Optimal Test Suite Generation for Modified Condition Decision Coverage using SAT solving

Takashi Kitamura[1], Quentin Maissonneuve[1,2], Eun-Hye Choi[1], Cyrille Artho[3], and Angelo Gargantini[4]

[1] Nat. Inst. of Advanced Industrial Science and Technology (AIST), Japan
{t.kitamura,e.choi}@aist.go.jp
[2] University of Nantes, France, quentin.maisonneuve1@etu.univ-nantes.fr
[3] KTH Royal Institute of Technology, Sweden, artho@kth.se
[4] Università di Bergamo, Italy, angelo.gargantini@unibg.it

Abstract. Boolean expressions occur frequently in descriptions of computer systems, but they tend to be complex and error-prone in complex systems. The modified condition decision coverage (MCDC) criterion in system testing is an important testing technique for Boolean expression, as its usage mandated by safety standards such as DO-178 [1] (avionics) and ISO26262 [2] (automotive). In this paper, we develop an algorithm to generate optimal MCDC test suites for Boolean expressions. Our algorithm is based on SAT solving and generates minimal MCDC test suites. Experiments on a real-world avionics system confirm that the technique can construct minimal MCDC test suites within reasonable times, and improves significantly upon prior techniques.

1 Introduction

Boolean expressions occur frequently in descriptions of computer systems, and are prevalent in software and hardware artifacts, such as programs, specifications, and hardware descriptions. On the other hand, they tend to be complicated and thus error-prone. *Boolean expression testing* is a technique to effectively test such complicated Boolean expressions. Modified Condition Decision Coverage (MCDC), a. k. a., Active Clause Coverage (ACC), is one of main coverage criteria for testing Boolean expressions. The MCDC criterion requires that each condition in a decision is shown by execution to independently affect the outcome of the decision, where “decision” means “Boolean expression” [3].

To demonstrate the notion, take the following Boolean expression: $\phi_1 = s \wedge (t \vee u)$ (1). The test suite in Table 1 consisting of the four test cases satisfies the MCDC criterion for this Boolean expression. Observe that, for example, the pair of the first and third test cases confirms that condition s independently affects the outcome of the expression: the value of condition s changes the

Table 1. A MCDC test suite for $\phi_1 = s \wedge (t \vee u)$.

No.	s	t	u	ϕ_1
1	1	1	0	1
2	1	0	0	0
3	0	1	0	0
4	1	0	1	1

value of ϕ_1 while the values of the other conditions, i. e. t and u, remain unchanged. Observe similarly that the first and second test cases confirm that for t, and the second and fourth confirm that for u. Boolean expressions such as (1) often represent abstract versions of actual logical expressions used in software artifacts; e. g., the following logical expression, $(z > 1) \wedge (f(x) \vee (x < y + 1))$, can be represented by (1), where the conditions $(z > 1)$, $f(x)$, and $(x < y + 1)$ are abstracted by s, t, and u respectively.

The usage of the MCDC criterion is motivated by several rationales.

1. MCDC can detect an erroneous use of *and* for *or* (or vice versa) in a Boolean expression; this is called "Operator Reference Fault (ORF)" [4].
2. MCDC is a stricter form of decision coverage than decision coverage, which stipulates each decision must evaluate to *true* and *false*.
3. The size of a test suite to satisfy the MCDC criterion (MCDC test suite, for short) is reasonably small, and hence testing with MCDC incurs a reasonable test cost. For a Boolean expression ϕ with n conditions, the size of its minimal MCDC test suite is $n + 1$ (e. g., [4], [5], [6]), instead of 2^n required for exhaustive testing. (This will be explained more in details in Section 2.2.)

Thus, MCDC effectively detects faults at reasonable cost. Due to such practical rationales, MCDC has been widely used. Various safety standards mandate its use in testing safety-critical components, such as DO-178 [1] (avionics) and ISO26262 [2] (automotive).

Driven by such industrial demands, MCDC has been actively studied from various aspects, e. g., development of variants of MCDC [4,7], model-based testing with MCDC [7], coverage-driven test generation (a. k. a., CDTG) for program codes [8], empirical studies on its effectiveness [9,10,11], and testing Deep Neural Networks (DNNs) with MCDC [12]. Among them, test case generation for MCDC for Boolean expressions, which, given a Boolean expression, to find its MCDC test suite, is a basic function for such testing techniques with MCDC, and thus has been an important subject. Jones and Harrold developed algorithms for MCDC test reduction [13]. Arcaini et al. [14] developed a Boolean expression testing framework to construct a small test suite including MCDC. Bloem et al. [7] proposed yet another approach for MCDC test generation in developing a model based testing technique using MCDC.

In this paper, we develop an algorithm to construct small or even minimum MCDC test suites; i. e., given a Boolean expression ϕ with n conditions, generate an MCDC test suite whose size is equal to or smaller than $n + 1$. To realize such an algorithm, we use SAT solving as the key technique. To evaluate the proposed technique, we conduct experiments where we apply it to a real-world avionics system in comparison with state-of-the-art techniques. The experiments confirm that the technique can construct minimal MCDC test suites within reasonable times, and improves significantly upon prior techniques.

This paper is organized as follows. The next section states the problem that we tackle. Section 3 describes technical details of the proposed technique. Section 4 reports our evaluation of the proposed technique via experimental results.

Section 5 states the significance of the work w. r. t., existing studies. Section 6 mentions possible future directions of this work.

2 Preliminaries

2.1 Definitions and Problem Formulations

In this subsection, we define the notion of ACC more formally. It is important to note that there have been proposed several variants of ACC, including General Active Clause Coverage (GACC), Restricted Active Clause Coverage (RACC), Correlated Active Clause Coverage (CACC), General Inactive Clause Coverage (GICC), Restricted Inactive Clause Coverage (RICC) [4,15], Reinforced Condition Decision Coverage (RCDC) [11,16], and Observable Modified Condition Decision Coverage (OMCDC) [17].

RACC is the classical and traditional form of ACC testing. On the other hand, recently, CACC has become known to be more practical in industrial usage. In this paper, we thus focus on RACC and CACC, but note that our proposed techniques in this paper can be applied to the other variants of ACC.

In the following, we define RACC and CACC. First, as in this paper we aim to develop ACC test generation techniques, we handle logical formulas as the target object that our techniques process; however, we mainly deal with them in the form of Boolean expressions by abstracting conditions in logical formulas by Boolean variables. This is as exemplified by the example in Section 1. We call Boolean variables in Boolean expressions *conditions*. We also use the term *predicate* as a shorthand for Boolean expression.

Next we introduce the notion of *determination*:

Definition 1 (Determination). *Let ϕ be a predicate and c be a condition of ϕ. We say that condition c determines ϕ, if all the other conditions in ϕ have values so that changing the truth value of c changes the truth value of ϕ.*

ACC stipulates that each condition of predicate ϕ independently determines the result of ϕ. Thus, it is convenient to name the condition on which we are focusing as the *major condition*, and all of the other conditions are *minor conditions*. Formally, for a predicate ϕ with n conditions $c_1, c_2, \cdots, c_n$, and a condition c_i in ϕ, i. e., $i \in \{1 \cdots n\}$ called the *major condition*, conditions c_j such that $j \in \{1 \cdots n\} \wedge (i \neq j)$ are called *minor conditions*.

Definitions of test cases and test suites are also provided as follows:

Definition 2 (Test case and Test suite). *A test case of a predicate ϕ is a possible truth assignment for all the conditions in ϕ. A test suite is a set of test cases.*

The definitions of RACC and CACC are now ready as follows:

Definition 3. *[RACC and CACC] Let ϕ be a predicate and T be a test suite. A* CACC pair *of a condition c of ϕ is a pair of test cases of ϕ, satisfying that (1)* *condition c determines ϕ as the major condition in both test cases and (2) the values of c are different. An* RACC pair *is a CACC pair such that the values for the minor conditions are same in the two test cases. We say that T covers a condition of ϕ in RACC (resp. CACC), if it includes an RACC (resp. CACC) pair of the condition. The test requirement of RACC (resp. CACC) for ϕ is that T should cover all the conditions of ϕ. RACC (resp. CACC) is a measure to describe the degree to which test requirements of RACC (resp. CACC), i. e., how many conditions, are covered by T.*

Note that RACC only differs from CACC in that the former requires the minor conditions of a test case pair have to have the same values, while the latter does not. Therefore, we can say RACC is stricter criterion than CACC; i. e., a RACC test suite for a predicate ϕ is also a CACC for it, but not vice versa. We call a test suite satisfying RACC (resp. CACC) test requirements a *RACC (CACC) test suite*.

2.2 Sizes of Minimum ACC Test Suites

The size of a test suite often plays a critical factor for the feasibility and success of real-world (software and hardware) system testing, and has been a main concern in theory and practice of testing techniques. A main reason for it is that "full automation" of test case execution, despite of a large body of research on it, is still largely limited in practice due to problems such as the *oracle problem*.[5] Another reason is that execution of one test case, even if automated, is often too expensive for allowed time and/or financial constraints; and thus the number of test cases that can be executed is limited.

Regarding properties on the ACC test suite size, it is often mentioned in literature that the size of a minimum ACC test suite is "around" $n + 1$, for a predicate with n conditions:

1. "*Achieving MCDC requires, in general, a minimum of* $n + 1$ *test cases for a decision with* n *inputs.*", by Hayhurst et al. [5], where "decision" here means "predicate" in our setting.
2. "*MCDC requires for* n *input variables a test suite at least of size* $n + 1$*.*", by Felbinger et al. [6], where 'variables' here mean 'conditions' in our setting.
3. "*It turns out that for a predicate with* n *clauses,* $n + 1$ *distinct test requirements, rather than the* $2n$ *one might expect, are sufficient to satisfy active clause coverage.*", by Amman and Offutt [4], where 'clause' here means 'condition' in our setting.

These claims are similar but not equivalent: the first two claims state that at least $n + 1$ test cases are required to for an ACC test suite for a predicate with n conditions, while the third one states $n + 1$ test cases are sufficient. Although it can be conjectured from these that the minimum sizes of ACC test suites exist around $n + 1$, according to our best knowledge, a formal proof of this property is still an open problem.

[5] The challenge of distinguishing the corresponding desired and correct behavior from potentially incorrect behavior given a test case for a system under test.

2.3 Boolean Satisfiability Problem (SAT) and SAT Solvers

The *Boolean satisfiability problem (SAT)*, known to be an NP-complete problem, is the problem of determining if there exists an interpretation, a. k. a., model, that satisfies a given Boolean expression. *SAT solvers* are software tools that automatically solve SAT problems. As many NP-complete problems can be reduced to SAT, the development of efficient SAT solvers is an important and active research subject. We use SAT solvers for our ACC test generation.

3 Algorithm for Optimal MCDC Test Generation

In this section, we develop an algorithm for generating an optimal test suite for MCDC. We explain our algorithm by first introducing the notion of Predicate Determining Condition (PDC), based upon which we develop our algorithm.

3.1 Predicate Determining Condition (PDC)

We explain the notion of *predicate determining condition (PDC)*, introduced in [4], which we use in developing our ACC test generation algorithm.

Definition 1 provided the definition of determination. PDC, given the target predicate ϕ and the major condition of ϕ, provides the condition, as a logical formula for minor conditions, under which the major condition c determines ϕ.

Definition 4 (Predicate determining condition: PDC). *Let ϕ be a predicate ϕ and c be a condition in ϕ.* The predicate determining condition (PDC) *of a predicate ϕ and a condition c, denoted by ϕ_c, is a predicate under which the value of c determines that of ϕ.*

Since a predicate expresses a set of test cases, the PDC of condition c can be also interpreted as all test cases for which c determines ϕ.

An advantage of PDCs is that we can compute them. Given a predicate ϕ and a condition c, the PDC is computed as follows:

Proposition 1 *Let $\phi_{c=\textsc{True}}$ (and $\phi_{c=\textsc{False}}$) represent the predicate ϕ with every occurrence of c replaced by '*True*' (and '*False*'), respectively. Neither $\phi_{c=\textsc{True}}$ nor $\phi_{c=\textsc{False}}$ has any occurrences of condition c. The PDC of condition c for a predicate ϕ is obtained by connecting the two expression with exclusive-or:*

$$\phi_c = \phi_{c=\textsc{True}} \oplus \phi_{c=\textsc{False}} \tag{2}$$

This proposition holds, as the requirement that condition c independently affects the outcome of predicate ϕ can be described as the following formula, which corresponds to (2):

$$(\phi_{c=\textsc{True}} \land \neg\phi_{c=\textsc{False}}) \lor (\neg\phi_{c=\textsc{True}} \land \phi_{c=\textsc{False}}),$$

Example 1. Let ϕ be $\phi_1 = s \wedge (t \vee u)$ of Boolean expression (1). The PDC of predicate ϕ and condition s, i. e., ϕ_s, can be derived as follows:

$$\begin{aligned}\phi_s &= \phi_{s=\textsc{True}} \oplus \phi_{s=\textsc{False}}\\ &= (\textsc{True} \wedge (t \vee u)) \oplus (\textsc{False} \wedge (t \vee u))\\ &= (t \vee u) \oplus \textsc{False}\\ &= (t \vee u)\end{aligned}$$

This means that for the major condition s to determine predicate ϕ, there are three choices for truth assignments, (t, u), $(t, \neg u)$, and $(\neg t, \neg u)$.

We remark that if the PDC of a predicate and a condition is not satisfiable, then this means that it is not feasible to create an ACC test pair for that condition. We call such a condition an *infeasible condition.*

3.2 SAT-encoding

The key device to develop our algorithm is the SAT-encoding of the following problem: "For a predicate ϕ and the number of test cases k, is it possible to construct an ACC test suite?". We explain about the SAT-encoding of this problem in this subsection. The encoding uses a *matrix model* with p columns and n rows as Boolean conditions x_p^n to encode the test suite, as follows:

$$X = \begin{pmatrix} x_1^1 \dots x_p^1 \\ \vdots \qquad \vdots \\ x_1^k \dots x_p^k \end{pmatrix}$$

This matrix model of a test suite expresses that the i-th row represents the i-th test case, and the p-th column of the row represents the value of the p-th parameter of the i-th test case.

Using the matrix model to represent a test suite, we construct the SAT encoding to construct an ACC test suite with the size k, as follows:

$$encode(\phi, conds_\phi, k) = \underbrace{\bigwedge_{c \in conds_\phi}}_{(1)} \underbrace{\bigvee_{i=1}^{k} \bigl((x_c^i \wedge \phi_c^i)}_{(2)} \wedge \underbrace{\bigvee_{j=1, i \neq j}^{k} \bigl((\neg x_c^j \wedge \phi_c^j)}_{(3)} \wedge \underbrace{\bigwedge_{d \in conds_\phi \ s.t. \ c \neq d} (x_d^i \Leftrightarrow x_d^j)}_{(4)} \bigr)\bigr)$$

where ϕ_c^i is PDC of condition c whose Boolean conditions are those in the i-th row of the matrix model, and $conds_\phi$ is a set of feasible conditions in ϕ. This SAT-encoding specifies the following: Sub-formula (2) specifies that for a condition of the predicate, say c, in at least one row of the matrix table, i. e., in one test case of the test suite, the value of the condition in the test case, x_c, must be '$\textsc{True}$', while the PDC of the condition c must be also true. On the other hand, sub-formula (3) specifies that the same condition must be '$\textsc{False}$', while the PDC of

c must be 'TRUE', in at least one test case. The $i \neq j$ in formula (3) clarifies that the value of condition c cannot be 'TRUE' and 'FALSE' in the same row. This sub-formula can be omitted, but carefully placed redundant SAT formula can speed up the solving process. The outermost conjunction (1) indexed by conditions of ϕ imposes that the above sub-formulas (2) and (3) are to be applied to all the conditions in the predicate under test ϕ.

Sub-formulas (1), (2), and (3) are encoding for CACC, since sub-formulas (2) and (3) together satisfy condition (1) and (2) of Definition 3, and sub-formula (1) guarantees it for all the conditions. Sub-formula (4) specifies the condition for RACC that stipulates that values of all the minor conditions are equivalent. For RACC, we may also omit ϕ_c^j in (2); this is possible, since if constraints (2) and (4) hold and $x_c^j = \text{FALSE}$, then the i-th test case and the j-th test case differ only in the value of x_c so that ϕ_c^j and ϕ_c^i should be equivalent.

The encoding for a given test suite size induces a SAT formula, such that 'SAT' for the evaluation of the formula entails the existence of an ACC test suite with that size; in that case, the solution contains all the information on the ACC test suite. On the other hand, 'UNSAT' means the refutation of the existence of such a test suite.

Example 2. The following snippet of SAT-formula is the SAT-encoding of Boolean expression ϕ_1 with the test suite size 4. The sub-formulas (i), (ii), and (iii), respectively, express that conditions s, t and u are confirmed to independently affect the outcome of the predicate.

$$
\begin{array}{rl}
& \underline{\left(\begin{array}{l}
\quad (s^1 \wedge \phi_s^1) \wedge \Big(((\neg s^2 \wedge \phi_s^2) \wedge (t^1 \Leftrightarrow t^2) \wedge (u^1 \Leftrightarrow u^2)) \vee \\
\qquad ((\neg s^3 \wedge \phi_s^3) \wedge (t^1 \Leftrightarrow t^3) \wedge (u^1 \Leftrightarrow u^3)) \vee ((\neg s^4 \wedge \phi_s^4) \wedge (t^1 \Leftrightarrow t^4) \wedge (u^1 \Leftrightarrow u^4)) \Big) \\
\qquad\qquad\qquad \vdots \\
\vee\ (s^4 \wedge \phi_s^4) \wedge \Big(((\neg s^1 \wedge \phi_s^1) \wedge (t^4 \Leftrightarrow t^1) \wedge (u^4 \Leftrightarrow u^1)) \vee \\
\qquad ((\neg s^2 \wedge \phi_s^2) \wedge (t^4 \Leftrightarrow t^2) \wedge (u^4 \Leftrightarrow u^2)) \vee ((\neg s^3 \wedge \phi_s^3) \wedge (t^4 \Leftrightarrow t^3) \wedge (u^4 \Leftrightarrow u^3)) \Big)
\end{array}\right)}_{(i)} \\
\bigwedge & \underline{\left(\begin{array}{l}
(t^1 \wedge \phi_t^1) \wedge \Big(((\neg t^2 \wedge \phi_t^2) \wedge (s^1 \Leftrightarrow s^2) \wedge (u^1 \Leftrightarrow u^2)) \vee \\
\quad ((\neg s^3 \wedge \phi_s^3) \wedge (t^1 \Leftrightarrow t^3) \wedge (u^1 \Leftrightarrow u^3)) \vee ((\neg s^4 \wedge \phi_s^4) \wedge (t^1 \Leftrightarrow t^4) \wedge (u^1 \Leftrightarrow u^4)) \Big) \\
\qquad\qquad\qquad \vdots
\end{array}\right)}_{(ii)} \\
\bigwedge & \underline{(\,(u^1 \wedge \phi_u^1) \wedge \cdots)}_{(iii)}
\end{array}
$$

3.3 Algorithm

We proceed with the actual algorithm on how to find a smaller ACC test suite using SAT-encoding. For the discussion, we first argue about properties on the size of ACC test suites. From our literature review in Section 2.2, we conjecture that the minimum size of ACC test suites should be around $n+1$ for a predicate with n conditions. However, because no formal proof for this lower bound exists yet, we use the following proposition, which is a weaker form of the conjecture, but which can be proved easily:

Algorithm 1: The main part of our algorithm for CACC

```
  Input: A Boolean Expression ($\phi$)
  Output: A CACC test suite
1 Compute the feasible conditions of $\phi$, and store them in $conds_\phi$;
2 if($|conds_\phi| == 0$) return ("Exception: No feasible conditions exist.");
3 Set $size \leftarrow 2 * |conds_\phi|$, for the initial test size;
4 while true do
5     (isSAT,$model$) $\leftarrow$ checkSat($encode(\phi, conds_\phi, size)$);
6     if(isSAT == UNSAT)  return $suite$;
7     Make test $suite$ from $model$;
8     $size \leftarrow size - 1$
9 end
```

Proposition 2 *For a predicate with* n *conditions, the minimum size of a CACC/RACC test suite is at most* $2n$.

Proof. From Definition 3, an RACC test suite should include at least one RACC pair for each condition in ϕ. Since we can make an RACC pair with two test cases, $2n$ is enough for the size of an RACC test suite for ϕ with n conditions. Since a RACC test suite is also an CACC one, this property also holds for CACC.

Using Proposition 2., the algorithm is designed as in Algorithm 1. The algorithm, at the beginning (in Line 1), computes feasible conditions of ϕ. Computing all the feasible conditions can be done by collecting conditions whose PDCs are satisfiable (possibly, using a SAT solver). Based on the computed feasible conditions ($conds_\phi$) and their number ($|conds_\phi|$), the algorithm starts searching for a minimum test suite. It iteratively attempts to generate an ACC test suite by decrementing the size of a test suite by one, starting from $2 * |conds_\phi|$. In each iteration, the algorithm applies the SAT-encoding of the current test size, and applies a SAT solver to the encoded formula. The decremental iteration continues while the encoded formula is satisfiable, and terminates when it encounters '*unsatisfiable (UNSAT)*' (Line 6). The test suite found in the last satisfying iteration is minimal. Line 2 handles a corner case, where no feasible conditions are contained in ϕ, by throwing an exception.

The encoded formula in each iteration is a SAT problem over Boolean variables, and the test size. The size starts at '$2 * |conds_\phi|$' and is reduced by one in each iteration. Thus, the algorithm is guaranteed to terminate with a minimum ACC test suite, in principle. In practice, however, limited computational resources may prevent us from finding the minimum one. The SAT solver may take a lot of time and/or a lot of memory, as the attempted search approaches the optimal size. Thus, a concern is scalability of the algorithm, which we examine by experiments in Section 5.

The algorithm uses $2n$ for the initial test size based on Proposition 2.. We have two main reasons to adopt this proposition, instead of the conjecture for $n+1$. First, our algorithm is certainly correct using the proved proposition with

$2n$, instead of using the (unproved) conjecture with $n+1$. Second, our algorithm with $2n$ for the initial size performs well in practice, outperforming existing techniques, as shown in Section 5. Moreover, if the worst-case bound of $n+1$ is proven in the future, we can easily adapt that result in our algorithm, by setting $n+1$ as the initial size.

4 Related Work

Test suite reduction and minimization is one of central subjects in software testing. To the best of our knowledge, the earliest work which discusses the test suite sizes in MCDC testing is by Jones and Harrold [13]. They develop a test suite reduction technique for MCDC, which, given a predicate ϕ and an MCDC test suite T, finds an MCDC test suite T' such that $T' \subset T$. The basic approach of the technique is to construct such an MCDC test suite T', by iteratively choosing or removing one test case from the given MCDC test suite T based on the *contribution* weight computed for each test case in T. For example, their 'build-up' approach constructs an MCDC test suite by iteratively adding a test case in T with the highest contribution weight to T', starting from the empty set as the initial test suite of T'. Note that the JH-algorithm is a test reduction technique, rather than MCDC test generation; however, by complementing the algorithm with a function to prepare initial MCDC test suites, it can be used as a test generation technique for MCDC.

Arcaini et al. [14] developed a general framework for test generation for Boolean expression testing with various coverage criteria, including MCDC. The technique constructs an MCDC test suite with the basic approach of accumulating an MCDC pair (a pair of test cases) for each condition in turn. Offutt et al. [18], in developing a model-based testing technique with MCDC testing, discuss a similar technique to construct an MCDC test suite. However, both techniques require a test suite of size $2n$ for a predicate with n conditions, which is usually larger than the test suites that our algorithm generates.

Bloem et al. [19] devised an algorithm to construct MCDC test suites, also in developing their model-based testing technique using MCDC. The technique is similar to those by Arcaini et al. [14] and Offutt et al. [18], in the respect that its basic procedure is to accumulate an MCDC pair for each condition in turn; however, they apply an improvement to this basic approach to reduce the size of generated test suites. The improvement is that the algorithm, for every condition, checks if the current test suite already includes an MCDC pair for the condition, when adding an MCDC pair for each condition. If the MCDC pair already exists in the test suite, the algorithm skips adding an MCDC pair for that condition. They also use SAT-solving to realize the technique.

Our algorithm is inspired by the SAT-based test generation technique for combinatorial interaction testing (CIT) by Hnich et al. [20]. The key of their technique is a SAT-encoding of the problem of finding a CIT test suite with a specified size. It is confirmed that the technique can construct CIT test suites with reasonably small sizes, compared with other approaches. Due to the significance, this work is followed by a number of studies for extension or acceleration

Table 2. The benchmark set consists of 20 logical expressions, retrieved from "*Traffic Collision Avoidance System (TCAS)*" of an avionics system [8]. The logical expressions in the benchmark set contain 5 to 15 (feasible and infeasible) conditions, as indicated by #conds. The table also shows the infeasible conditions (I.C.) for each benchmark; ACC pairs cannot be made for such infeasible conditions.

No	Boolean expression	#conds	I.C.
1	$a(!b+!c)d + e$	5	
2	$!(ab)(d!e!f+!de!f+!d!e!f)((ac(d + e)h) + (a(d + e)!h) + (b(e + f)))$	7	
3	$!(cd)(!ef!g!a(bc+!bd))$	7	
4	$ac(d + e)h + a(d + e)!h + b(e + f)$	7	
5	$!ef!g!a(bc+!bd)$	7	
6	$(!ab + a!b)!(cd)!(gh)((ac + bd)e(fg+!fh))$	8	
7	$(ac + bd)e(fg+!fh)$	8	
8	$(a((c+d+e)g+af+c(f+g+h+i))+(a+b)(c+d+e)i)!(ab)!(cd)!(ce)!(de)!(fg)!(fh)!(fi)!(gh)!(hi)$	9	
9	$a(!b+!c + bc!(!fgh!i+!ghi)!(!fglk+!g!ik)) + f$	9	
10	$a((c + d + e)g + af + c(f + g + h + i))(a + b)(c + d + e)i$	9	b, h
11	$(ac + bd)e(i+!g!k+!j(!h+!k))$	10	
12	$(ac + bd)e(i+!g!k+!j(!h+!k))(ac + bd)e(i+!g!k+!j(!h+!k))$	10	
13	$(!ab + a!b)!(cd)(f!g!h+!fg!h+!f!g!h)!(jk)((ac + bd)e(f + (i(gj + hk))))$	11	
14	$(ac + bd)e(f + (i(gj + hk)))$	11	
15	$(a(!d+!e + de!(!fgh!i+!ghi)!(!fglk+!g!ik))+!(!fgh!i+!ghi)!(!fglk+!g!ik)(b + c!m +$ $f))(a!b!c+!ab!c+!a!bc)$	12	
16	$a + b + c+!c!def!g!h + i(j + k)l$	12	
17	$a(!d+!e + de!(!fgh!i+!ghi)!(!fglk+!g!i!k))+!(fgh!i+!ghi)!(!fglk+!g!ik)(b + c!m + f)$	12	
18	$a!b!c!d!ef(g+!g(h + i))!(jk+!jl + m)$	13	
19	$a!b!c(!f(g+!f(h + i))) + f(g+!g(h + i)!d!e)!(jk+!jl!m)$	13	
20	$a!b!c(f(g+!g(h + i)))(!e!n + d)+!n(jk+!jl!m)$	14	

(e. g., [21]). Our work in this paper thus can be seen as a new application direction of the SAT-based test generation technique to MCDC testing, and also as an import of new technical element to MCDC testing field.

5 Experimental Evaluation

In this section, we conduct experiments to evaluate our proposed technique. Due to the space limitation, we only evaluate the proposed algorithm for the CACC case, as it is the most basic, practical, and interesting case among ACC variants. To clarify the purpose of the experiments and evaluation, we set the following research questions:

- RQ1: Does our algorithm perform better than existing techniques, with respect to the sizes of generated test suites and computation times?
- RQ2: Can our algorithm find minimal CACC test suites?

For investigating these RQs, we implemented our algorithm in Scala. Also to conduct the experiments, we prepared a benchmark set of logical expressions retrieved from "*Traffic Collision Avoidance System (TCAS)*" of a real-world avionics system [8]. The details about benchmark data are shown in Table 2.

For RQ1, we also implemented the MCDC test generation technique based on the JH-algorithm and the technique by Bloem et al., as the the state-of-the-art techniques by ourselves, since implementations of these techniques are not available. Since JH-algorithm is an MCDC test reduction technique instead of test generation, we complemented it with the function to randomly generate MCDC test suites so that JH-algorithm can reduce them as initial test suites. The function is realized to randomly generate m ACC pairs for each condition of a given formula. Therefore, the most basic form of $m = 1$ means $2n$ test cases, where n is the number of conditions in the given formula. We prepared several variants that vary in the sizes of initial MCDC test suites, as follows: $m = 1, 50, 200$, and 400, respectively denoted by JH_1, JH_{50}, JH_{200}, and JH_{400}. Also for RQ1, the timeout is set to 60 seconds for our algorithm for a proper comparison and evaluation.

For RQ2, we measure the time required for our algorithm to find minimal test suites by finding UNSAT for the benchmark set, thus proving that no test suite smaller than the previously found satisfiable assignment exists. For this, the timeout is set to 1800 seconds for our algorithm.

Table 3 shows the experimental results. We answer the research questions, by observing the experimental results, in the following.

Answer to RQ1. From the experimental results, we conclude that our algorithm performs better than the state-of-the-art techniques in both the size of generated test suites and computation times. First, our algorithm wins against the other algorithms for all the benchmark data. On other hand, the algorithm variants based on the JH-algorithm or the technique by Bloem et al., only perform equally well to ours only in a couple of cases. Moreover, our algorithm can build such

Table 3. The results of our experiments. This table shows the sizes of the generated CACC test suites ('#tests') and the computation times in seconds ('time') of the algorithms (JH_1, JH_{50}, JH_{200}, JH_{400}, by Bloem et al., and our algorithm) for the benchmark set in Table 2. To properly answer RQ1 and RQ2, we use two timeout settings for our algorithm, 60 seconds ($time_{<60}$), and 1800 seconds ($time_{<1800}$). A '—' in columns '$time_{<1800}$' means a minimum test suites is found within in 60 seconds, and the time is given in column '$time_{<60}$'. The bold font in the columns for '#tests' denote instances in which the test suite size is smaller than or equal to those found by other algorithms, while that in the columns for 'time' means the algorithms returned the test suite within 60 seconds. For columns of our algorithms, a '*' denotes a minimal test suite is found with the guarantee of finding 'UNSAT' by the SAT solver. '#wins' denotes the number of cases where the algorithm generates the smallest test suite among all algorithms within 60 seconds; multiple winners may exist for each benchmark data. 'impr(%)' denotes the reduction rate on the test suite size by our algorithm compared with that by the corresponding algorithm of the column; e.g., the test suite size by our algorithm is smaller than that by JH_{400} by 19.7% in total.
The experiments were performed on a machine with a Quad-Core Intel Xeon E5 3.7 GHz and 64 GB Memory running on Mac OS High Sierra 10.13.3. For running the programs, Scala option "-Xmx8g -Xms1024m" is used. As the back-end SAT solver, we used SMT solver Z3 [22].

	JH_1		JH_{50}		JH_{200}		JH_{400}		Bloem et. al.		our algorithm (2n for the initial size)			
No	#tests	time	#tests	time	#tests	time	#tests	time	#tests	time	#tests	$time_{<60}$	#tests	$time_{<1800}$
1	**6**	**1.0**	**6**	**4.1**	**6**	**12.6**	**6**	**10.2**	**6**	**0.8**	**6***	**0.5**	—	
2	8	**1.6**	7	**13.7**	7	**18.2**	7	**17.0**	7	**1.6**	**5***	**1.3**	—	
3	9	**1.7**	9	**2.7**	9	**3.5**	9	**2.7**	8	**1.4**	**6***	**1.0**	—	
4	12	**1.7**	7	**20.9**	**6**	**50.9**	**6**	95.6	9	**1.4**	**6***	**0.9**	—	
5	9	**1.5**	**8**	**5.0**	**8**	**4.9**	**8**	**4.9**	9	**1.4**	**8***	**0.9**	—	
6	12	**2.0**	9	**5.4**	9	**5.5**	9	**5.3**	9	**1.8**	**3***	**1.5**	—	
7	10	**1.7**	8	**21.2**	**7**	68.5	**7**	85.2	10	**1.7**	**7***	**1.0**	—	
8	13	**2.5**	8	**24.2**	7	90.7	7	119.7	10	**2.4**	**4***	**2.6**	—	
9	11	**2.7**	10	**17.4**	10	**26.5**	10	**45.6**	11	**2.9**	**9***	**3.0**	—	
10	9	**2.0**	8	**18.5**	**7**	68.1	**7**	144.2	9	**1.9**	**7***	**1.0**	—	
11	12	**2.7**	10	**26.6**	10	120.3	9	321.4	12	**2.9**	**8***	**1.8**	—	
12	12	**3.0**	10	**26.9**	10	119.7	9	348.7	12	**3.1**	**8***	**2.5**	—	
13	15	**3.7**	13	**26.3**	12	95.3	13	283.9	13	**3.8**	**9***	**3.5**	—	
14	12	**3.1**	13	**30.0**	12	147.5	12	423.4	13	**3.4**	**9***	**1.8**	—	
15	18	**4.9**	17	**33.2**	16	151.0	16	439.0	17	**5.0**	**9***	**5.8**	—	
16	13	**3.4**	13	**32.5**	12	140.8	**11**	301.8	13	**3.6**	**11***	**2.0**	—	
17	18	**4.8**	15	**35.7**	13	166.5	13	441.2	16	**5.2**	**11**	**4.6**	**11***	80.7
18	15	**4.1**	13	**31.6**	13	140.1	13	284.9	14	**3.9**	**13**	**2.6**	**13**	>1800
19	20	**4.7**	18	**37.0**	16	180.2	16	499.1	14	**4.5**	**12**	**3.3**	**12***	197.7
20	18	**4.7**	18	**40.7**	16	190.7	15	555.1	16	**4.5**	**12**	**3.6**	**12***	167.3
#wins	1		2		3		2		1		20	—		
impr(%)	35.3		25.9		20.9		19.7		32.0		—		—	

smaller test suites fairly quickly, i. e., within a few seconds for all the benchmark data, which is much faster than the other techniques. The improvement rate achieved by our algorithm is also significant; it can generate test suites that are from 16.2% up to 32.0% smaller than the the other techniques.

Answer to RQ2. From the experiments, we confirm that the resulting test suite is of size equal or less than $n+1$ in 17 out of 20 cases. We can also confirm that in 19 cases our algorithm can find the guaranteed minimum sizes by finding 'UNSAT' within 1800 seconds. For the case where the algorithm cannot guarantee the minimum size of the test suite (i. e., for benchmark #18), there may exist a smaller CACC test suite.

6 Conclusion and Future Work

We believe our algorithm to construct small or even the minimum MCDC test suites will play a significant role broadly in the software testing field, since (1) MCDC has become a de-facto standard coverage criterion in testing safety-critical systems, and (2) the technique, as a basic testing technique, can be used in combination with other testing techniques, such as model-based testing and program-code-level testing.

Model-based testing, a sub-discipline of model-based development (MBD), is another key technique in safety-critical domains such as in avionics and automotive. A number of studies have attempted to introduce MCDC testing in model-based testing techniques, to enhance the accountability as well as its bug-detecting ability [18,7,23,24]; also see Section 4. Generally, these techniques deal with state transition systems as models, and regard a sequence of states as a test case. To derive effective test cases, these techniques apply the MCDC criterion to the *transition guards*, which specify as a logical formula the conditions for transitions to take place. This approach is reasonable since real-world transition systems often become complex and error-prone, however, their test generation components are not efficient w.r.t. the test suite size and computation time. Our algorithm can complement such model-based testing techniques.

Another application area of our algorithm is *Coverage-Directed Test case Generation (CDG/CDTG)*. CDTG is a white-box testing technique, working at program code level to generate test cases, to achieve a higher code coverage. CDTG and its adaptation for MCDC have been actively studied [25,26,27]. Such CDTG techniques targeting MCDC analyze the structure of the given program, especially logical formulas embedded in the program, and try to find test inputs that reach each logical formula in the program. Although these techniques differ in several dimensions such as search techniques (e. g., dynamic symbolic execution [25], model checking [26], search-based optimization [27]), they have in common that they prepare MCDC test suites for embedded logical formulas before starting the search. Similar to the situation of model-based testing, our proposed algorithms can effectively complement such CDTG techniques.

We consider several directions for future work. One direction is to extend our algorithm to deal with general logical formulas, e. g., $x = 0 \land (x > 0 \lor y = 0)$, instead of their abstracted forms in Boolean expressions, i. e., $s \wedge (t \vee u)$. It is since our algorithm may produce unusable tests for such an abstracted predicate. For example, recall Table 1 is an MCDC test suite for $s \wedge (t \vee u)$. Note, however, that the first test case, i. e., $\{s = \textsc{True}, t = \textsc{True}, u = \textsc{False}\}$, is unusable if $s \wedge (t \vee u)$ originates from $x = 0 \wedge (x > 0 \vee y = 0)$ by abstraction, since there is no assignment that makes $s = \textsc{True}$ (i. e., $x = 0$) and $t = \textsc{True}$ (i. e., $x > 0$) at the same time. One approach for this would be to impose additional constraints among abstracted variables, to specify hidden relations between s and t. In the example, a constraint like $\neg(s \wedge t)$ would encode that. The PDC of s for this predicate would be constructed as follows: $\phi_{s=\textsc{True}} \oplus \phi_{s=\textsc{False}} \wedge \neg(s \wedge t)$. We need careful investigations on this approach including how to specify such additional constraints automatically and how it affects the size of test suite.

Another direction is to further accelerate or scale our algorithm. Although it was shown by the experiments that our algorithm works fairly well for a real-world system in the avionics domain, it may need to be prepared for larger and more complex systems. Toward this technical improvement, we think the technique using incremental SAT solving [21] can be effectively applied. Thirdly, we consider to apply our technique to other ACC variants explained in Section 2. We are also interested in applying it to improve the testing technique for Deep Neural Networks based on MCDC proposed by [12].

Acknowledgment. We thank Hélène Waeselynck and anonymous reviewers whose comments have greatly improved this paper.

References

1. RTCA: DO-178C: Software considerations in airborne systems and equipment certification (2011)
2. ISO26262: Int. org. for standardization, road vehicles - functional safety - (2009)
3. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal **9**(5) (1994) 193–200
4. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
5. Kelly J., H., Dan S., V., John J., C., Leanna K., R.: A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA (2001)
6. Felbinger, H., Pill, I., Wotawa, F.: Classifying test suite effectiveness via model inference and ROBBDs. In: Proc. 10th Int. Conf. on Tests and Proofs (TAP 2016), Vienna, Austria. (2016) 76–93
7. Bloem, R., Hein, D.M., Röck, F., Schumi, R.: Case study: Automatic test case generation for a secure cache implementation. In: Proc. 9th Int. Conf. on Tests and Proofs (TAP 2015), L'Aquila, Italy. (2015) 58–75
8. Weyuker, E.J., Goradia, T., Singh, A.: Automatically generating test data from a boolean specification. IEEE Trans. Software Eng. **20**(5) (1994) 353–363
9. Dupuy, A., Leveson, N.: An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In: Proc. 19th Digital Avionics Systems Conf. (DASC2000)

10. Vilkomir, S., Starov, O., Bhambroo, R.: Evaluation of t-wise approach for testing logical expressions in software. In: Proc. Sixth IEEE Int. Conf. on Software Testing, Verification and Validation, ICST 2013 Workshops, Luxembourg. (2013) 249–256
11. Kapoor, K., Bowen, J.P.: A formal analysis of MCDC and RCDC test criteria. Softw. Test., Verif. Reliab. **15**(1) (2005) 21–40
12. Sun, Y., Huang, X., Kroening, D.: Testing deep neural networks (2018) arXiv:1803.04792.
13. Jones, J.A., Harrold, M.J.: Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Software Eng. **29**(3) (2003) 195–209
14. Arcaini, P., Gargantini, A., Riccobene, E.: How to optimize the use of SAT and SMT solvers for test generation of boolean expressions. Comput. J. **58**(11) (2015) 2900–2920
15. Chilenski, J.J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report DOT/FAA/AR-01/18, Office of Aviation Research (2001)
16. Vilkomir, S.A., Bowen, J.P.: From MC/DC to RC/DC: formalization and analysis of control-flow testing criteria. In: Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers. (2008) 240–270
17. Whalen, M.W., Gay, G., You, D., Heimdahl, M.P.E., Staats, M.: Observable modified condition/decision coverage. In: 35th Int. Conf. on Software Engineering, ICSE '13, San Francisco, CA, USA. (2013) 102–111
18. Offutt, A.J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. Softw. Test., Verif. Reliab. **13**(1) (2003) 25–53
19. Bloem, R., Greimel, K., Koenighofer, R., Roeck, F.: Model-based MCDC testing of complex decisions for the java card applet firewall. In: Proc. of the Fifth Int. Conf. on Advances in System Testing and Validation Lifecycle (VALID 2013). (2013) 1–6
20. Hnich, B., Prestwich, S., Selensky, E.: Constraint-based approaches to the covering test problem. In: Proc. of CSCLP 2004. Volume 3419 of LNAI. (2005) 172–186
21. Yamada, A., Kitamura, T., Artho, C., Choi, E.H., Oiwa, Y., Biere, A.: Optimization of combinatorial testing by incremental SAT solving. In: Proc. of 8th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST 2015). (2015) 1–10
22. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. of 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, (TACAS 2008) Hungary. (2008) 337–340
23. Heimdahl, M.P.E., Devaraj, G.: On the effect of test-suite reduction on automatically generated model-based tests. Autom. Softw. Eng. **14**(1) (2007) 37–57
24. Gargantini, A., Heitmeyer, C.L.: Using model checking to generate tests from requirements specifications. In: Proc. 7th European Conf. of Software Engineering, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, (ESEC/FSE'99). (1999) 146–162
25. Pandita, R., Xie, T., Tillmann, N., de Halleux, J.: Guided test generation for coverage criteria. In: Proc. of 26th IEEE Int. Conf. on Software Maintenance (ICSM 2010), Timisoara, Romania. (2010) 1–10
26. Bokil, P., Darke, P., Shrotri, U., Venkatesh, R.: Automatic test data generation for C programs. In: Proc. of Third IEEE Int. Conf. on Secure Software Integration and Reliability Improvement, SSIRI 2009. (2009) 359–368
27. Awedikian, Z., Ayari, K., Antoniol, G.: MC/DC automatic test input data generation. In: Proc. of Genetic and Evolutionary Computation Conf. (GECCO 2009), Canada. (2009) 1657–1664