Generating Minimal Fault Detecting Test Suites for General Boolean Specifications

Angelo Gargantini[a,*], Gordon Fraser[b]

[a] *Dip. di Ing. dell'Informazione e Metodi Mat., University of Bergamo, Dalmine, Italy*
[b] *Software Engineering Chair, Saarland University, Saarbrücken, Germany*

Abstract

Context: Boolean expressions are a central aspect of specifications and programs, but they also offer dangerously many ways to introduce faults. To counter this effect, various criteria to generate and evaluate tests have been proposed. These are traditionally based on the structure of the expressions, but are not directly related to the possible faults. Often, they also require expressions to be in particular formats such as disjunctive normal form (DNF), where a strict hierarchy of faults is available to prove fault detection capability.

Objective: This paper describes a method that generates test cases directly from an expression's possible faults, guaranteeing that faults of any chosen class will be detected. In contrast to many previous criteria, this approach does not require the Boolean expressions to be in DNF, but allows expressions in any format, using any deliberate fault classes.

Method: The presented approach is based on creating test objectives for individual faults, such that efficient, modern satisfiability solvers can be used to derive test cases that directly address the faults. Although the number of such test objectives can be high depending on the considered fault classes, a number of optimizations can be applied to reduce the test generation effort.

Results: Evaluation on a set of commonly used benchmarks shows that despite guaranteeing fault coverage, the number of test cases can be reduced even further than that produced by other state of the art strategies. At the same time, the fault detection capability is not affected negatively, and clearly improves over state of the art criteria for general form Boolean expressions.

Conclusion: The approach presented in this paper is shown to improve over the state of the art with respect to the types of expressions that can be handled, the fault classes that are guaranteed to be covered, and the sizes of test suites generated automatically. This has implications for several fields of software testing: A main application is specification based testing, but Boolean expressions also exist in normal source code and need to be tested there as well.

*Corresponding author
Email addresses: angelo.gargantini@unibg.it (Angelo Gargantini), fraser@cs.uni-saarland.de (Gordon Fraser)

Keywords: test case generation, Boolean testing, fault-based testing

1. Introduction

Boolean expressions are frequently found in logical predicates inside programs and specifications to model complex conditions under which some code is executed or an action is performed. In theory, a Boolean predicate p with n variables requires 2^n test cases in order to be distinguished from any other predicate not equivalent to p. In practice, n can be quite big, as for an example shown in a study by Chilenski and Miller [8], who found Boolean expressions with 30 or more conditions in an electronic flight implementation system. Consequently, exhaustive testing is not feasible in practice and therefore testing criteria are applied to select subsets of all possible test cases. Thanks to these criteria test suites become tractable, but the fault detection capability of the test suites is reduced with respect to exhaustive testing.

As a consequence of this problem new criteria are introduced and existing criteria are improved, with two contrasting goals: reducing the number of test cases while maximizing the fault detection capability. Some of these criteria are even mandated in certain domains, as in the case of MCDC [8] for avionic software. Traditionally, test criteria are defined by an algorithm or rules to build test cases starting from a given Boolean expression. Such algorithms consider the syntactical structure of the Boolean expression but not explicitly the fault classes, and therefore the fault detection capability must be proved later. Some criteria, like MAX-A and MAX-B [41], and MUMCUT [44] require that the Boolean expression under test is first translated to a normal form (like the Disjunctive Normal Form) and then the algorithm for test generation is applied. In such cases, the fault detection capability with respect to the original Boolean expression (if it is not already in DNF) may be affected by the translation to DNF and must be further investigated [5].

The fault detection capability is measured using explicit fault classes – each fault class represents a different type of error that can occur in a Boolean expression, similar to mutation operators of Boolean expressions in mutation testing. There is a hierarchy of such fault classes [32, 7], and research is performed in order to discover further fault classes or new relations among fault classes.

In this paper, we investigate an approach in which test cases are generated directly from general form (GF) Boolean expressions targeting specific fault classes and several reduction policies are applied to minimize the size of resulting test suites without reducing the desired fault detection capability. This guarantees that all considered fault classes are fully covered while reducing the number of test cases more than any single previously introduced criterion could achieve. A change of perspective from classical coverage criteria is required: The test case generation process is based solely on the faults the tester targets, and the structure of the Boolean expression is not explicitly considered. In this approach, a test case is generated for each possible fault of a fault class, given a Boolean predicate. This by itself may lead to larger test suites than other

test criteria would create. However, as we show in this paper, a reduction larger than that achieved by any single test criterion can be achieved by optimizing the test case generation process instead of the test criteria, while still guaranteeing that the considered fault classes are fully covered.

In particular, we explore a range of optimizations: When generating test cases explicitly for faults it is possible to check if a fault has been covered before actually generating the test case, and only if the fault has not previously been covered a new test case is generated; we call this optimization *monitoring* of faults. The order in which faults are considered during test case generation has an impact on the size of resulting test suites, therefore we apply different heuristic *orderings* of faults. *Collecting* is a technique where several independent faults are considered at the same time to be detected by just one test case, which leads to a significant reduction of test cases. Finally, we also consider traditional *minimization*, which is an optimization that is applied to test suites after generation; a heuristic selects a subset of a test suite that guarantees full coverage with respect to a criterion or fault class.

In order to automatically generate the test cases we formalize the goals of the test case generation as Boolean predicates and the problem of finding a test case that covers a test goal reduces to the problem of finding a model for a Boolean formula. This problem can be efficiently solved by means of several techniques; in the experiments in this paper we use both SAT and SMT solvers for test case generation and we compare them.

In this paper we build on our past experience [18] on generating test cases that explicitly target specific faults. This paper extends our previous work [15] in the following ways: (i) It considers not only DNF expressions but also General Form Boolean expressions, which have different fault classes and a different hierarchy among them. (ii) It extends the original approach in order to deal with constraints over the variables in the expressions; such constraints for example arise in source code to describe when a predicate is reached. (iii) The discussed optimization techniques are thoroughly evaluated and statistically analysed. (iv) A comparison with other testing criteria for DNF and GF Boolean expressions in terms of test suite size and fault detection capability is presented. (v) We evaluate the use of a SAT solver besides an SMT solver to compare them in terms of performance.

This paper is organized as follows: First, we introduce some basic definitions and the notation used as well as a background on test criteria and fault classes in Section 2. Section 3 describes our approach to generate test cases specifically to address certain fault classes, and Section 4 presents the optimizations we apply in order to reduce the number of test cases that are generated. Section 5 describes the setup, aims, and results of experiments performed in order to evaluate this approach. Section 6 discusses related work. Finally, Section 7 summarizes and concludes the paper.

2. Background

Programs and specifications often use Boolean expressions as guards for conditional instructions, cycles, or transitions. Many specification formalisms such as the often used AND-OR tables (as those used in RSML [34] or in SCR [21]) can also be seen as Boolean expressions. This section gives all necessary definitions for the remainder of the paper and introduces fault classes.

2.1. Definitions and notation

In this paper we do *not* assume that Boolean expressions are given in minimal disjunctive normal form (DNF) but we consider general form (GF) Boolean specifications. Boolean expressions are those involving Boolean operators like AND, OR, and NOT (denoted by $\wedge$, $\vee$, $\neg$). We follow the definitions and notations used by Kapoor and Bowen [25]: the symbols x_1, x_2, etc. are referred to as inputs or variables, and an occurrence of an input in a formula is referred to as a condition. For example, the formula $x_1 \wedge x_2 \vee x_1$ contains two variables (x_1 and x_2), whereas the number of conditions is three (two x_1's and one x_2). There are no restrictions on how operators and conditions are joined together in general form expressions, while DNF requires disjunctions of conjunctive conditions. In the context of testing Boolean predicates, a *test case* is a value assignment to every Boolean variable in the formula (a "complete" model of the formula). A *test suite* simply is a set of test cases.

2.2. Fault classes

Fault-based testing methods first hypothesize certain types of faults that may be committed by programmers, and then design test cases targeted at these faults [35]. In contrast to other testing methods, fault-based testing can demonstrate the absence of hypothesized faults in specific classes, called *fault classes*. In this paper we use the classical fault classes studied by Weyuker et al. [41], DeMillo and Offutt [11], Okun et al. [36], and Kapoor and Bowen [25].

We use the same Boolean expression $(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$ to explain the classical 10 fault classes which are organized into two categories, as follows.

Operator Faults

- Operator Reference Fault (ORF). An occurrence of a logical connective $\wedge$ replaced by $\vee$ or vice versa. For example, $(x_1 \vee \neg x_2) \vee (x_3 \wedge x_4)$ is an ORF of $(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$.

- Expression Negation Fault (ENF). An ENF is a mutant with a subexpression (except conditions) replaced by its negation. For example, $\neg(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$ is an ENF of $(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$.

- Variable Negation Fault (VNF). An occurrence of a condition is replaced by its negation. For example, $(\neg x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$ is a VNF of $(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$.

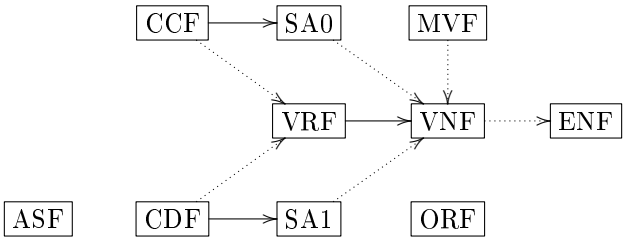

Figure 1: Hierarchy among fault classes: Lines between fault classes indicate subsumption relations, dotted lines represent subsumption relations which were initially established [25] but were later proved not to hold [7].

- Associative Shift Fault (ASF). ASF is caused by omission of the brackets because of the misunderstanding about operator evaluation priorities. For example, $x_1 \vee \neg x_2 \wedge (x_3 \wedge x_4)$ is an ASF of $(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$.

Operand Faults

- Missing Variable Fault (MVF). An occurrence of a condition is omitted in the expression. For example, $(x_1 \vee \neg x_2) \wedge (\ x_3\ )$ is an MVF of $(x_1 \vee \neg x_2) \wedge (\ x_3 \wedge x_4\ )$. Note that a condition of MVF may be connected by $\wedge$ or $\vee$.
- Variable Reference Fault (VRF). An occurrence of a condition is replaced by another possible condition. A condition is said to be possible if its variable has already appeared in the expression. For example, $(x_1 \vee \neg x_2) \wedge (\ \neg x_1\ \wedge x_4)$ is a VRF of $(x_1 \vee \neg x_2) \wedge (\ x_3\ \wedge x_4)$.
- Clause Conjunction Fault (CCF). An occurrence of condition c is replaced by c$\wedge$c', in which c' is a possible condition. For example, $(x_1 \vee \neg x_2) \wedge (\ x_1 \wedge x_3\ \wedge x_4)$ is a CCF of $(x_1 \vee \neg x_2) \wedge (\ x_3\ \wedge x_4)$.
- Clause Disjunction Fault (CDF). An occurrence of condition c is replaced by c$\vee$c', in which c' is a possible condition. For example, $(x_1 \vee \ \neg x_2 \vee x_3\ ) \wedge (x_3 \wedge x_4)$ is a CDF of $(x_1 \vee \ \neg x_2\ ) \wedge (x_3 \wedge x_4)$.
- Stuck-At-0 Fault (SA0). An occurrence of a condition is replaced by 0 in the expression. For example, $(x_1 \vee \ 0\ ) \wedge (x_3 \wedge x_4)$ is an SA0 of $(x_1 \vee \ \neg x_2\ ) \wedge (x_3 \wedge x_4)$.
- Stuck-At-1 Fault (SA1). An occurrence of a condition is replaced by 1 in the expression. For example, $(x_1 \vee \ 1\ ) \wedge (x_3 \wedge x_4)$ is an SA1 of $(x_1 \vee \ \neg x_2\ ) \wedge (x_3 \wedge x_4)$.

Among fault classes, there is a hierarchy of *subsumption* relations. We say that a fault class F_1 subsumes another class F_2 if a test suite that is able to detect all the faults in F_1 will also detect all the faults in F_2. The hierarchy for the 10 fault classes presented above has been studied by Kapoor and Bowen

[25] and by Okun et al. [36], and was later corrected by Chen et al. [7]. Figure 1 depicts the hierarchy, where arrows denote subsumption. Dotted arrows show the subsumption relations stated by Kapoor and Bowen [25] which were later proved not correct and removed by Chen et al. [7] by also considering equivalent mutants. Chen et al. [7] also introduced the notion of co-subsumption: two classes F_1 and F_2 co-subsume another class E if the test suite generated for both F_1 and F_2 is guarantee to detect also all the faults in E. They found that CCF and CDF co-subsume VRF, and SA1 and SA0 co-subsume both MVF and VNF. Practical experiments [18] suggest that in many cases the VNF subsumes the ORF which in turn subsumes the ENF (but there is no guarantee of this relation). The ORF can be considered in-between VNF and ENF.

A hierarchy among fault classes is useful when generating tests: if one generates a test suite detecting fault classes at the top of the hierarchy, then also all the other faults in the hierarchy will be detected by the same test suite. For this reason, hierarchies are used to devise new testing criteria, for example in the case of DNF, Kaminski and Ammann [24] exploit this weak subsumption relation between LIF and LRF and between LRF and LOF in the Minimal-MUMCUT strategy to obtain test suites with the same fault detection capability with fewer test cases than MUMCUT.

However, considering that the correct hierarchy contains only very few subsumption relations for Boolean specifications in Generalized Form, it seems very difficult to devise a technique targeting just some fault classes and guarantee that also all the other fault classes are covered as well. For this reason, a method to generate tests directly targeting all the faults, like that presented in this paper, seems more suitable for fault-based tests generation for GF Boolean expressions.

2.3. Fault-based testing

The erroneous implementation φ' of a Boolean expression φ can be discovered only when there exists a test case t in which the condition $\varphi \oplus \varphi'$, called *detection condition* [30], evaluates to true, i.e., $t \models \varphi \oplus \varphi'$ where $\oplus$ denotes the logical *exclusive or* operator, and $t \models \phi$ means that t is a model of ϕ. Indeed, $\varphi \oplus \varphi'$ is true only if φ' evaluates to a different value than the correct predicate φ. This detection condition is also called *Boolean difference* or *derivative* [1].

The erroneous implementation φ' is also used in the context of mutation analysis, which is a technique to evaluate the quality of a test suite by the number of artificially introduced faults that can be distinguished from the original program. For every fault class C it is possible to define a mutation operator μ_C, which can be seen as a function that returns all possible faulty Boolean expressions that can be obtained from a given Boolean expression according to the fault class. In mutation testing, the erroneous implementation φ' is called *mutant*, since it can be obtained by applying a small syntactical change (mutation) to φ. Each mutation can be seen as belonging to a fault class, and it can be automatically generated by applying mutation operators to Boolean expressions.

In accordance with test case generation from logical predicates, we call the predicate $\varphi \oplus \varphi'$ *test predicate* or test goal.

Example 1. If the Boolean predicate $a \wedge b$ is implemented as a (a missing variable fault – MVF), then the test predicate is $a \wedge b \oplus a$ which is equivalent to $a \wedge \neg b$. Only a test case in which a is true and b is false can uncover the fault.

Let φ be a predicate and C a fault class. We denote with $F_C(\varphi)$ the set of all the possible faulty implementations of φ according to the fault class C (as explained in Section 2.2). $F_C(\varphi)$ can be obtained by repeatedly applying the mutation operator μ_C that represents the fault class C to φ. The test predicates to discover the fault class C in φ are the expressions $\varphi \oplus \varphi'$ for all φ' in $F_C(\varphi)$.

Example 2. Consider the expression $a \wedge b$ and let the fault class C be VNF, then $F_{VNF}(a \wedge b) = \{\neg a \wedge b, a \wedge \neg b\}$ and the test predicates are the following two expressions: $(a \wedge b) \oplus \neg a \wedge b$ (which is equivalent to b) and $(a \wedge b) \oplus a \wedge \neg b$ (which is a).

Definition 1. Test Predicates. Let φ be a Boolean predicate. The set $\Gamma_C(\varphi)$ of test predicates for the fault class C is given by the expressions $\{\varphi \oplus \varphi' \mid \varphi' \in F_C(\varphi)\}$.

A test suite $\mathcal{T}$ is adequate to test the predicate φ with respect to a fault class C if it covers every test predicate generated for φ and C: i.e., if for every test predicate tp in $\Gamma_C(\varphi)$ there exists a test case t in $\mathcal{T}$ such that t is a model of the test predicate tp (i.e., it evaluates to true in t).

Definition 2. Fault Detecting Adequacy. The test suite $\mathcal{T}$ is adequate to test the predicate φ with respect to the fault class C if and only if $\forall \mathsf{tp} \in \Gamma_C(\varphi) \exists t \in \mathcal{T} : t \models \mathsf{tp}$.

3. Generating Fault Detecting Test Cases

This section presents a general method to derive test cases for specific fault classes. Given a Boolean predicate φ and a fault class C, one can easily derive the set of test predicates $\Gamma_C(\varphi)$. Given any test predicate $\mathsf{tp} \in \Gamma_C(\varphi)$ representing a concrete fault of the predicate, Definition 2 reduces the problem of finding a test case that covers the test predicate to the problem of finding a model for a Boolean formula (tp).

Finding a model of a Boolean expression, if there exists one, can be efficiently solved by means of several techniques. Previously [18], we used a model checker for operational specifications, which, however, is less efficient for our current scenario considering that here we deal only with test predicates that are simple Boolean expressions. In this paper we evaluate SAT and SMT solvers, which can solve the problem of satisfiability of a Boolean expressions efficiently, and they are therefore well suited in this setting.

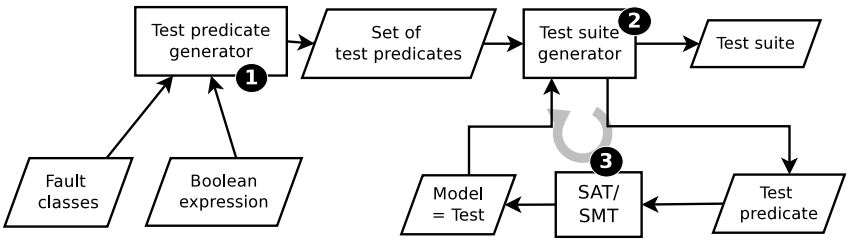

Figure 2: The basic process of generating tests: Given a set of fault classes and Boolean expressions, test predicates are generated (1). The test suite generator (2) uses a SAT or SMT solver to look for a model for each of these test predicates; if a model exists, this can serve as a test case.

In this paper we investigate the use of both techniques, as both approaches have their advantages and disadvantages. Note that the choice of one algorithm over another should only influence the time taken to solve the problem and not the size, assuming models can be found for all feasible test predicates.

SAT solvers. SAT solvers are efficient tools for solving instances of the Boolean satisfiability problem. They, however, cannot immediately be employed in our process because: (1) not all SAT solvers return the actual model of a Boolean expression; some of them just solve the satisfiability problem by checking whether a model exists or not without printing out this model, and (2) the SAT solvers available generally have their input Boolean expressions in CNF (Dimacs format) while our test predicates are an exclusive or between two general form expressions. The efficient conversion of a generic Boolean expression – containing the $\oplus$ operator – to a CNF formula is itself a research problem.

SMT solvers. SMT solvers apply to Satisfiability Modulo Theories (SMT) problems which are decision problems for logical formulas containing some functions and predicates symbols which have additional interpretations (like numbers, and so on). Generally SMT solvers are built on top of powerful SAT solvers and they attempt to solve SMT instances by translating them to Boolean SAT instances. The input language of SMT solvers is surely powerful enough to allow complete use of Boolean operators, including the $\oplus$ operator.

The entire process of generating a test suite using a SAT/SMT solver for test case generation is depicted in Figure 2. The test predicate generator **(1)** takes the Boolean predicate φ and the fault classes and generates a list of test predicates $\Gamma_{C_i}(\varphi)$ for every fault class C_i. The generation of test predicates consists of first taking the original predicate φ and applying the mutation operator for the fault class C_i in order to obtain all the possible faulty versions φ'_k of φ. Then, test predicates are obtained by simply combining the original predicate with all the mutants as $\mathsf{tp}_k = \varphi \oplus \varphi'_k$. The test suite generator **(2)** takes *one* test predicate tp that has not been considered yet and finds a test case t that satisfies the chosen test predicate **(3)**, i.e., $t \models \mathsf{tp}$. By iterating the activities

(2) and (3), one can build a test suite that is adequate to cover φ with respect to the desired fault classes.

3.1. Feasibility problem

Not all faults of a fault class can be distinguished from the original Boolean predicate: For some faults φ' of a predicate φ it may be the case that for any model $t \models \varphi$ it also holds that $t \models \varphi'$ and vice versa. In this case, φ' is logically equivalent to φ. In mutation testing, such faults are referred to as *equivalent mutants*, and in the general case of program mutants, detecting equivalent mutants is not decidable. Under the assumption that the Boolean space is complete, i.e., there are no constraints among the inputs and every combination of the Boolean variables is feasible, equivalent faults of Boolean predicates can be detected by the SAT/SMT solver by proving that the test predicate $\mathsf{tp} = \varphi \oplus \varphi'$ is unsatisfiable, i.e., $\nexists t : t \models \mathsf{tp}$ or, briefly, $\not\models \mathsf{tp}$. We say that the test predicate is infeasible. Taking infeasible test predicates into account requires a modified version of Definition 2:

Definition 3. Fault Detecting Adequacy with Infeasible Test Predicates. The test suite $\mathcal{T}$ is adequate to test the predicate φ with respect to the fault class C if and only if $\forall \mathsf{tp} \in \Gamma_C(\varphi)\,(\not\models \mathsf{tp} \vee \exists t \in \mathcal{T}\ t \models \mathsf{tp})$.

Note that equivalent faults consume time during test case generation in order to be detected but do not contribute to the resulting test suite. The main problem in our scenario is in the case when a SAT/SMT solver takes a long time to find a solution and is timed out by the user or the system – in this case it is not known whether the fault is equivalent or not.

3.2. Dealing with constraints over the variables

In general, some input combinations may be infeasible due to dependencies over the inputs of the Boolean expression. For example, when a predicate is embedded in source code, there might be constraints on the variables that are required to hold for the predicate to be *reached* in the first place. Deriving the constraints (e.g., using symbolic execution [27]) and determining inputs that drive variables to appropriate values is out of the scope of this paper, but such constraints can easily be integrated into our approach.

Generally, these dependencies are included in original Boolean specification as conjoints (e.g., [41]). However, this approach cannot distinguish assignments to the Boolean inputs which make the condition fault from those which are simply not valid. Furthermore, it may result in further tests because also the constraints are considered in the test generation process as they were part of the condition under test. Including them in the specification is necessary only if one wants to find faults also in the constraints, otherwise they should not be part of the expression under test.

Example 3. Consider for example the expression 20 used by Weyuker et al. [41] $\bar{e}f\bar{g}\bar{a}(bc+\bar{b}d)$. Weyuker et al. [41] discovered in the specification that conditions c and d could never be both true, so they transformed the original specification into specification 9 as $(\mathbf{c\bar{d}})\bar{e}f\bar{g}\bar{a}(bc+\bar{b}d)$.

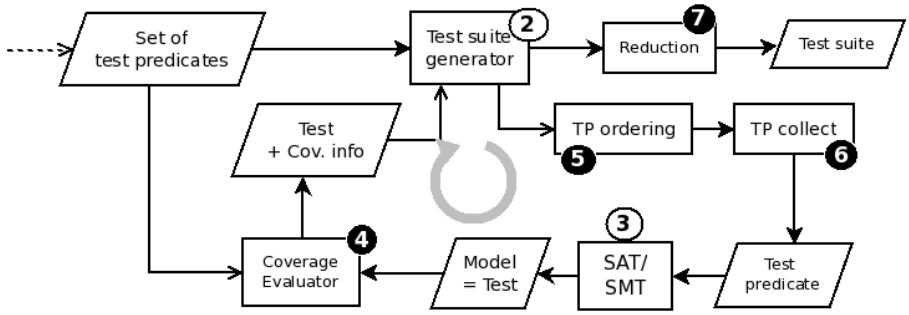

Figure 3: The improved test case generation process: By monitoring coverage (4) generation of redundant tests can be avoided; the order of test predicates (5) can affect the number of iterations and therefore of the tests; several test predicates can be collected to conjoint test predicates (6), further reducing the number of tests; and after the test case generation the resulting test suite can be minimized (7).

Our approach can be extended by considering also the constraints among the variables in φ *without* including them in φ. Given the constraints $\delta_1, \cdots, \delta_n$ as Boolean expressions which must be satisfied by any valid test, we search for tests which cover the test predicates while satisfying all the constraints.

Definition 4. Fault Detecting Adequacy in the Presence of Constraints. Let φ be a Boolean predicate and $\Delta = \delta_1 \wedge \cdots \wedge \delta_n$ the constraints over the variables in φ. The test suite $\mathcal{T}$ is adequate to test φ with respect to the fault class C if and only if $\forall \mathsf{tp} \in \Gamma_C(\varphi)\,(\not\models (\Delta \wedge \mathsf{tp}) \vee \exists t \in \mathcal{T}\ t \models (\Delta \wedge \mathsf{tp}))$.

Our method is still capable of finding valid tests, as models for the test predicates and the constraints. Note that φ' may be an equivalent mutant of φ even though $\varphi \oplus \varphi'$ has a model but such model does not satisfy the constraints Δ. In this case, the SAT/SMT solver can still discover equivalent faults by proving that $\Delta \wedge (\varphi \oplus \varphi')$ is infeasible.

4. Improving the Test Case Generation Process

The basic process of generating test cases consists of generating a set of test predicates for a given Boolean predicate and a set of fault classes and then deriving one test case per test predicate. This process can be improved with respect to the number of test cases generated by several activities, as summarized in Figure 3.

4.1. Monitoring coverage ((4) in Figure 3)

A test case explicitly generated for one test predicate may satisfy a number of further test predicates (*collateral coverage*). Consequently, it is not strictly necessary with respect to achieving the test objective (i.e., satisfaction of all test predicates) to generate test cases for all test predicates. Instead, each time a test case is generated the remaining uncovered test predicates can be checked

against the new test case (i.e., they are *monitored* for satisfaction), and any satisfied test predicate can be omitted from test case generation because it is already covered.

Checking whether a test predicate tp is covered by a test case t already generated for another test predicate simply requires evaluating the test predicate with the model that t represents. If $t \models \mathsf{tp}$ then t also covers tp. This process is usually cheaper than running a SAT/SMT solver on each test predicate, even if the number of test predicates is large.

4.2. Ordering test predicates ((5) in Figure 3)

When monitoring is applied, the order in which test predicates are selected may impact the size of the resulting test suite. In theory, there might be cases where choosing a single test predicate leads to satisfaction of all other test predicates, and other cases where a bad order leads to one test case for every test predicate. We have previously investigated the ordering of test predicates [16], showing that the ordering of the test predicates can have an impact on the number of test cases generated. Some orders that are applicable to the test predicates for fault classes are:

1. *Random order:* We use random order as a sanity check; any feasible heuristic should achieve better results. Otherwise a strategy to achieve good results is to use several runs with different random order and pick the best result, which minimizes the risk that a bad ordering leads to larger test suites. Our previous research [16] showed that it is difficult to find a heuristic that improves over the average random case.

2. *Subsuming order*: If the subsumption relation between fault classes is known, or at least a subsumption relationship is suspected to be in place due to some empirical data, one can choose a test predicate ordering depending on that relation. The hierarchies of fault classes for specification-based testing have been established to prioritize test cases so as to achieve earlier detection of more faults [32]. Fault classes could be used before the classes they subsume in order to reduce the number of test cases that are generated (if F_1 subsumes F_2, the test cases for F_1 will cover also the test predicates for F_2). For example, CCF and CDF subsume most of all the other fault classes presented in Section 2.2, so subsuming order would start with test predicates from these fault classes. Note that in this paper we use the corrected version of the fault hierarchy which takes into account also the feasibility problem (see Section 3) of testing criteria and therefore the order will be ASF, CCF, CDF, VRF, SA0, SA1, MVF, VNF, ORF, and ENF.

4.3. Collecting test predicates ((6) in Figure 3)

Instead of generating one test case for each test predicate, one can *collect* many test predicates [2] in a unique conjoint in a way that a model for the conjoint is a model of all the collected test predicates, as proved by the following theorem:

Theorem 1. *Given a test predicate* $\mathsf{TP} = \mathsf{tp_1} \wedge \ldots \wedge \mathsf{tp_n}$*, a model* t *of* TP *(i.e.* $t \models \mathsf{TP}$*) is a model for* $\mathsf{tp_1}, \ldots, \mathsf{tp_n}$*.*

PROOF. The proof for Theorem 1 follows directly from the interpretation of $\wedge$: $t \models (\Phi \wedge \Psi)$ iff $t \models \Phi$ and $t \models \Psi$.

Consequently, one can collect many test predicates not covered yet and generate one test case that covers them all. However, when collecting test predicates we must add a test predicate tp to the collected TP only if it is *consistent* with TP, i.e., there exists a model for both TP and tp. Furthermore, special care must be given to infeasible test predicates: Since they are never consistent with any other test predicate they should be detected as early as possible to avoid repeatedly trying to collect them. The resulting process of collecting is presented in Algorithm 1, which shows the single activity of obtaining a collected test predicate and its test case from a set of test predicates TPS.

The algorithm works on the set TPS of test predicates that still need to be considered. In a loop it randomly chooses one test predicate tp out of this set at a time, and checks if there exists a model for the conjunction of the variable constraints Δ, the set of selected test predicates C, and the newly selected test predicate tp. If there is a model then this tp is covered and removed from TPS and added to C, else we need to check if tp by itself is feasible or not. If there is no model that satisfies tp and the constraints Δ, then we know it is infeasible and can remove it from TPS. In the end, the algorithm returns a test case that is a model for the set of selected test predicates in C, while the remaining test predicates are still in TPS. If one does not bound the number of test predicates that can be collected at a time, then after the first run all infeasible test predicates are removed from TPS and the feasibility check can be omitted in the next run. The algorithm also removes from TPS infeasible test predicates and predicates that are covered because they are collected. Initially TPS contains all the test predicates and the algorithm must be iterated until TPS becomes empty.

4.4. Post reduction (minimization, (7) in Figure 3):

A test suite is minimal [20] with regard to an objective if removing any test case from the test suite will lead to the objective no longer being satisfied. The problem of finding the optimal (minimal) subset is NP-hard, which can be shown by a reduction to the minimum set covering problem [17]. In this paper, we use a simple greedy heuristic to the minimum set covering problem for test suite minimization: The heuristic selects the test case that satisfies the most test predicates and remove all test predicates satisfied by that test case. This is repeated until all test predicates are satisfied.

Monitoring and minimization[1] can behave very differently: Minimization requires existing, full test suites while monitoring checks test predicates on the fly

[1] In the remainder of the paper, we will use the terms "minimization" and "post reduction" interchangeably.

Algorithm 1 collection process

Require: TPS : set of all the test predicates to be considered
  $C \leftarrow \{\}$
  for tp $\in$ TPS **do**
    if $\exists t : t \models \Delta \wedge \bigwedge_{c \in C} c \wedge$ tp **then**
      $C \leftarrow C \cup \{\text{tp}\}$
      TPS $\leftarrow$ TPS$\backslash\{\text{tp}\}$ {tp is covered}
    else if $\nexists t : t \models \Delta \wedge$ tp **then**
      TPS $\leftarrow$ TPS$\backslash\{\text{tp}\}$ {tp is infeasible}
    else
      { tp cannot be collected together with C }
    end if
  end for
  return $t : t \models \Delta \wedge \bigwedge_{c \in C} c$

during test case generation. On the other hand, monitoring does not guarantee minimal test suites. Note that like any reduction strategy, the post reduction may reduce the fault detection capability of the test suite, but not with respect to the fault classes it initially covered since the set of test predicates covered remains the same.

5. Experiments

To evaluate the approach described in this paper, we performed a set of experiments, trying to answer the following research questions:

- What are the effects of the described optimizations?
- How does the presented approach compare to other criteria?

5.1. Experimental setup

Boolean expressions. For experimentation, we considered the same set of predicates introduced by Weyuker et al. [41], who selected 13 Boolean conditions from the specification of TCAS II, an aircraft collision avoidance system. They also added 7 specifications after having identified variable dependencies. This set was originally used by Weyuker et al. [41] to evaluate several testing criteria and generation techniques, and the same set (except for one expression – number 12 – which contains a typo) is still commonly used as benchmark for test generation techniques.

Chen et al. [3] translated these Boolean specifications to irredundant disjunctive normal form to allow the application of the testing criterion MUMCUT, which was later improved by Yu et al. [44]. The same set has also been used by Kobayashi et al. [28] for evaluating the combinatorial and random/anti-random approaches to test generation, and by Kaminski and Ammann [24] to evaluate the Minimal-MUMCUT strategy.

For the 7 specifications with the constraints representing input dependencies, we use two versions: one with the constraints added as conjoints (as originally done by Weyuker et al. [41] and in literature thereafter), and another with the constraints modeled separately, as explained in Section 3.2. In total, we have 26 specification: 12 expressions without constraints, 7 with the constraints added as conjoint and 7 with the constraints explicitly modeled. We ran the experiments on a Linux cluster with 2 Intel(R) Xeon(R) CPUs E5430 @ 2.66GHz and 16 GB of RAM

SAT/SMT solvers. We have chosen as SAT solver SAT4J [33], because SAT4J is a mature, open source library of SAT-based solvers in Java. The library has been adopted by several academic projects, and it provides SAT, MaxSat and pseudo-Boolean solvers for easy integration in Java programs. Those solvers have been evaluated regularly in the corresponding international competitive events.

As SMT solver, we decided to use Yices [13], which we have also used in the past and which includes a very efficient SAT solver; it claims to be *"competitive as an ordinary SAT and MaxSAT solver"* [13].

Test predicates. We have generated the test predicates for all the fault classes presented in Section 2.2, obtaining 12752 test predicates (with an average of 490 test predicates per expression). Table 1 lists details of the expressions and resulting test predicates. The fault classes with most test predicates are VRF with 4209 (33% of the total) and CDF with 3616 (28%).

We generated test suites for all expressions using all different combinations of options; Table 1 summarizes the sizes of the resulting test suites. To cover all the test predicates we needed from a minimum of 508 to a maximum of 10249 tests, depending on the options used. The minimum time for completing the test generation of all the test predicates and to discover the infeasible test predicates was 3.38 seconds, while the maximum was 8.5 hours. We found that an average of 18% of test predicates are infeasible, and only ASF has no infeasible test predicates. The fault classes with higher percentage of infeasible test predicates are CDF with 52%, CCF with 19%, and VRF with 18%. All the other classes had a very low percentage of infeasible test predicates.

5.2. Experiment 1: Which options are best

Given the high variability in size and time of the test generation process, we wanted to asses the influence of the options we have identified in the previous sections on the test generation process: Post reduction or minimization {yes/no}, monitoring {yes/no}, collecting {yes/no}, generation tool or solver {Yices/SAT4J}, and ordering {subsume, random}. The hypothesis we want to test is that each option has an impact over the test generation process in terms of the size of the final test suite and the time taken to obtain it.

We have run the test generation algorithm with all the options ($2^5 = 32$) for all the 26 specifications 30 times using different random seeds, obtaining 24960 completed runs. In this way, we obtained *balanced* observations (i.e., an equal

	Test Predicates		Tests			
Specification	Total	Infeasible	(1)	(2)	(3)	(4)
1	462	107	24	24.0	24	28.3
2	919	143	37	37.9	39	40.6
3	1,478	467	38	39.3	45	59.3
4	73	3	8	8	8	8
5	519	92	18	18.7	21	23.7
6	768	175	30	32.3	32	35.3
7	522	117	22	22.4	26	28.1
8	364	32	19	19	19	19.2
9	212	34	13	13	13	13
10	493	74	24	24.7	36	36.4
11	696	131	25	27.5	56	62.3
13	388	56	14	16.6	26	29.1
14	229	15	16	16.1	19	20.5
15	426	77	13	13	24	26
16	1,185	148	36	38.8	71	75.2
17	365	13	16	16	29	36.3
18	334	18	19	19.8	32	40.3
19	222	7	11	11.3	16	17.5
20	168	26	9	9.2	12	12
21	229	15	13	13	13	13.7
22	426	77	18	18.0	18	19.2
23	1,185	362	32	32.8	36	44.8
24	365	13	16	16.9	17	17.6
25	334	18	11	11.7	12	12.9
26	222	7	14	14	14	14.3
27	168	26	12	12	12	12
Σ	12,752	2,253	508	525	683	745

Table 1: Test predicates and resulting tests: (1) minimum number of tests, (2) mean of number of tests with all the best options activated, (3) minimum number of tests with all the options except collecting, (4) average number of tests with all the options activated except collecting

Optimization		Minimiz.	Monit.	Collecting	Solver	Order
All	*Size*	++	++	++	=	=
	Time	=	=	+++	+	=
All/Collecting	*Size*	+	=		=	=
	Time	=	=		+	=
All/No collecting	*Size*	+++	+++		=	=
	Time	=	+		++	=

Table 2: Kruskal–Wallis test output — p-value evaluation: higher + denotes higher significance level of the option (also called factor), = means that our hypothesis is rejected and the null hypothesis, i.e., the optimization had no effect, is accepted instead.

number of observations for every specification and option). For each run we have registered the final *size* and the *time* taken to complete it. We have applied analysis of variance on the size and time depending on the options; because we found that the distribution of time and size is not normal, we applied the Kruskal-Wallis one-way analysis of variance [29]. This is a non-parametric method for testing equality of population medians among two or more groups identified by factors. It does not assume a normal population, unlike the analogous one-way analysis of variance (anova). However, the test does assume an identically-shaped and scaled distribution for each group, except for any difference in medians. Because, in our case, size and time strongly depend on the size and structure of the specifications, we normalized the data with respect the data for the single specification under consideration before applying the Kruskal–Wallis test: We have computed $\mathsf{X}Norm = (\mathsf{X} - median_{spec}(\mathsf{X}))/(max_{spec}(\mathsf{X}) - min_{spec}(\mathsf{X}))$ with X both *size* and *time*. $\mathsf{X}Norm = 0$ means that the value for the expression under test is equal to the median for that specification, while $\mathsf{X}Norm > 0 (< 0)$ means that a value greater (smaller) than the median is obtained. We then applied the test to the normalized data. Table 2 reports the qualitative evaluation of the p-values computed by the Kruskal–Wallis technique. Since we observed a strong correlation between each factor and the collecting option, we applied the Kruskal–Wallis test also to the subset of observations where collecting was applied and to the subset without collecting applied, and the evaluation of the p-value is reported in Table 2 as well.

We report here only a summary of the data collected during the experiments; Figure 4 illustrates the distribution and the mean (as text) of the size depending on the application of the options, and Figure 5 shows the average time for test suite generation for one expression depending on the options used.

Post reduction. As proved by Table 2, minimization has a significant effect on the test suite size while it is uninfluential regarding the time (its application requires a negligible amount of time). Its application proves to be effective in reducing the test size (see Figure 4). On the average, post reduction is able to reduce the test suite size by 81% (22.9 instead of 120.5 tests). Considering only

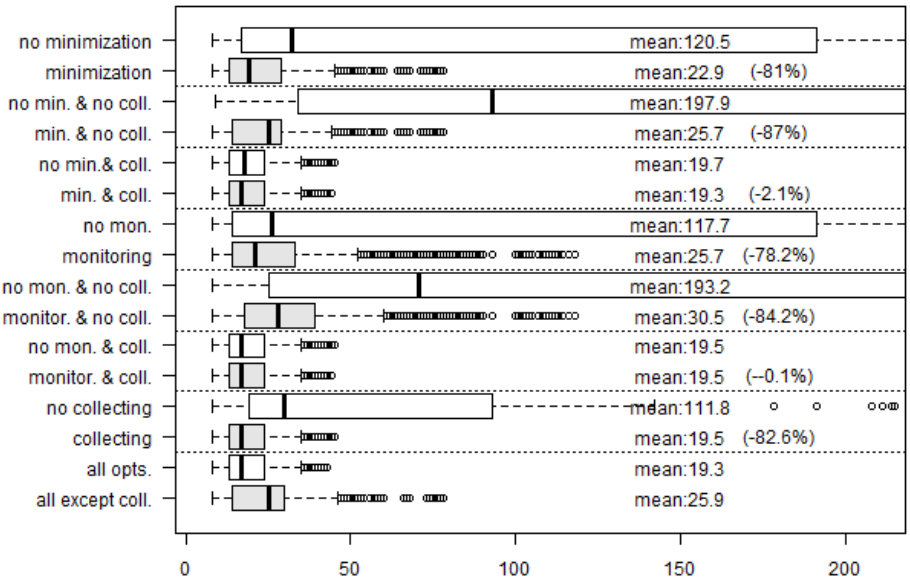

Figure 4: Size distribution and mean depending on the options

the tests in which collection is not applied, the average is 87%, and when used in conjunction with collection, it is very ineffective reducing the size only by 2%.

Monitoring. Monitoring also has an impact over the final size and it is effective in reducing it. The average size of the test suite with monitoring is 25.7 against 117.7 when monitoring is not applied, reducing the test suite size by 78%. Considering only the data where collection is not applied, monitoring improves on the average of 84% the final test suite size. However it is not influential (as proved also by a small F-value in Table 2) when collection is applied (improvements of 0%). Using monitoring with collection has no impact over the time. However, the time without collection is reduced to around 1/4 by monitoring (see Figure 5).

Collecting. Table 2 proves that collecting has a great impact over both the size and the time. Collecting proves to be a very powerful reduction technique which is able to reduce by itself the test suite size by 83% (see Figure 4). It is the most powerful reduction technique since it is able to produce by itself the smallest test suite on average (19.5). The other techniques alone are not able to produce test suites as small as those produced by collecting: Using all the optimizations techniques except collect, the average size is 25.9 (+ 33%). Collecting reduces the effectiveness of other techniques, although when used in combination of collecting they produce even smaller test suites. All optimizations together are able to obtain the smallest test suites we observed, with an average size of 19.38. While the other optimization options improve both the size and the time, collecting

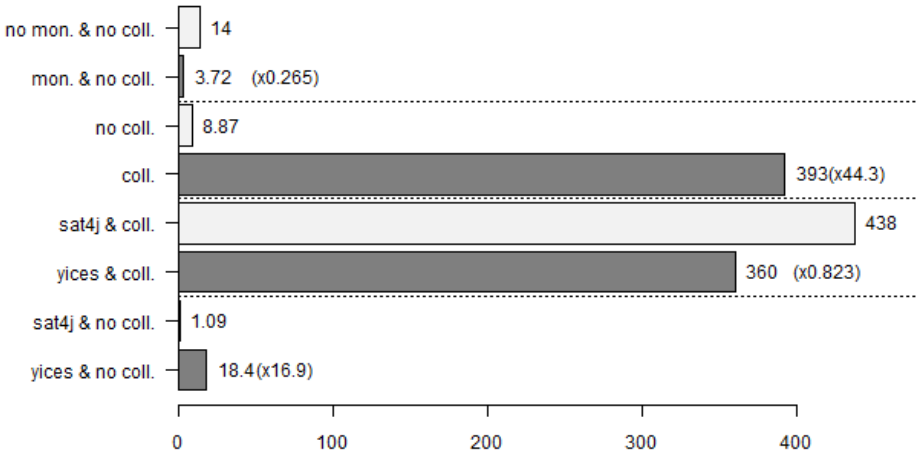

Figure 5: Average time in seconds to complete the test generation for one expression depending on the options

reduces the size but increases the time. This technique is computationally very expensive, taking on average 44 times as long as without it. The user should choose between optimization of the test suite size vs. having the test faster: Applying monitoring and reduction without collect the average time is around 3 seconds instead of 393 seconds with collect.

Solver. As expected, the use of one solver instead of the other has no impact on the size of the test suites. In our experiments, the choice of solver had only a minimal impact on the time. When using collection, Yices performed better than SAT4J, while without collection SAT4J performs significantly better than Yices. However, we must consider a number of issues when using Yices: While SAT4J offers an API we can access directly from our tool, we had to call Yices by exchanging files and running it as external program in a shell, and this may reduce its responsiveness. We are working on integrating Yices directly in our code using Java Native Interface (JNI).

Ordering. The order in which test predicates were considered had no impact on the size and the time. In our previous work [16], we found that test predicate ordering may have impact over the test generation process, but it is difficult to devise an efficient heuristic to order the test objectives. Previously [15], we found that subsumption was a good criterion to reduce the test suite size, but in that case we considered DNF Boolean specifications, which have a stronger fault hierarchy. For Boolean expressions in general form, the hierarchy is not so strong as explained in Section 2.2 and it is not so useful.

5.3. Experiment 2: Comparison with other testing criteria

We have compared our approach in terms of test suite size and fault detection capability with the following testing criteria by considering the original 19

Boolean TCAS expressions (1 to 20). *** The hypothesis we want to test is that our method performs better than classical test generation techniques for Boolean expressions in terms of test suite size and fault detection capability.

MUMCUT. Chen et al. [3] developed the MUMCUT coverage criterion specifically to guarantee detection of all the faults of the seven fault classes (which resemble our fault classes given for GF, and which are considered the classical fault classes for DNF) in Boolean expressions given in irredundant disjunctive normal form (iDNF). The MUMCUT strategy integrates three constituent criteria: the Multiple Unique True Point (MUTP), Multiple Near False Point (MNFP), and Corresponding Unique True Point Near False Point (CUTPNFP) criteria. MUMCUT already proved to be able to reduce the test suite size with respect to MAX-A and MAX-B [41] by 30% to 40%. Yu et al. [44] improved MUMCUT and defined several versions; for comparison we also use their MUMCUT G-CUN strategy.

Minimal-MUMCUT. Later, Kaminski and Ammann [24] introduced the Minimal-MUMCUT strategy which improves the MUMCUT algorithm by taking into account the feasibility problem of the three testing constituents of the MUMCUT strategy. They found that Minimal-MUMCUT reduces the test suite size without sacrificing any fault detection capability with respect to the original MUMCUT. Note, that also Minimal-MUMCUT accepts specifications only in irredundant DNF.

MCDC. The Modified Condition Decision Coverage (MCDC), originally introduced by Chilenski and Miller [8], accepts Boolean expression in a general form and requires a test suite such that every condition in the expression is shown to independently affect the final outcome of the expression. This criterion is mandated for safety critical aviation software by the RCTA/DO-178B standard. Its precise interpretation has been the subject of study for many years, in this paper we used *masking* MCDC [9].

Table 3 reports the test suite size of our fault-based method with and without collection, the size for the MUMCUT variants considered, and for MCDC. For MUMCUT we used the data available in the literature, while for Minimal-MUMCUT we used a web application developed by the inventors of the criterion [22] and the translation to iDNF available in the literature. For MCDC, we used our own code. For Minimal-MUMCUT and MCDC we were able to also apply our post reduction technique introduced in Section 4.4, although that is not expected by their original definitions and the data is reported in Table 3 as well. As proved by Table 3, our technique even without collect was able to produce a test suite smaller than every other testing criteria except MCDC, which however does not target all the fault classes we consider. However, if one applies the minimization technique presented in this paper to Minimal-MUMCUT, he/she obtains a smaller test suite without reducing the fault detection capability with respect to the classes we have introduced. This proves that minimization can be efficiently employed by other test generation methods as well. However, we need

Technique	Form	#Tests
Fault-based all faults, all optimizations	GF	392
Fault-based all faults, all optimizations but collection	GF	548
MUMCUT [3]	iDNF	7,391
MUMCUT G-CUN [44]	iDNF	1,413.6
Minimal-MUMCUT	iDNF	936
MCDC	GF	285
Minimal-MUMCUT + minimization technique of Section 4.4	iDNF	268
MCDC + minimization technique of Section 4.4	GF	273
Fault-based SA, MVF, VNF, ORF, ENF faults, all optimizations	GF	229
Fault-based SA, MVF, VNF, ORF, ENF faults, all optimizations but collection	GF	260

Table 3: Comparison of the size with other testing criteria

to take a closer look at the fault detection capability of Minimal-MUMCUT and MCDC in comparison with our technique.

For Minimal-MUMCUT and MCDC, we were also able to compute their fault detection capability by evaluating the number of feasible faults covered by their test suite. Table 4 reports the ratio of faults detected for all the fault classes.

We found that Minimal-MUMCUT was able to detect only 64% of the faults by covering 5145 test predicates out of 8088 feasible test predicates for the original 19 TCAS Boolean expressions. For the CDF class, the test suite was able to detect only around 50%. Minimal MUMCUT was not able to detect all the faults in any class. Overall the quality seems questionable, because it required 70% of the tests of the fault-based approach to cover only 64% of the faults.

This provides further evidence that testing Boolean expressions in general form (GF) using the IDNF-oriented approaches may result in bigger test suites while missing many faults. This problem has been already presented by Chen et al. [6], which found similar shortcomings of the iDNF testing approaches.

Note that there exists a similar empirical study [5] which investigates the fault detection capability of the *non* minimized MUMCUT strategy with respect to general form Boolean expressions as well as mutated expressions. It showed that over 99% of the mutants were killed by MUMCUT test sets among all fault types of the original GF expressions. It seems that minimizing the test suite for iDNF expressions, as proposed by Minimal-MUMCUT, even if that does not change the fault detection capability with respect to the DNF faults, it reduced the fault detection capability with respect to the original GF expressions.

MCDC detected all the faults in 6 classes confirming the theoretical studies

Fault class	Min. MUMCUT	MCDC
ASF	93%	93%
CCF	66%	91%
CDF	51%	58%
VRF	64%	95%
SA0	66%	100%
SA1	68%	100%
MVF	60%	100%
VNF	77%	100%
ORF	75%	100%
ENF	89%	100%
Σ	64%	87%

Table 4: Percentage of faults detected by other testing criteria

presented by Kapoor and Bowen [26]. Overall, MCDC covered 87% of the faults with 69% of tests of the fault-based approach, which seems a positive aspect of MCDC. However, we have generated with our method the tests for the same subset of faults MCDC guarantees to detect. The last two rows of Table 3 list the resulting test suite sizes: Both with and without collecting, our method is able to generate a smaller test suite than that produced by MCDC. This proves that our approach is competitive with *any* testing criterion because the set of targeted faults can be easily modified, and still the resulting test suite is compact.

5.4. Threats to validity

Threats to *construct validity* are on how the performance of a testing criterion is defined. An intrinsic feature of our approach is that all considered fault classes are guaranteed to be covered, so we focused in our evaluation on test suite size and generation time. In practice, a concrete testing scenario might have different requirements. For example, given a limited amount of time, it is not clear which criterion will achieve the highest fault coverage. However, by offering different optimizations one can adapt our technique to different demands of different testing scenarios.

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the risk of having faults in our testing framework, it has been carefully tested. As randomized algorithms (such as the ordering) are affected by chance, we ran each experiment 30 times. Another possible threat to internal validity might come from suboptimal use of the SAT and SMT solvers.

Although we used the standard benchmark for testing Boolean expressions, there is the threat to *external validity* regarding the generalization to other specifications and types of software, which is common for any empirical analysis.

*** Moreover, we assume that only a *single* fault may occur in the expression under test. Multiple faults in a single expression should be rare accord-

ing to the competent programmer hypothesis DeMillo et al. [12], which states that programmers tend to develop programs very close to the correct version. According to the *coupling effect* [12] hypothesis, multiple faults in a single expression should also be detected by tests generated for single faults. However, interactions between individual faults in a multiple fault scenario may still lead to faults that pass undetected by our test suites. In general, all the minimization techniques we presented may reduce the *residual* fault detection capability, that is the capability of detecting faults not directly targeted by the testing criterion. Other criteria, like MCDC or MUMCUT, while leading to bigger test suites, may have a better residual fault detection capability than ours. We plan to adapt our technique to deal with multiple faults and to compare it in this case with other testing criteria.

6. Related Work

This paper joins fault-based testing with testing of Boolean expressions. Fault-based testing methods first hypothesize certain types of faults that may be committed by programmers, and then design test cases targeted at these faults [35]. In contrast to other testing methods, fault-based testing can demonstrate the absence of hypothesized faults. There has been an increasing interest in the use of a fault-based approach to generate test cases from software specifications in the past years [18, 25]. Since logic expressions are extensively used in both software specifications and codes, many investigations of fault-based testing have been conducted on logic Boolean expressions [14, 40, 41, 3, 23].

For this reason there exist many testing criteria for Boolean expressions. Traditional criteria include (using the nomenclature common in white box testing literature) decision coverage, condition coverage, or multiple condition coverage. Modified condition/decision coverage (MCDC [8]) is a popular criterion that gives a compromise between the large number of test cases that multiple condition coverage creates and the fault detection capability offered by that amount of test cases. Several testing criteria have been developed to directly target common faults in Boolean expressions. For instance Weyuker et al. [41] introduced a family of strategies for automatically generating test cases from Boolean expressions. The MAX-A and MAX-B strategies are the most powerful in this family of criteria, as they subsume all the others.

Chen and Lau [4] introduced three testing criteria: the *Multiple Unique True Point (MUTP), the Multiple Near False Point (MNFP), and the Corresponding Unique Tree Point and Near False Point par (CUTPNFP)* strategies. Chen et al. [3], Yu et al. [44] introduced the MUMCUT testing strategy which integrates MUTP, MNFP and CUTPNFP and (1) guarantees to detect seven types of fault in Boolean expressions in irredundant disjunctive normal form as MAX-A and MAX-B, and (2) requires only a subset of the test suites that satisfy the previously proposed MAX-A and MAX-B strategies. Kaminski and Ammann [24] presented a new extension of MUMCUT, called Minimal-MUMCUT. Minimal-MUMCUT takes into account the feasibility of the three components of MUMCUT and guarantees to detect the same types of faults with fewer test

cases. There exists another attempt to reduce the size of MUMCUT test suites, which employs a SAT solver [42]. In this case SAT is used to improve the MUMCUT generation process and although some advances are reached, in the few cases published, their test suite was on average 2.5 times the size of test suites generated by our method.

Because Boolean expressions in realistic programs or specification are often not in the restricted form and the faults are introduced in the context of general form (GF), there exist several studies to assess the fault detection capability of DNF testing approaches and several attempts to explicitly target faults in GF expressions. Chen et al. [5] found that a fault in general form Boolean expression induces a large number of possible faults in the corresponding iDNF, but that MUMCUT was able to detect from 97.8% to 99.5% of the faults. A similar result is presented by Sun et al. [39]. However, the size of the test suites is not considered and, as proved in this paper, optimizing the size of the MUMCUT strategy reduces its fault detection capability. This could mean that the non minimized MUMCUT test suite has many tests which are useless with respect to the DNF faults, but still can detect faults in the GF counterparts.

There are some attempts to extend the DNF testing criteria to GF expressions, in order to avoid the expensive translation to DNF, which increases the size of the specification and of the test suites and hides some faults such that they could pass undetected. Sun et al. [38] analyze MUMCUT to find how to improve it to detect all the faults in GF expressions. Chen et al. [6] introduce two general fault-based testing strategies, called general meaningful impact strategies GBMIS and GMMIS, which extend the BMIS [41] and MUMCUT strategies to GF expressions. Applied to the TCAS expressions, GBMIS could not detect all the faults, while GMMIS achieved 100% mutation score. However, the GMMIS test suite was about 86% of the one of MUMCUT, i.e., around 6,000 tests – still much bigger than the test suites found by our method.

Several other testing criteria target faults in Boolean expressions regardless of their form. A survey on most logic based testing criteria is presented by Kaminski et al. [23] (12 testing criteria are considered) where the authors distinguish between semantics criteria (like MCDC) and syntactic criteria (like MUMCUT). They also discuss the test suite size and the fault detection capability of the testing criteria assessing their subsumption relationships.

A comparison among MCDC, MUMCUT, and other testing criteria is presented by Yu and Lau [43]: MUMCUT outperformed MCDC in detecting faults in DNF and GF Boolean expressions (including those from the TCAS case study). However, again, the test suite size is not considered and MUMCUT is not minimized.

Also in hardware testing, recent advances in Boolean-satisfiability (SAT) solvers are increasingly rendering SAT-based automatic test generation for combinatorial circuits, called *test pattern generation* (ATPG), an attractive alternative to traditional structural approaches (like the D-algorithm). Larrabee introduced the use of the Boolean difference for ATPG [31], and his approach is identical to ours regarding the use of the exclusive or to model the detection conditions. However, hardware circuits differ from Boolean expressions in soft-

ware because they have a simpler fault model (mainly only stuck at true and stuck at false) while they have hundreds ports and inputs (and generally also multiple outputs). For this reason, the research in hardware testing has tried to improve SAT algorithms and combine them with the efficiency of structural algorithm (like the D-algorithm in [37]) by exploiting information about the faults in order to improve the generation efficiency in terms of the time taken to generate the final test suite. These techniques do not use test predicates as we do in this paper.

Recently, the problem of test minimization, in hardware testing called *compaction*, has been intensively studied, although not in combination with SAT-based ATPG. Traditionally, there is a distinction between static and dynamic compaction: Static compaction [19] starts with a generated test set and produces a smaller test set which detects (at least) the same faults as the original test set. This technique is similar to our post reduction algorithm, but it relies on different heuristics. For example, it generally tries to detect or to find in the tests the *don't care* values which in hardware testing methods like the D-method are frequent. Dynamic compaction considers test set minimization during the test generation process by generating test patterns which detect multiple faults, and it is somehow analogous to our collection technique. There exist some attempts [10] to apply dynamic test compaction combined with the use of SAT solving and extracting testing conditions with D-chains. This technique, in which *fault groups* correspond to our collected test predicates, heavily relies on the simple fault model for circuits and again it does not use test predicates for test generation and it is optimized for test generation time.

7. Conclusions

Boolean expressions are an essential part of programs and specifications. Boolean expressions are also an important source of faults. Therefore, thorough testing of Boolean expressions is necessary. In this paper, we described a fault-based approach that improves over the state of the art as it does not require a particular format for the Boolean expressions, but guarantees that all faults of the considered fault classes are detected. Although the possible number of faults can be large, several optimization techniques lead to overall test suites that are smaller than those produced by other criteria.

In addition to often resulting in smaller test suites, this approach has the advantage that new fault classes can be added or removed (for example if there is the knowledge of a particular error pattern) and targeted without the need to introduce new test criteria, for which the fault detection capability has to be investigated. All that is necessary in order to support an additional fault class is to define a mutation operator that represents the fault class and creates faults usable for the test predicates. In contrast, introducing a new test criterion targeting a specific fault class is much more difficult than just defining the mutation operator for that class.

The findings presented in this paper have implications for several fields of software testing where Boolean expressions occur. A main application is test

case generation from specifications; this is the context where most previous work on testing Boolean expressions focused on. However, Boolean expressions also exist in normal source code, and these expressions need to be tested as well.

[1] S. B. Akers. On a theory of boolean functions. *Journal of the Society for Industrial and Applied Mathematics*, 7(4):487–498, 1959.

[2] A. Calvagna and A. Gargantini. Combining Satisfiability Solving and Heuristics to Constrained Combinatorial Interaction Testing. In *TAP'09: Proceedings of the 3rd International Conference on Tests and Proofs*, pages 27–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] T. Chen, M. Lau, and Y. Yu. MUMCUT: A fault-based strategy for testing boolean specifications. In *Asia-Pacific Software Engineering Conference*, page 606, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[4] T. Y. Chen and M. F. Lau. Test case selection strategies based on Boolean specifications. *Software Testing, Verification and Reliability*, 11(3):165–180, 2001.

[5] T. Y. Chen, M. F. Lau, K. Y. Sim, and C. A. Sun. On detecting faults for Boolean expressions. *Software Quality Journal*, 17(3):245–261, 2009.

[6] Z. Chen, B. Xu, and C. Nie. Comparing Fault-based Testing Strategies of General Boolean Specifications. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 621–622, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Z. Chen, T. Y. Chen, and B. Xu. A revisit of fault class hierarchies in general boolean specifications. *ACM Transactions on Software Engineering and Metholodogy*, 2010.

[8] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5), 1994.

[9] J. Chilenski and L. A. Richey. Definition for a masking form of modified condition decision coverage (MCDC). Technical report, Boeing, 1997.

[10] A. Czutro, I. Polian, P. Engelke, S. M. Reddy, and B. Becker. Dynamic compaction in SAT-based ATPG. *Asian Test Symposium*, 0:187–190, 2009.

[11] R. A. DeMillo and J. A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17:900–910, 1991.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[13] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI Available at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[14] K. A. Foster. Sensitive test data for logic expressions. *SIGSOFT Softw. Eng. Notes*, 9:120–125, April 1984.

[15] G. Fraser and A. Gargantini. Generating minimal fault detecting test suites for boolean expressions. In *6th Workshop on Advances in Model Based Testing A-MOST 2010*. IEEE Computer Society, 2010.

[16] G. Fraser, A. Gargantini, and F. Wotawa. On the order of test goals in specification-based testing. *Journal of Logic and Algebraic Programming*, 78(6):472–490, July 2009.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[18] A. Gargantini. Using model checking to generate fault detecting tests. In *TAP'07: Proceedings of the 1st International Conference on Tests and Proofs*, volume 4454 of *Lecture Notes in Computer Science (LNCS)*, pages 189–206. Springer Verlag, 2007.

[19] I. Hamzaoglu and J. H. Patel. Test set compaction algorithms for combinational circuits. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, ICCAD '98, pages 283–289, New York, NY, USA, 1998. ACM.

[20] M. J. Harrold, R. Gupta, and M. L. Soffa. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[21] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions of Software Engineering Methodology*, 5(3):231–261, 1996.

[22] G. Kaminsk and N. Lii. Minimal mumcut coverage web application. `http://cs.gmu.edu:8080/offutt/coverage/MinimalMUMCUTCoverage`, 2010. [Online; accessed 15-Dec-2010].

[23] G. Kaminski, G. Williams, and P. Ammann. Reconciling perspectives of software logic testing. *Software Testing, Verification and Reliability*, 18(3): 149–188, 2008.

[24] G. K. Kaminski and P. Ammann. Using Logic Criterion Feasibility to Reduce Test Set Size While Guaranteeing Fault Detection. In *ICST'09: Proceedings of the 2nd International Conference on Software Testing Verification and Validation*, pages 356–365, Washington, DC, USA, Apr. 1–4, 2009. IEEE Computer Society.

[25] K. Kapoor and J. P. Bowen. Test conditions for fault classes in Boolean specifications. *ACM Transactions on Software Engineering and Methodology*, 16(3):10, 2007.

[26] K. Kapoor and J. P. Bowen. A formal analysis of MCDC and RCDC test criteria. *Software Testing, Verification and Reliability*, 15(1):21–40, 2005.

[27] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19 (7):385–394, 1976.

[28] N. Kobayashi, T. Tsuchiya, and T. Kikuno. Non-specification-based approaches to logic testing for software. *Information and Software Technology*, 44(2):113 – 121, 2002.

[29] W. H. Kruskal and W. A. Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47 (260):583–621, 1952. ISSN 01621459. doi: 10.2307/2280779. URL `http://dx.doi.org/10.2307/2280779`.

[30] D. R. Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, Oct. 1999.

[31] T. Larrabee. Test pattern generation using boolean satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 11(1):4 –15, Jan. 1992.

[32] M. F. Lau and Y.-T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 14(3):247–276, 2005.

[33] D. Le Berre and A. Parrain. The SAT4J library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation(JSAT)*, 7:59–64, 2010.

[34] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, Sept. 1994.

[35] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, Aug. 1990.

[36] V. Okun, P. E. Black, and Y. Yesha. Comparison of fault classes in specification-based testing. *Information and Software Technology*, 46:525–533, 2004.

[37] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(9):1167–1176, 1996.

[38] C. Sun, Y. Dong, R. Lai, K. Y. Sim, and T. Y. Chen. Analyzing and extending MUMCUT for fault-based testing of general boolean expressions. In *Sixth IEEE International Conference on Computer and Information Technology (CIT)*, page 184. IEEE Computer Society, 2006.

[39] C.-A. Sun, K. Y. Sim, T. H. Tse, and T. Y. Chen. An empirical evaluation and analysis of the fault-detection capability of MUMCUT for general Boolean expressions. In *Proceedings of International Computer Symposium (ICS 2004)*, pages 926–932, Taipei, Taiwan, 2004.

[40] K. Tai. Condition-based software testing strategies. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pages 564 –569, 1990.

[41] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994. ISSN 0098-5589.

[42] J. Yan and J. Zhang. SAT based automated test case generation for MUMCUT coverage. In *The 17th IEEE International Symposium on Software Reliability Engineering (Student Program Papers) ISSRE*, 2006.

[43] Y. T. Yu and M. F. Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *J. Syst. Softw.*, 79:577–590, May 2006.

[44] Y. T. Yu, M. F. Lau, and T. Y. Chen. Automatic generation of test cases from boolean specifications using the MUMCUT strategy. *Journal of Systems and Software*, 79(6):820–840, 2006.