getNextTestVector$ to select an appropriate and unvisited condition vector of the decision in s in order to increase the searching speed. The condition vector selection strategy can be seen in the next subsection. If there are no unvisited condition vector in s, this method will return $null$. The upper bound on path length is $maxLen$, which is used to ensure the program can be terminated. $isFeasible$ and

Algorithm 1 Feasible CLTP extraction

```
Input: A CFG $G$
Output: A CLTP set $\mathcal{P}$
1: CLTP $p$ = $\varnothing$; Stack $st$ = $\varnothing$; $\mathcal{P}$ = $\varnothing$;
2: $\mathcal{C}$ = $extractAllConditions(G)$; $\mathcal{C}_{cov}$ = $\varnothing$;
3: $st.push(p)$;
4: while $\mathcal{C}_{cov}$ is not equal to $\mathcal{C}$ and $st$ is not empty do
5:   $p$ = $st.top()$;
6:   $st.pop()$;
7:   $s$ = $p.tailnode()$;
8:   $tv$ = $getNextTestVector(s, \mathcal{P})$;
9:   if $tv$ is not $null$ and $p.size() < maxLen$ then
10:    $p.add(tv)$;
11:  else
12:    continue;
13:  end if
14:  if $isFeasible(p)$ then
15:    if $isComplete(p)$ then
16:      $\mathcal{P}.add(p)$;
17:      for each CLTP $p'$ in $\mathcal{P}$ do
18:        for each condition $c$ in ($\mathcal{C} \setminus \mathcal{C}_{cov}$) do
19:          if the condition vectors in $p$ and $p'$ can
             MC/DC cover $c$ then
20:            $\mathcal{C}_{cov}.add(c)$;
21:          end if
22:        end for
23:      end for
24:    end if
25:    $st.push(p)$;
26:  end if
27: end while
```

$isComplete$ determine whether path p is feasible and whether p is a complete path that begins with the entry node and ends with the exit node in the CFG. Line 18-24 update $\mathcal{C}_{cov}$ when a new CLTP is added into $\mathcal{P}$. Specifically, we enumerate each uncovered condition c and determine whether there exists a CLTP p' in $\mathcal{P}$, whose condition vectors can MC/DC cover c with the condition vectors in p. After getting CLTP set $\mathcal{P}$, we will generate test data that can trigger all CLTPs in $\mathcal{P}$.

C. Condition vector selection strategy

The random method is an intuitive and commonly used strategy to solve the selection problem in the searching process. That is, we select the next condition vector randomly in function $getNextTestVector$. However, the random method utilizes less context information and fewer properties generated by the searching process, which are essential to improve the efficiency of searching process. Therefore, we propose a greedy strategy to overcome this shortcoming of random method so that the searching process can be accelerated. In this greedy strategy, we define an evaluation method to measure the contributions of distinct condition vectors of the decision for MC/DC, and each condition vector will have an evaluation score. These evaluation scores can be used for guiding the searching process. Specifically, we will choose the one which has the highest score, as the next condition vector. The details about this strategy are shown as follows.

Assume that p denotes the current CLTP, $\mathcal{P}$ denotes the selected path set. The decision of tail node of p is denoted as $D(C_1, C_2, \ldots, C_n)$, where C_i $(1 \leq i \leq n)$ indicates the ith condition. Let $BS(c_1, c_2, \ldots, c_n)$ denote the boolean skeleton corresponding to D. Suppose a new condition vector of D is $tv = \langle v_1, v_2, \ldots, v_n \rangle$, where v_i is a boolean variable. The followings are a number of experimental observations about contributions of tv for condition C_i $(1 \leq i \leq n)$.

- **Observation 1** If there are already two condition vectors in path set $\mathcal{P}$ such that these two condition vectors can make a MC/DC pair for C_i, in other words, we do not need any extra CLTPs for MC/DC covering C_i, then this condition vector has no contribution;
- **Observation 2** If $BS(v_1, v_2, \ldots, v_i, \ldots, v_n) \oplus BS(v_1, v_2, \ldots, \neg v_i, \ldots, v_n)$ does not hold, that is to say, this condition vector is impossible to be a MC/DC pair for C_i with any other condition vector for D, then this condition vector has no contribution;
- **Observation 3** If there are no two condition vectors tv_1 and tv_2 in $\mathcal{P}$ such that $f_i(tv_1, tv_2)$ holds, but there exists condition vector tv_3 in $\mathcal{P}$ such that $f_i(tv, tv_3)$ holds, that is to say, it is most likely that tv and tv_3 can make up a MC/DC pair for C_i;
- **Observation 4** If the condition of Observation 1 does not occur and $BS(v_1, v_2, \ldots, v_i, \ldots, v_n) \oplus BS(v_1, v_2, \ldots, \neg v_i, \ldots, v_n)$ holds, then it is somewhat likely that tv can make up a MC/DC pair for C_i with other condition vectors. So this condition vector has some contributions;
- **Observation 5** If the conditions of Observation 3 and Observation 4 occur at the same time, then it indicates that tv actually make up a MC/DC pair for uncovered condition C_i with some condition vector in $\mathcal{P}$, so this condition vector has the greatest contributions.

Based on the above observations, we define several formulae to calculate the evaluation score of condition vector tv for each condition as follows. In these equations, the boolean variables c_{i1}, c_{i2}, c_{i3}, c_{i4}, c_{i5} correspond to the above observations respectively.

$$c_{i1} = \begin{cases} T, \; if\ the\ condition\ of\ Observation\ 1\ occurs \\ F, \; otherwise \end{cases} \tag{1}$$

$$c_{i2} = BS(v_1, v_2, \ldots, v_i \ldots v_n) \oplus BS(v_1, v_2, \ldots, \neg v_i, \ldots, v_n) \tag{2}$$

$$c_{i3} = \begin{cases} T, \; if\ the\ condition\ of\ Observation\ 3\ occurs \\ F, \; otherwise \end{cases} \tag{3}$$

$$c_{i4} = \neg c_{i1} \wedge c_{i2} \tag{4}$$

$$c_{i5} = c_{i3} \wedge c_{i4} \tag{5}$$

In the following Pseudo-Boolean equations, positive integer c_i $(1 \leq i \leq n)$ indicates the score of this condition vector for each condition C_i and it is the sum of c_{i4} and c_{i5}. Integers w_1 and w_2 are two weight values to adjust the contributions, which are tentatively set as 1 and n. Integer c denotes the total evaluation score of condition vector tv for D and it is the sum of c_i.

$$c_i = w_1 \cdot c_{i4} + w_2 \cdot c_{i5} \tag{6}$$

$$c = \sum_{i=1}^{n} c_i \tag{7}$$

Example 3. Consider the CFG in Figure 3. Assume that the selected CLTP set $\mathcal{P}$ has only one element ($\langle F, F\rangle$) and the tail node of current CLTP is $Entry$.

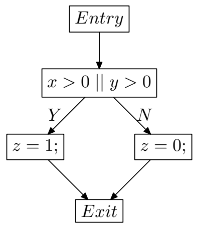

Fig. 3: A sample CFG

Now we need to determine the next condition vector $\langle v_1, v_2\rangle$ of this decision. Since the decision of $Entry$ has two conditions, we need to calculate the value of c_1 and c_2, and then sum up them to obtain the value of c. The values of c_1 and c_2 are shown as follows.

$$\begin{aligned} c_1 &= ((v_1 \vee v_2) \oplus (\neg v_1 \vee v_2)) \\ &\quad + 2 \cdot ((v_1 \wedge \neg v_2) \wedge ((v_1 \vee v_2) \oplus (\neg v_1 \vee v_2))) \\ c_2 &= ((v_1 \vee v_2) \oplus (v_1 \vee \neg v_2)) \\ &\quad + 2 \cdot ((\neg v_1 \wedge v_2) \wedge ((v_1 \vee v_2) \oplus (v_1 \vee \neg v_2))) \end{aligned}$$

And then, we can know that

$$\begin{aligned} c &= ((v_1 \vee v_2) \oplus (\neg v_1 \vee v_2)) \\ &\quad + 2 \cdot ((v_1 \wedge \neg v_2) \wedge ((v_1 \vee v_2) \oplus (\neg v_1 \vee v_2))) \\ &\quad + ((v_1 \vee v_2) \oplus (v_1 \vee \neg v_2)) \\ &\quad + 2 \cdot ((\neg v_1 \wedge v_2) \wedge ((v_1 \vee v_2) \oplus (v_1 \vee \neg v_2))) \end{aligned}$$

Table II shows the evaluation scores of each condition vector. According to it, we should choose $\langle F, T\rangle$ or $\langle T, F\rangle$ as the next condition vector since they have the highest score, instead of $\langle T, T\rangle$. It is a reasonable choice because the first two condition vectors can make one condition satisfy MC/DC requirements with the condition vector of a CLTP in $\mathcal{P}$, while the last one can not.

TABLE II: The evaluation scores of each condition vector in Example 3

Condition vector	$\langle F, T\rangle$	$\langle T, F\rangle$	$\langle T, T\rangle$
c	3	3	0

Through the above example, we can see that the evaluation score c will be represented as an expression, which is only composed of the condition vector values with several logical and arithmetical operators. Our goal is to discover a condition vector of conditions to maximize the score. Quite naturally, this problem can be described as a Pseudo-Boolean Optimization (PBO) problem [15] as follows. The constraints of this problem require that each condition vector is selected by $getNextTestVector$ at most once.

$$\begin{aligned} &maximize: && c = \textstyle\sum_{i=1}^{n} c_i \\ &subject\ to: && test\ vector\ \langle v_1, v_2, \ldots v_n\rangle\ has\ not\ been \\ & && selected\ by\ function\ getNextTestVector \end{aligned}$$

D. Path feasibility checking and test data generation

After extracting a CLTP from a CFG, the next thing we need to do is to check its feasibility, and generate test data if it is a complete path. A CLTP of CFG can be regarded as a program fragment and the goal of producing test data for it is to find test data such that it can trigger this CLTP. In our previous work [18] [19], we have used a method based on symbolic execution and constraint solving, to solve this problem. The only difference here is that we divide each decision into several conditions and each decision has a condition vector, thus we need consider this difference when constructing the constraints for them. Specifically, if each component in the condition vector, which corresponds to a condition, is assigned TRUE, then its constraint is itself; otherwise, the constraint is the negation of the condition. For instance, continue with Example 3, assume the condition vector value is $\langle T, F\rangle$, then the constraints of this decision are $(x > 0)$ and $\neg(y > 0)$.

E. Path set reduction

Definition 6: A CLTP p is a ***redundant CLTP*** in path set $\mathcal{P}$ ($p \in \mathcal{P}$), if and only if the coverage of $\mathcal{P}$ is the same as that of $\mathcal{P}^*$, where $\mathcal{P}^*$ = $\mathcal{P}$ - $\{p\}$.

Definition 7: A CLTP set $\mathcal{P}$ is a ***reduced CLTP set*** if and only if there is no redundant CLTP in $\mathcal{P}$.

In this subsection, we briefly describe a method for translating a CLTP set to a reduced CLTP set. For more details, see [16]. The path set reduction problem can be simply reduced to the set cover problem (SCP) [24]. For instance, let $\mathcal{P}$ indicate the CLTP set and $\mathcal{C}$ denote the condition set in the program. Then we just consider C as the universal set and regard each condition which is MC/DC covered by two paths in $\mathcal{P}$ as subsets. The SCP can be formulated as a PBO problem as follows:

$$\begin{aligned} &minimize: && \textstyle\sum_{i=1}^{|\mathcal{P}|} x_i \\ &subject\ to: && \textstyle\sum_{i=1}^{|\mathcal{P}|} \sum_{j=i}^{|\mathcal{P}|} x_i \cdot x_j \cdot \delta_{ijk} \geq 1 \quad (1 \leq k \leq |\mathcal{C}|) \end{aligned}$$

where x_i is a boolean variable that indicates whether the ith path is selected and δ_{ijk} is also a boolean variable which denotes whether the corresponding condition vector of ith and jth CLTP make a MC/DC pair for the kth condition.

IV. Implementation and Experimental Results

We developed a tool based on our approach to generate test data satisfying MC/DC for a program. Since we only focus on test data generation for unit testing, if the programs under test have function calls, we inline all the subfunctions in their main functions. The frontend of our tool uses Clang [21] to translate a source code into a CFG. In the backend, we implemented the algorithm, including CLTP extraction and test data generation, in Java. In the path extraction phase, we used clasp [22] to solve the PBO problems, which determine the next condition vectors. In the test data generation phase, we employed an SMT solver Z3 [26] to solve constraints.

Before giving experimental results of our method, we show the style of our target programs and test data by a simple example. $Triangle$ is a classical program in software testing area, since it has just a few lines of code but its program structure and branches are quite complex. It has many versions, we use the version presented by N. Williams et al. [17], because this version owns more conditions in each decision in the whole program, which MC/DC focuses on. It has three integer input variables that denote the length of three sides in a triangle. The goal of this code is to find the type of this triangle, including $not\ triangle$, $isosceles\ triangle$, and $equilateral\ triangle$ and so on. The code of $Triangle$ can be seen in Figure 4. We applied our tool to this instance, and it found 14 test cases (with input data) to achieve MC/DC, which can be seen in Table III and they can MC/DC cover all the conditions in this program.

```
int tritype(int i, int j, int k){
  int type_code;
  if ((i == 0) || (j == 0) || (k == 0))
    type_code = 4;
  else {
    type_code = 0;
    if (i == j) type_code = type_code + 1;
    if (i == k) type_code = type_code + 2;
    if (j == k) type_code = type_code + 3;
    if (type_code == 0){
      if ((i+j <= k) || (j+k <= i) || (i+k <= j))
        type_code = 4;
      else
        type_code = 1;
    }
    else if (type_code > 3)
      type_code = 3;
    else if ((type_code == 1) && (i+j > k))
      type_code = 2;
    else if ((type_code == 2) && (i+k > j))
      type_code = 2;
    else if ((type_code == 3) && (j+k > i))
      type_code = 2;
    else type_code = 4;
  }
  return type_code;
}
```

Fig. 4: The C code of Triangle

To study the effectiveness of our approach, we raise two evaluation factors (EF) as follows.

- **EF1:** Compared with existing test generation approaches, which target at achieving branch coverage, what is the increase in MC/DC coverage by the test data generated with our approach?
- **EF2:** For our test generation method, is the greedy strategy for selecting next condition vector effective? That is, can this strategy speed up the generation process and decrease the number of test cases?

TABLE III: Test data for Triangle generated by our tool

Test Data number	Test Data	Test Data number	Test Data
#1	(0, 1, 1)	#8	(1, 2, 2)
#2	(1, 0, 1)	#9	(2, 1, 1)
#3	(1, 1, 0)	#10	(2, 3, 2)
#4	(1, 3, 2)	#11	(1, 2, 1)
#5	(1, 2, 3)	#12	(2, 2, 3)
#6	(2, 3, 4)	#13	(1, 1, 2)
#7	(3, 1, 2)	#14	(1, 1, 1)

A. Subject programs

We totally used 8 programs, including 6 benchmarks and 2 custom-written programs, to evaluate our approaches. The first program $CalDate$ is taken from [8]. This program converts the special date (given as three arguments: day, month, year) into a Julian date. The second program is $Quadratic$ [8], which calculates the root of a quadratic equation. The third program is $Triangle$ [17]. $Showhand$ is a custom-written program that describes a popular card game Showhand. $CreateNumber$ is C code translated from Java Class *org.apache.commons.lang.math.NumberUtils*. $Tcas$ [25] is an aircraft collision avoidance system designed to reduce the incidence of mid-air collisions between aircraft. $Space\_manage$ is a program supplied by our industry partner. The last one $Fieldbackch$ is a function from the source code $field.c$. The common character of these programs is that they all have many complex decisions, each of which has a number of conditions. Table IV shows some statistics of these programs, containing the number of lines, decisions and conditions of each program.

TABLE IV: The statistics of experimental programs

Program	**#Line**	**#Decison**	**#Conditon**
CalDate	37	6	12
Quadratic	48	6	10
Triangle	60	8	17
Showhand	88	9	28
CreateNumber	150	16	30
Tcas	174	7	14
Space_manage	103	8	16
Fieldbackch	90	16	33

B. Evaluation Setup

Since existing test generation approaches aim at achieving branch coverage, there are many tools to measure the achieved branch coverage ratio of test data set, like Gcov [23]. However, our goal for EF1 requires the statistics of the achieved MC/DC coverage ratio of test data set and there are currently no specific tool to address this requirement. Therefore, to obtain the achieved coverage, we insert several statements into each branch to collect the condition vector and outcome of this branch under some test data. Then we execute the programs with the test data generated by our approaches as input and get the condition vectors and outcomes of all decisions in the programs under these test data. Finally, we combine any two condition vectors and determine whether they MC/DC cover some condition.

For EF1, we first count the number of conditions (denoted by c) in the programs. And then we execute the programs with test data generated by an existing tool (CREST) and our test generation method. Finally, we measure the number of conditions (denoted by c_1, c_2) that can be MC/DC covered by the test data, which are generated by CREST and our approach, respectively. In addition, we define the coverage ratio cr_1, cr_2 as the percentage of coverage under the test data generated by the above two methods, where $cr_1 = \frac{c_1}{c}$, $cr_2 = \frac{c_2}{c}$.

For EF2, to evaluate the effectiveness of our generation approach, we implemented another version of our tool, which uses a random search strategy in the path extension process. That is to say, the next condition vector is randomly selected in this version. We measure the number of constraint solving calls (denoted by csc_1 and csc_2) and the next condition vector determination (denoted by tvd_1 and tvd_2) in the test data generation process with random strategy and greedy strategy for each program, respectively. In addition, we count the original number of CLTPs generated by random and greedy strategy (denoted by sz_1 and sz_3) and the number of CLTPs generated by random and greedy strategy after reduction (denoted by sz_2 and sz_4).

C. Results for EF1

Table V shows our experimental results about the achieved coverage ratio of test data generated by the existing test data generation method (CREST) and our method. The first column shows the name of program. The second column shows the number of conditions in each program. The third column shows the number of conditions, which are MC/DC covered under test data generated by CREST. The fourth column shows the number of conditions MC/DC covered by test inputs, which are generated by our method. The last column gives the execution time for each program. All the instances are run on an Intel Core i7 PC with 2.93 GHz CPU and 3 GB memory.

TABLE V: Results for EF1

Program	c	c_1 (cr_1)	c_2 (cr_2)	Time(s)
CalDate	12	11 (91.7%)	12 (100%)	3.28
Quadratic	10	9 (90%)	10 (100%)	1.03
Triangle	17	14 (82.4%)	17 (100%)	1.75
Showhand	28	23 (82.1%)	28 (100%)	6.50
CreateNumber	30	21 (70.0%)	30 (100%)	15.76
Tcas	14	6 (42.9%)	10 (71.4%)	2.18
Space_manage	16	10 (62.5%)	16 (100%)	9.76
Fieldbackch	33	30 (90.9%)	33 (100%)	6.55

From Table V, we can see that the test data generated by CREST actually do not MC/DC cover all conditions in each program. And the test data generated by our method can always achieve a higher coverage ratio, which are often 100% for most subject programs. For the benchmark $Tcas$, the coverage ratio only reaches 71.4% under test inputs generated by our methods. We analyzed the source code and found the reason is that there exists an infeasible decision in the $Tcas$ program, because the boolean variables $need\_upward\_RA$ and $need\_downward\_RA$ can not be true at the same time.

D. Results for EF2

Table VI shows our experimental results about the effectiveness of greedy strategies in the test data generation method. The first column shows the name of program. The second column to fifth column show the number of constraint solving calls (csc_1), the number of next condition vector determination (tvd_1), the size of CLTP set generated initially (sz_1) and the size of CLTP set after reduction sz_2 in the test generation procedure with random strategy. And the last four columns $(csc_2,\ tvd_2,\ sz_3\ and\ sz_4)$ show these information in the test generation procedure with greedy strategy. In the previous section, we mention that the number of test data for MC/DC is at most twice as many as the number of conditions for each branch. To observe the values of sz_2 and sz_4 (5th column and 9th column), we can notice that this conclusion indeed holds and all the number of test data generated by our method are almost close to the number of conditions in our experiments. In addition, from this table, we can see that the values of csc_2 and tvd_2 in the greedy strategy are less than the values of csc_1 and tvd_1 in the random strategy. For instance, in $Showhand$, the value of csc_2 decreases to 67.2% of csc_1 and the value of tvd_2 decreases to 73.8% of tvd_1. Furthermore, compared with the random strategy, sz_3 and sz_4 in the greedy strategy are also smaller. For example in $Quadratic$, they decrease to 58.8% and 76.2% of sz_1 and sz_2.

V. Related works

Automatic test data generation is a very important topic in software testing. There are many methods for doing this, such as SBSE [11], random testing and DSE [3], [4], [10], [14]. However, all of them aim at achieving high statement or branch coverage. In order to implement MC/DC criterion, K. Hayhurst et al. [12] and J. R. Chang and C. Y. Huang [5] raised several methods. However, these methods only consider single decisions, not a complete program.

Z. Awedikian et al. [1] and K. Ghani et al. [8] used evolutionary testing (ET) method to automatically generate test data achieving MC/DC for a real program. However, their approach faces the drawback of local maxima. In addition, they do some experiments on the programs $CalDate$, $Quadratic$ and $Triangle$. The execution time for these programs are in the range between 16 seconds to 38 seconds, while our execution time are all within 4 seconds.

Pandita et al. [13] and S. Godboley [9] used code transformation to guide existing test generation approach to achieve high MC/DC. However, they did not give formal conversion rules and their approaches only focus on the test generation for weak MC/DC.

VI. Conclusion and Future work

We discuss the test data generation for MC/DC in this paper. Since the existing approaches do not support effective test

TABLE VI: Results for EF2

Program	Random				Greedy			
	csc_1	tvd_1	sz_1	sz_2	csc_2	tvd_2	sz_3	sz_4
CalDate	156.5	77.1	33.9	11.2	124	61	25	11
Quadratic	140.9	78.89	13.6	9.19	106	58	8	7
Triangle	167.3	74.4	31.1	14.3	159	60	29	14
Showhand	646.9	363.3	82.7	24.1	435	268	56	22
CreateNumber	596.2	368.0	204.3	32.4	513	330	160	30
Tcas	309.3	108.7	36.5	9.6	255	103	28	8
Space_manage	642.0	139.0	80.0	16.5	467	133	60	16
Fieldbackch	926.4	224.3	124.7	35.2	750	144	93	32

generation for this criterion, we present an automatic method to produce test data that achieve MC/DC. Experimental results show that the coverage of the test data generated by our approaches actually increases a lot, compared with existing approaches. Currently, our methods cannot deal with programs with function calls. Fortunately, our techniques can be easily extended into the dynamic methods. Hence we will try to improve our tool with dynamic test generation techniques to support testing larger programs.

References

[1] Z. Awedikian, K. Ayari and G. Antoniol. MC/DC Automatic Test Input Data Generation. In *Proceeding of 11th Annual conference on Genetic and Evolutionary Computation*, 2009, pp. 1657-1664.

[2] A. Bertolino and M. Marre. Automatic Generation of Path Covers based on the Control Flow Analysis of Computer Programs. *IEEE Transaction on Software Engineering*, vol. 20, no. 12, pp. 885-899, 1994.

[3] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443-446.

[4] C. Cadar, D. Dunbar and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX conference on operating systems design and implementation*, 2008, pp. 209-224.

[5] J. R. Chang and C. Y. Huang. A Study of Enhanced MC/DC Coverage Criterion for Software Testing. In *Proceeding of 31st Annual International Computer Software and Applications Conference*, 2007, pp. 457-464.

[6] J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, vol. 9, no. 5, 193-200, 1994.

[7] C. R. Fang, Z. Y. Chen and B. W. Xu. Comparing Logic Coverage Criteria on Test Case Prioritization. *Science China Information Science*, vol. 55, pp. 2826-2840, 2012.

[8] K. Ghani and J. A. Clark. Automatic Test Data Generation for Multiple Condition and MCDC Coverage. In *Proceeding of Fourth International Conference on Software Engineering Advances*, 2009, pp. 152-157.

[9] S. Godboley, G. S. Prashanth, D. P. Mohapatro and B. Majhi. Increase in Modified Condition/Decision Coverage Using Program Code Transformer. In *Proceeding of Third International Advance Computing Conference (IACC)*, 2013, pp. 1400-1407.

[10] P. Godefroid, N. Klarlund and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213-223.

[11] M. Harman, S. A. Mansouri and Y. Y. Zhang. Search-based Software Engineering: Trends, Techniques and Applications. *ACM Computing Surveys (CSUR)*, vol. 45, 2012.

[12] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski and L. K. Rierson. A Practical Tutorial on Modified Condition/Decision Coverage. *Report NASA/TM-2011-210876, NASA*, 2001.

[13] R. Pandita, T. Xie, N. Tillmann and J. d. Halleux. Guided Test Generation for Coverage Criteria. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1-10.

[14] K. Sen, D. Marinov and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 263-272.

[15] H. M. Sheini and K. A. Sakallah. Pueblo: A Hybrid Pseudo-boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 165-189, 2006.

[16] H.S. Wang, S.R. Hsu and J.C. Lin. A Generalized Optimal Path-selection Model for Structural Program Testing. *Journal of Systems and Software*, vol. 10, pp. 55-63, 1989.

[17] N. Williams, B. Marre, P. Mouy and M. Roger. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *Proceeding of 5th European Dependable Computing Conference*, 2005, pp. 281-292.

[18] Z. X. Xu and J. Zhang. A Test Data Generation Tool for Unit Testing of C Programs. In *Proceeding of 6th International Conference on Quality Software*, 2006, pp. 107-116.

[19] J. Zhang, C. Xu and X. Wang. Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques. In *Proceeding of SEFM*, 2004, pp. 242-250.

[20] H. Zhu, P Hall and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[21] "clang: a C language family frontend for LLVM", http://clang.llvm.org/.

[22] "Clasp", http://www.cs.uni-potsdam.de/clasp/.

[23] "Gcov", http://en.wikipedia.org/wiki/Gcov.

[24] "Set cover problem", http://en.wikipedia.org/wiki/Set_cover_problem.

[25] Traffic collision avoidance system, http://en.wikipedia.org/wiki/Traffic_collision_avoidance_system.

[26] "Z3: An Efficient Theorem Prover", http://research.microsoft.com/en-us/um/redmond/projects/z3/index.html.

Appendix

In this Appendix, we give a concise proof to explain the correctness of Algorithm 1. For a program, let property (†) denote that there does not exist a CLTP set that satisfies MC/DC for this program while the length of CLTPs in it are all less than $maxLen$. Property (‡) indicates the negation of property (†). Let $p_1 = tv_1tv_2 \cdots tv_n$ denote a CLTP, where tv_i $(1 \leq i \leq n)$ indicates a condition vector. And p_2 is a CLTP, which is constructed by deleting the last condition vector from p_1, that is, $p_2 = tv_1tv_2 \cdots tv_{n-1}$.

***Theorem** 1:* If property (†) holds and p_1 is feasible and $n \leq maxLen$, then p_1 must be found in Algorithm 1.

PROOF. Let us prove this theorem by induction on n.

Base step: If n = 0, that means there is no condition vector in p_1, then the theorem obviously holds.

inductive step: Assume that the theorem holds when $n \leq k$ $(k < maxLen)$. If $n = k + 1$ and (1) p_1 is feasible and (2) $n \leq maxLen$, then we want to prove that p_1 must be found in Algorithm 1. Since the property (1) holds, we can get that p_2 is feasible. According to formula (2), we obtain that $(n-1) = k < maxLen$, so the induction hypothesis can be applied to p_2. That is, p_2 must be found by Algorithm 1. Since condition vector tv_n will be selected once by function $getNextTestVector$, p_2 will be extended to be p_1. Since the property (1) holds, the branch expression in Line 15 is $true$, so p_1 is found.

***Theorem** 2:* If property (†) holds, then $\mathcal{P}$ will contain all feasible and complete CLTPs whose length are less than $maxLen$.

PROOF. With Theorem 1, we can find all feasible CLTPs whose length are less than $maxLen$. If a feasible CLTP is also complete, the branch in Line 16 is $true$ and this CLTP will be added into $\mathcal{P}$.

***Theorem** 3:* If property (‡) holds, then Algorithm 1 must be able to find a CLTP set that satisfies MC/DC.

PROOF. Assume that our approach can not find a CLTP set to satisfy MC/DC criterion. With theorem 2, we can get that our approach will find all feasible and complete CLTPs whose length are less than $maxLen$, and these CLTP cannot satisfy MC/DC. There is a contradiction, thus this conclusion holds. At this point, we already proved the correctness of our approach.