Generating minimal fault detecting test suites for Boolean expressions

Gordon Fraser
Software Engineering Chair
Saarland University, Saarbrücken, Germany
E-mail: fraser@cs.uni-saarland.de

Angelo Gargantini
Dip. di Ing. dell'Informazione e Metodi Mat.
University of Bergamo, Dalmine, Italy
E-mail: angelo.gargantini@unibg.it

***Abstract*—New coverage criteria for Boolean expressions are regularly introduced with two goals: to detect specific classes of realistic faults and to produce as small as possible test suites. In this paper we investigate whether an approach targeting specific fault classes using several reduction policies can achieve that less test cases are generated than by previously introduced testing criteria. In our approach, the problem of finding fault detecting test cases can be formalized as a logical satisfiability problem, which can be efficiently solved by a SAT algorithm. We compare this approach with respect to the well-known MUMCUT and Minimal-MUMCUT strategies by applying it to a series of case studies commonly used as benchmarks, and show that it can reduce the number of test cases further than Minimal-MUMCUT.**

I. 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 example shown in a study by Chilenski and Miller [1], 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. Using these criteria can lead to reasonably small test suites, but their fault detection capability is reduced with respect to exhaustive testing.

As a consequence of this problem new criteria are introduced and existing criteria are improved, aiming to maximize the fault detection capability while keeping the number of test cases small. These criteria are defined by algorithms 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 thereof, and so the fault detection capability has to be analyzed later. This paper investigates an approach that generates test cases directly targeting specific fault classes and uses several reduction policies to minimize the size of resulting test suites. 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, as we show in this paper. The problem tackled by this approach is how to determine values for the literals of Boolean expressions; if the expression is embedded in source code then determining inputs that drive variables to appropriate values is another problem that is not the scope of this paper.

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, a range of optimizations can be applied to the process: *Monitoring* guarantees that only faults not yet detected are considered for test case generation, *ordering* can reduce the number of faults that need to be considered during test case generation, *collecting* merges several faults into a single test predicate, and *minimization* as a post-processing step removes redundant test cases with regard to the considered fault classes.

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 a SAT solver for test case generation.

The research question we are addressing in this paper is whether the reduction achieved by such an optimized approach can reduce the number of test cases over the well known MUMCUT and Minimal-MUMCUT test generation strategies.

II. 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 [2] or in SCR [3]) can also be seen as Boolean expressions. Note that derivation of input values that drive variables in programs to the desired values is not focus of this work. This section gives all necessary definitions for the remainder of the paper, presents some related work, and introduces fault classes.

A. Definition and notation

For simplicity, we assume that Boolean expressions are given in minimal disjunctive normal form (DNF) which allows comparison with the existing literature, but the described approach can be extended to Boolean expressions in any form. In this paper we follow the notation proposed by Lau

and Yu [4]: Boolean expressions are those involving Boolean operators like AND, OR, and NOT (denoted by $\wedge$, $\vee$, $\neg$ or by "$\cdot$", "+", "$^-$", respectively). The "$\cdot$" is omitted if it is clear from the context. A *literal* is an occurrence of a variable[1] inside a predicate (note that a variable may occur several times in the same predicate). DNF expressions consist of *terms*, i.e., Boolean expressions that are conjunctions consisting only of possibly negated literals, and terms are connected disjunctively. For example, the DNF expression $a\bar{b} + \bar{a}c$, contains two terms $a\bar{b}$ and $\bar{a}c$, and the variable a occurs twice as a positive literal in the first term and as a negative literal (i.e., negated) in the second term. 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.

B. Testing criteria - Related work

There is a body of literature on testing Boolean expressions, including a range of well-known standard criteria such as decision or condition coverage. Several testing criteria have also been developed to target specific common faults in Boolean expressions. Weyuker et al. [7] introduced a family of strategies for automatically generating test cases from Boolean expressions, of which the MAX-A and MAX-B strategies are the most powerful and subsume all others. Chen and Lau [8] introduced three testing criteria: *Multiple Unique True Point (MUTP)*, *Multiple Near False Point (MNFP)*, and *Corresponding Unique True Point and Near False Point (CUTPNFP)*. These three strategies were integrated by Chen et al. [9], [10] in the MUMCUT testing strategy which (1) guarantees to detect nine types of faults in Boolean expressions in irredundant disjunctive normal form just like MAX-A and MAX-B do, and (2) requires only a subset of the test suites that satisfy the previously proposed MAX-A and MAX-B strategies. Kaminski and Ammann [11] introduced an extension of MUMCUT, called Minimal-MUMCUT, which takes the feasibility of the three components of MUMCUT into account and guarantees to detect the same types of faults with fewer test cases. In this paper, we compare all results to those of MUMCUT and Minimal-MUMCUT, because they subsume the other criteria in terms of the detected fault classes and because they have been shown to result in significantly less test cases.

C. Some fault classes

The possible faults in Boolean expressions can be categorized into fault classes; research on this topic has resulted in the definition of several such classes that represent *typical* programmer mistakes and in a hierarchy among them (where possible). The approach presented in this paper does not target any specific fault classes but can be applied to any possible fault classes.

[1]A variable is sometimes called *clause* [5], while a literal is sometimes called *condition* [6].

Table I
EXAMPLE FAULT CLASSES ILLUSTRATED ON EXPRESSION $ab + cd$.

Fault class		Example
TNF	Term negation fault	$\overline{a\bar{b}} + cd$
TOF	Term omission fault	ab
LNF	Literal negation fault	$\bar{a}b + ac$
LOF	Literal omission fault	$a + cd$
LIF	Literal insertion fault	$abc + cd$
LRF	Literal reference fault	$ab + bd$
ENF	Expression negation fault	$\overline{ab + cd}$
ORF[+]	Operator + reference fault	$abcd$
ORF[$\cdot$]	Operator $\cdot$ reference fault	$a + b + cd$

In Table I we list the fault classes targeted by most coverage criteria like MAX-A, MAX-B, MUMCUT, and Minimal-MUMCUT, with which we compare our approach in the experiments. Further fault classes exist, and we refer to the literature [4], [5], [12]–[14] for more information.

Figure 1 relates the fault classes presented in Table I in the fault hierarchy presented by Lau and Yu [4]. An arrow from a fault class F_1 to another class F_2 indicates a *subsumption* relation, i.e., if a test suite is able to detect all the faults in F_1 then it will also detect all the faults in F_2. Such a hierarchy can be useful when generating tests: In theory, if one generates a test suite detecting LIFs and LOFs, then all the other faults in the hierarchy will be detected by the same test suite as well. In practice this is not always the case, as even if the tests of a class F_1 can detect all the faults of another class F_2 this does not guarantee the *existence* of a test case for every fault in F_1 [15]. Sometimes faults of a given fault class do not exist for part of an expression where faults of subsumed fault classes do exist, and sometimes a fault simply does not change the truth values of the predicate.

Example 1 *Consider the Boolean expression $S = ab + a\bar{b}$. The test suite for LIFs of S is empty since there is no literal that can be inserted in any of the terms – both literals a and b are already present in both terms. However, a TOF would cause S to be implemented as either ab or $a\bar{b}$ and the empty test suite for LIF would not discover these faults.*

Kaminski and Ammann [11] 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 (LIF, LRF, and LOF and all the other subsumed classes) with fewer test cases than MUMCUT.

In addition subsumption relations are not always known; sometimes only empirical data about the relationships among fault classes are available (e.g., [7]). Consequently, to guarantee that all fault classes are fully detected the approach we describe in this paper does not depend on a strong subsumption hierarchy, and takes the faults of all considered fault classes into account. As we will show, however, subsumption relations can still be used to improve the efficiency of the process.

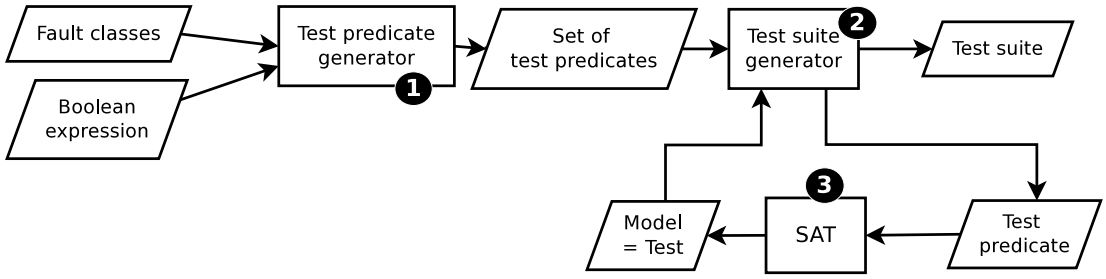

Figure 2. The general test case generation process.

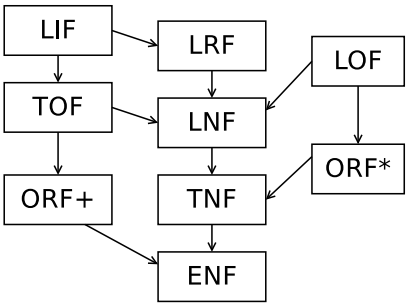

Figure 1. Fault hierarchy [4].

D. 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 [12]*, evaluates to true, i.e., $t \models \varphi \oplus \varphi'$ where $\oplus$ denotes the logical *exclusive or* operator. 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* [16].

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 2 *If the Boolean predicate $a \wedge b$ is implemented as a (a literal omission fault), 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 II-C). $F_C(\varphi)$ 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 C in φ are the expressions $\varphi \oplus \varphi'$ for all φ' in $F_C(\varphi)$.

Example 3 *Consider the expression $a \wedge b$ and let the fault class C be LOF, then $F_{LOF}(a \wedge b) = \{a, b\}$ and the test predicates are the following two expressions: $(a \wedge b) \oplus a$ (which is equivalent to $a \wedge \neg b$) and $(a \wedge b) \oplus b$ (which is $\neg a \wedge b$).*

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 $\mathtt{tp}$ in $\Gamma_C(\varphi)$ there exists a test case t in $\mathcal{T}$ such that t is a model of the test predicate $\mathtt{tp}$ (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 $\forall \mathtt{tp} \in \Gamma_C(\varphi) \exists t \in \mathcal{T}\ t \models \mathtt{tp}$.*

III. Generating Fault Detecting Test Cases

This section presents a general method to derive test cases for specific fault classes. Given a Boolean predicate φ, a fault class C, and a test predicate $\mathtt{tp}$ representing a concrete fault of the predicate belonging to the fault class, 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 ($\mathtt{tp}$).

Finding a model of a Boolean expression, if there exists one, can be efficiently solved by means of several techniques. Previously [6], 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. SAT algorithms can solve the problem of satisfiability of a Boolean expressions efficiently and they are therefore the best choice in this setting.

The entire process of generating a test suite using a SAT 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 $\mathtt{tp}_k = \varphi \oplus \varphi'_k$. 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., $t \models \mathtt{tp}$. 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.

A. 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 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, equivalent faults of Boolean predicates can be detected by the SAT solver, as $\varphi \oplus \varphi'$ is unsatisfiable if φ' is equivalent to φ. If there exist constraints among the literals in φ and therefore the Boolean space is not complete, then φ' may be an equivalent mutant of φ even though $\varphi \oplus \varphi'$ has a model but such model does not satisfy the constraints. In this case, the SAT solver can still find equivalent faults provided that the constraints are modeled as Boolean predicate Φ, by proving that $(\varphi \oplus \varphi') \wedge \Phi$ is unsatisfiable.

Consequently, 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 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.

IV. 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.

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

A test case generated for one test predicate may satisfy a number of further test predicates. 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 simply requires evaluating the test predicate with the model that t represents. If $t \models \mathtt{tp}$ then t also covers tp. This process is usually cheaper than running a SAT solver on each test predicate, even if the number of test predicates is large.

B. 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 [17], 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 [17] showed that it is difficult to find a heuristic that improves over the average random case.
2) *Subsuming order*: If the subsuming 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 [4]. 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, LIF and LOF weakly subsume all the other 6 fault classes presented in Section II-C, so subsuming order would start with test predicates from these fault classes. In this paper we use also the extended subsumption relation presented by Kaminski and Ammann [11] and Chen et al. [15], which takes into account also the feasibility problem (see Section III) of testing criteria and therefore the order will be LIF, LRF, LOF, TOF, LNF, ORF+, ORF*, TNF, and ENF.

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

Instead of generating one test case for each test predicate, one can *collect* many test predicates [18] in a unique conjoint in a way that a model for the conjoint is a model of all the collected test predicates. Per definition, given a test predicate $\mathsf{TP} = \mathtt{tp}_1 \wedge \ldots \wedge \mathtt{tp}_n$, a model t of TP (i.e. $t \models \mathtt{tp}$) is a model for $\mathtt{tp}_1, \ldots, \mathtt{tp}_n$.

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,

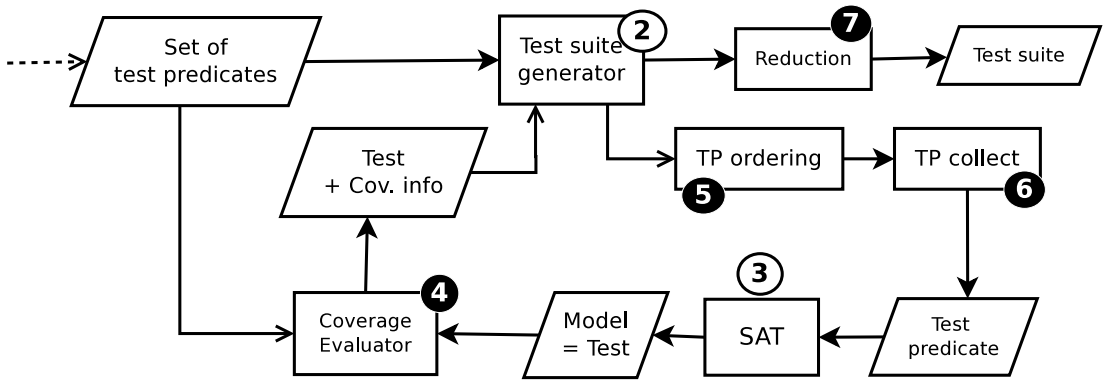

Figure 3. The improved test case generation process.

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 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 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, and the remaining test predicates 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.

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

A test suite is minimal [19] 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. 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 can behave very differently: Minimization requires existing, full test suites while monitoring checks test predicates on the fly during test case generation. On the other hand, monitoring does not guarantee minimal test suites.

Algorithm 1 collection process

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

Note that 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.

V. Experiments

A. A challenge to traditional coverage criteria

The approach explained in the previous section can be used to set up a challenge about the final test suite size to the coverage testing criteria like those presented in Section II-B in the following way: Given a testing criterion which guarantees to detect a set of fault classes $\{c_1, , \ldots, c_n\}$ we can use our method with the test predicates $\Gamma = \cup_{i=1}^{n} \Gamma_{c_i}(\varphi)$, and the final test suite will guarantee the same fault detection but possibly with fewer test cases. This section presents experimental results to witness that this is indeed the case.

B. Experimental setup

For experimentation, we considered the same set of predicates commonly used as benchmarks in several papers on testing DNF expressions: Weyuker et al. selected 13 of the larger transition specifications from a traffic collision avoidance system (TCAS). They also added 7 specifications after

having identified variable dependencies. This set was used to evaluate several testing criteria and generation techniques introduced by Weyuker et al. [7]. Chen, Lau, and Yu evaluated the MUMCUT criterion against the same specifications (except the 12th predicate, as it contains a typo). The same set was used by Kaminski and Ammann [11] to evaluate the Minimal-MUMCUT strategy.

We generated test predicates for all the fault classes presented in Section II-C, as these fault classes are the types of faults that can be detected with MUMCUT [10]. Test suites were generated with and without collecting test predicates, and for the two orderings presented in Section IV.

The reason for comparing with MUMCUT and Minimal-MUMCUT is that these criteria can detect the same nine fault classes presented in Sect. II-C as MUTP, MNFP, CUTPNFP, MAX-A and MAX-B but result in fewer test cases [10], [11]; we used the same fault classes to generate test cases in our experiments.

For test case generation we initially investigated the use of standard SAT implementations. However, we found two main problems: (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 we evaluated have their input Boolean expressions in CNF (Dimacs format) while our test predicates are an exclusive or between two DNF expressions. The conversion of a generic Boolean expression to a CNF formula is itself a research problem, so we preferred to use a more powerful tool that was able to deal with generic Boolean formulas. We decided to use the SMT solver Yices [20], which includes a very efficient SAT solver and claims to be *"competitive as an ordinary SAT and MaxSAT solver*" [20]. 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.

C. Results

Table II summarizes the results of our experiments: Part A of Table II reports the number of variables in the benchmarks specifications (vars), the total number of test predicates (tot.) and how may of those are infeasible (inf.).

The first column (*Orig.*) of part B of Table II lists the mean sizes of the test suites obtained by the MUMCUT strategy as computed by the GUCN method [10]. As the GUCN method achieves better results than the original method [9], we only compare to the improved values. The second column (*Min*) of Part B reports the sizes computed by Minimal-MUMCUT [11].

Part C lists the test suite size obtained by our method using different optimizations: no collect/collect, random order (RND)[2], and by considering the subsumption relation (SUB). The SAT solver was able to find test cases for all feasible test predicates. The *red.* column reports the number of test cases after performing post reduction; ('-' means reduction removed no test cases); For each expression, the minimum test suite size is displayed in boldface.

Finally, Part D of Table II reports the time required to build the final test suite.

[2]The table shows the average over 20 runs with random order

Table III
IMPROVEMENTS OF THE TEST SUITE SIZE

Optimization	Relative to	Resulting size reduction		
		Average	Variance	Max
Subsumption order	Random order	5%	±0.4%	19%
Reduction	No reduction	6%	±0.4%	31%
Collect	No collect	24%	±4%	71%

VI. DISCUSSION

The main result of the experiments is evidence that the approach presented in this paper using the collect/subsumption strategy can cover all faults of the fault classes that test cases generated for MUMCUT and Minimal-MUMCUT would also cover.

A. Comparison among strategies

The optimizations considered in our approach allow a number of different strategies on how to reduce the number of test cases. Table III shows the reduction (average, variation, and maximum of 400 runs) of the size of the final test suite when using different strategies.

Ordering by *subsumption* produces test suites 5% smaller on average compared to test suites generated with random ordering. In one case, without reduction and collecting, the subsumption ordering reduced the test suite size by 19%.

On average, the test suite *reduction* post optimization decreased the number of test cases by 6%, and achieved a maximum of 31%. While optimal minimization of a test suite is an NP-hard problem, this heuristic is computationally quite easy to perform. We conclude that it is worthwhile to always apply it when size is important. Theoretically, reduction could lead to even smaller test suites without monitoring, as results with monitoring depends on the ordering. We did not observe this in our experiments, and as omitting monitoring would mean one would have to create test cases for all test predicates (including subsumed faults) we conclude that *monitoring* should always be used.

The *collection* strategy reduced the number of test cases by 24% on average with a maximum of 71%.

We can conclude that, in terms of the test suite size, the best strategy is to apply monitoring, collection, ordering test predicates using the subsumption strategy, and applying test suite reduction. This strategy produced the smallest test suites in all cases.

> The smallest test suites are generated with monitoring, ordering by subsumption, collecting, and minimizing.

While the results in terms of the reduction in the number of test cases generated clearly suggests the use of the collect

Table II
EXPERIMENTAL DATA

	Part A: Expressions			Part B: #Tests		Part C: #Tests using fault based approach								Part D: Time (secs)			
	Vars.	Test pred.		MUMCUT		NO COLLECT				COLLECT				NO COLLECT		COLLECT	
		tot.	inf.	Orig.	Min	RND (avg)		SUB		RND (avg)		SUB		RND	SUB	RND	SUB
				[10]	[11]	full	red.	full	red.	full	red.	full	red.				
1	7	186	4	39	**27**	34.4	30.4	28	-	30.5	29.2	**27**	-	0.6	0.4	45.6	52.3
2	9	634	9	116	**81**	86.0	83.3	**81**	-	89.7	85.5	**81**	-	2.1	2.6	1821.1	613.4
3	12	2859	67	238.7	157	221.5	187.8	229	188	143.5	139.2	**128**	-	27.2	13.6	20680.4	9448.9
4	5	76	4	11.8	**9**	13.8	12.9	14	13	10.2	10.1	**9**	-	1.4	0.4	43.2	9.2
5	9	550	33	43	36	49.2	43.0	53	43	35.4	35.0	**33**	-	7.3	1.9	823.0	238.4
6	11	360	4	84	66	77.0	65.6	66	**62**	67.7	63.8	**62**	-	7.3	1.5	879.2	242.0
7	10	559	8	106	66	81.0	79.6	72	72	64.3	63.0	**58**	-	7.5	1.8	1860.0	377.8
8	8	109	0	16	**36**	36.9	**36.0**	**36**	-	**36.0**	-	**36**	-	3.2	0.8	261.4	57.3
9	7	46	0	86	**16**	16.2	**16.0**	**16**	-	**16.0**	-	**16**	-	1.4	0.3	34.6	10.4
10	13	602	12	124	**62**	73.5	68.1	**62**	-	69.5	66.6	**62**	-	8.1	1.7	2151.4	446.6
11	13	1094	12	112.4	72	95.0	65.7	79	65	58.3	51.7	**51**	-	62.8	2.4	2816.5	935.6
13	12	425	17	36.1	22	55.7	52.7	58	53	19.9	19.7	**17**	-	6.5	1.5	536.9	128.8
14	7	262	12	34	**22**	31.5	28.5	32	30	24.3	23.6	24	**22**	3.9	0.9	292.1	101.5
15	9	694	41	60.7	39	54.7	50.0	55	50	38.5	37.9	35	**34**	8.1	2.1	1007.5	422.1
16	12	2327	95	153.1	107	159.4	139.6	158	140	94.1	91.9	**86**	-	24.1	8.0	10267.8	4541.8
17	11	542	4	76.3	40	65.0	61.0	66	61	26.7	25.1	27	**21**	6.2	1.5	843.0	249.2
18	10	637	14	78.4	48	73.9	66.5	69	64	40.8	38.4	39	**32**	7.8	1.8	1206.5	431.1
19	8	217	0	44.6	**16**	33.3	28.3	34	29	18.2	17.4	20	**16**	3.1	0.7	208.4	63.6
20	7	68	0	24	**14**	14.8	**14.0**	15	**14**	14.3	**14.0**	**14**	-	1.4	0.3	42.5	9.6
Σ				1184	936	1272.7	1129.0	1223	1107	897.9	864.0	825	805	190.0	44.2	45821.2	18380

strategy, this approach is costly: The time required by using the collection strategy is on average 328 times the time required by the strategy without collection (see Table II). This increase in the time required is caused by the additional calls to the SAT solver (see Figure 1); the SAT solver is called each time a new test predicate is collected.

Collecting test predicates is effective at reducing the number of test cases, but computationally expensive.

Consequently, the preferred strategy is a trade-off between time necessary to generate test cases and time needed to execute the test cases: If resources for test case generation are not critical, then collecting is clearly preferable as it results in smaller test suites. However, if the resources for test case generation are limited, then collecting is less recommendable.

B. Comparison with the MUMCUT strategies

Compared to the original MUMCUT strategy [9] (data not included in Table II to keep it readable), the number of test cases produced with our approach is even smaller when using no optimizations at all: On average our worst result is 4.77 times smaller, while our best result was on average 11.77 times better with a maximum of 99 times for case 13 (17 vs. 1687).

With respect to the more recent MUMCUT experiments using the GUCN method [10] (see Table II), the test suites produced by our approach are smaller for all the specifications and all policies. Our test suite obtained with the best methods was on average 32% smaller than the MUMCUT test suite.

The comparison with respect to the Minimal-MUMCUT strategy [21] is shown in Fig. 4, which reports the percentages of cases we performed better, equally, or worse than Minimal-MUMCUT, depending on the strategy adopted. The best strategy (SUB+COLL+RED) performed equally or better, but never worse than Minimal-MUMCUT.

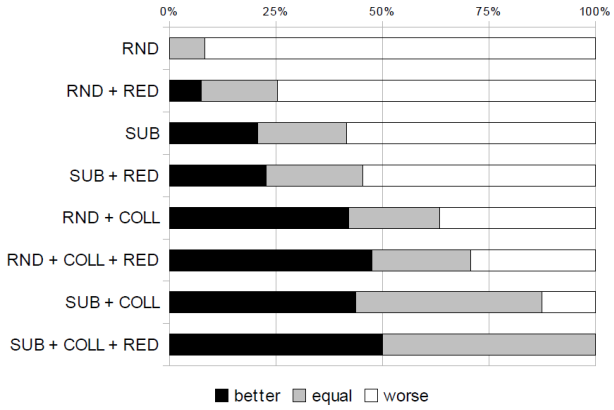

Figure 4. Comparison with Minimal-MUCUT

Creating test cases directly for fault classes using optimizations can reduce the number of test cases necessary to cover all faults of these classes in comparison to MUMCUT and Minimal-MUMCUT.

We cannot compare our approach to MUMCUT and Minimal-MUMCUT in terms of the time necessary to create test suites, as our approach is the first to use a SAT solver to generate the test cases, and previous work on MUMCUT does not include data on the time necessary to automatically generate the test suites.

C. Threats to validity

External threats to validity are given by the choice of predicates for experimentation. Although the comparison to

previous results is valid in that these predicates are a common benchmark for testing Boolean predicates, generalization of our findings is based on the assumption that these are realistic predicates. In particular, all predicates have at least 5 literals, while in practice many predicates will be simpler, in which case the effects on the test suite size will be smaller.

Using a SAT solver for test case generation has the drawback that SAT can be very computationally expensive. It might occur that the SAT solver is timed out even though the test predicate is feasible, thus suggesting the test predicate is infeasible, although of course no definitive conclusions information about whether the test predicate is feasible or not can be drawn. Depending on the implementation, a strategy like MUMCUT might not have this problem for the same predicate. In our experiments, the SAT solver was able to find test cases for all feasible test predicates easily.

Although minimization does not impact the fault detection capability with respect to the considered fault classes, it might still reduce the *residual* fault detection capability. This is a general observation about minimization and not specific to our approach.

Internal threats to validity may result from the ordering of test predicates: Even though we ordered test predicates according to subsumption of fault classes the test predicates *within* the same fault class were chosen randomly, thus potentially influencing results to the worse or for the better. Collecting test predicate is also susceptible to the order in which predicates are considered, in theory. Based on our previous observations [17] we believe that the ordering is unlikely to change the general findings.

The test predicates generated in our approach could theoretically also be used to minimize test suites derived with other criteria (e.g., MUMCUT); as we only compared to published results we have no data on this. However, as the main reduction in our approach is not caused by the minimization but the collecting we believe this would not impact the general findings. We also did not consider the controllability problem, i.e., whether and how a program can be driven to reach a specific valuation for a Boolean expression.

D. Extension to generic Boolean expressions

The work presented in this paper focuses on Boolean predicates given in DNF, because it is possible to prove fault detection capability of coverage criteria when using DNF predicates. However, our approach is not limited to DNF predicates. Therefore, as future work we plan to extend our approach to generic Boolean expressions, not only predicates given in DNF. This would require the definition of suitable fault classes for generic Boolean expressions together with their mutation operators.

There would be several advantages in doing so: First of all, it would remove the need to transform expressions to DNF that are not already in this form. Translating a generic Boolean expression to DNF may require exponential time.

Second, fault-based testing of normalized predicates may miss (few) faults which can be detected if test cases are generated from the original predicates [7], [22]. Chen et al. [23] performed an empirical study to assess the fault detection capability of the MUMCUT strategy with respect to Boolean predicates written in a general form instead of the irredundant disjunctive normal form. They found that 99,5% of the faults guaranteed to be detected by MUMCUT for DNF predicates were also detected by MUMCUT for non DNF predicates. Although this figure is very high, it may not suffice for safety critical software.

Finally, the number of terms and literals after conversion to DNF can be greater than in the original predicate. This means that less test cases are required for the original predicate (as proved also by Kaminski et al. [24]). Consider for example the expression $\varphi = a \vee (b \wedge c)$ containing 3 variables and 3 literals. In DNF, it becomes $\varphi_{DNF} = ab+ac$ which contains 4 literals. The number of possible faults of φ_{DNF} is greater than the number of faults of φ; for example, there are three possible LOFs for φ, while there are 4 possible LOFs forφ_{DNF}. Therefore, normalized expressions may require more test cases than the original expression. A recent study [25] shows that general form (GF)-oriented strategies could enhance the fault detection capability and reduce the sizes of test suites.

VII. Conclusions

In this paper, we have shown that instead of introducing new criteria or improving existing criteria and then investigating their fault detection capability it is also feasible to generate test cases directly targeting certain fault classes. This guarantees the capability to detect all faults of any given fault class while reducing the number of test cases necessary to achieve this in comparison to existing coverage criteria. Experimental evaluation demonstrated that the number of test cases is indeed smaller than for related coverage 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.

References

[1] J. Chilenski and S. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, 1994.

[2] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 684–707, Sep. 1994.

[3] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Transactions of Software Engineering Methodology*, vol. 5, no. 3, pp. 231–261, 1996.

[4] M. F. Lau and Y.-T. Yu, "An extended fault class hierarchy for specification-based testing," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 3, pp. 247–276, 2005.

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

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

[7] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a Boolean specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 353–363, May 1994.

[8] T. Y. Chen and M. F. Lau, "Test case selection strategies based on Boolean specifications." *Software Testing, Verification and Reliability*, vol. 11, no. 3, pp. 165–180, 2001.

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

[10] 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*, vol. 79, no. 6, pp. 820–840, 2006.

[11] 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*. Washington, DC, USA: IEEE Computer Society, Apr. 1–4, 2009, pp. 356–365.

[12] D. R. Kuhn, "Fault classes and error detection capability of specification-based testing," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 4, pp. 411–424, Oct. 1999.

[13] T. Tsuchiya and T. Kikuno, "On fault classes and error detection capability of specification-based testing," *ACM Transactions of Software Engineering Methodology*, vol. 11, no. 1, pp. 58–62, 2002.

[14] K. Kapoor and J. P. Bowen, "Test conditions for fault classes in Boolean specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 3, p. 10, 2007.

[15] Z. Chen, B. Xu, and C. Nie, "A detectability analysis of fault classes for Boolean specifications," in *SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2008, pp. 826–830.

[16] J. Sheldon B. Akers, "On a Theory of Boolean Functions," *Journal of the Society for Industrial and Applied Mathematics*, vol. 7, no. 4, pp. 487–498, 1959.

[17] G. Fraser, A. Gargantini, and F. Wotawa, "On the order of test goals in specification-based testing," *Journal of Logic and Algebraic Programming*, vol. 78, no. 6, pp. 472–490, July 2009.

[18] 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*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 27–42.

[19] 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*, vol. 2, no. 3, pp. 270–285, 1993.

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

[21] G. Kaminski and P. Ammann, "Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing," in *ICST'09: Proceedings of the 2nd International Conference on Software Testing Verification and Validation*. Washington, DC, USA: IEEE Computer Society, Apr. 1–4, 2009, pp. 386–395.

[22] P. E. Black, V. Okun, and Y. Yesha, "Mutation Operators for Specifications," in *ASE'00: Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 2000, p. 81.

[23] T. Y. Chen, M. F. Lau, K. Y. Sim, and C. A. Sun, "On detecting faults for Boolean expressions," *Software Quality Journal*, vol. 17, no. 3, pp. 245–261, 2009.

[24] G. Kaminski, G. Williams, and P. Ammann, "Reconciling perspectives of software logic testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 149–188, 2008.

[25] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 621–622.