How to Optimize the Use of SAT and SMT Solvers for Test Generation of Boolean Expressions

Paolo Arcaini[1], Angelo Gargantini[1] and Elvinia Riccobene[2]

[1]*Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy*
[2]*Dipartimento di Informatica, Università degli Studi di Milano, Italy*
Email: angelo.gargantini@unibg.it

In the context of automatic test generation, the use of propositional satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers is becoming an attractive alternative to traditional algorithmic test generation methods, especially when testing Boolean expressions. The main advantages are the capability to deal with constraints over the inputs, the generation of compact test suites, and the support for fault detecting test generation methods. However, these solvers normally require more time and a greater amount of memory than classical test generation algorithms, making their applicability not always feasible in practice. In this paper we propose several ways to optimize the SAT/SMT-based process of test generation for Boolean expressions and we compare several solving tools and propositional transformation rules. These optimizations promise to make SAT/SMT-based techniques as efficient as standard methods for testing purposes, especially when dealing with Boolean expressions, as proved by our experiments.

Keywords: boolean expression testing; test case generation; SAT solvers; SMT solvers

Received 00 January 2009; revised 00 Month 2009

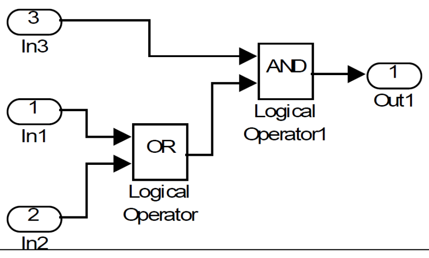
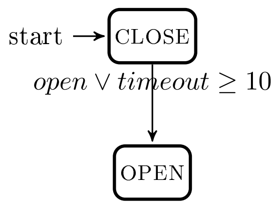
1. INTRODUCTION

Boolean expression testing plays an important role in code and model-based testing, since Boolean expressions can be found in almost all software and system design artifacts. Boolean expressions frequently occur in complex conditions under which some program code is executed or a specification action is performed. They are frequently used to provide semantics to other formalisms (like feature models [1]). Boolean inputs are explicitly found in models of digital logic circuits, like the Simulink model *a* in Fig. 1, which is taken from [2]. In these cases, the extraction of Boolean expressions is rather straightforward. More often, Boolean inputs derive from abstraction techniques that consist in replacing complex formulae with Boolean predicates. These techniques can be applied to high level specifications – see example *b* in Fig. 1 – in which complex conditions about the state are replaced with atomic predicates. Similar abstraction techniques can be applied to source code, for instance in order to obtain Boolean programs [3]. The example *c* in Fig. 1 reports a Boolean expression obtained from a conditional statement in a Java program. Note that in this case there exists a constraint among the inputs (since `i` $<$ `0` and `i` $>$ `100` cannot be both true).

It is widely known [4, 5] that a Boolean expression may be affected by certain types of errors that can occur in it and which are known as fault classes of the expression. Exhaustive testing of Boolean conditions is not feasible in practice, since a Boolean expression f with n variables requires 2^n test cases and this n may be big. Therefore, testing criteria (like condition coverage, decision coverage, MCDC [6], and MUMCUT [7]) are usually applied to select only subsets of all possible test cases having, obviously, a reduced fault detection capability. These traditional approaches build a test suite from the syntactical structure of the Boolean expression, but they do not explicitly consider the expression fault classes. They may also require expressions to be in a particular normal (usually disjunctive) form, and they have difficulties with *coupled* conditions [6] (i.e., conditions influencing the value of other conditions in the same expression) and constraints over the inputs.

To overcome limitations of traditional algorithmic testing generation methods for Boolean expressions, recent results [8, 9] show how it is possible to reduce the problem of finding fault detecting test cases to a logical satisfiability problem, which can be solved by a SAT-based algorithm. However, the effort required by SAT solvers in terms of time and memory, questions their practical usability in test generation.

	a) A Simulink model	*b)* High level specification	*c)* A small fragment of a Java program

	 3
	In3
	1
	In1
	2
	In2
	OR
	Logical Operator
	AND
	Logical Operator1
	1
	Out1
			`int i;`
			`...`
			`if (i < 0 || i > 100) {`
			`    ...`
			`}`

ABS		α : *state* = CLOSE β : *open* γ : *timeout* ≥ 10	a : `i` < 0 b : `i` > 100
BE	$out_1 = (in_1 \vee in_2) \wedge in_3$	$\alpha \wedge (\beta \vee \gamma)$	$a \vee b$ with constraint $\neg(a \wedge b)$

FIGURE 1: Examples of specifications, abstractions (ABS row), and Boolean expressions (BE row)

Surely, SAT solvers are being increasingly used for solving practical problems where one needs to satisfy several potentially conflicting constraints, and satisfiability solvers can now be effectively deployed in practical applications [10]. But also SMT solvers are increasingly used in applications, and can have further potentialities [11]. Although they are far more complex tools than SAT solvers, they should be as powerful as SAT solvers when applied to satisfiability problems, with a minimum overhead. Besides applying some pre-transformations of the predicates that permits them to accept as input generic Boolean formula, SMT solvers have a richer command interface, allowing, for instance, the addition and the retraction of assertions. All these features make them more flexible tools.

Although SAT and SMT solvers are already successfully employed in several projects of software testing and verification (for instance, at Microsoft with SAGE [12] and Pex [13], or in the automotive domain [14]), in some areas, like testing of Boolean specifications, they are rarely used. For Boolean expressions, classical testing criteria are widely used together with simple yet fast algorithms for test generation. What are exactly the advantages SAT/SMT-solvers can bring to test generation for Boolean expressions?

The SAT/SMT-based test generation technique proposed in [8, 9] can explicitly deal with complex constraints over the inputs and coupled conditions [15]. It also produces compact test suites without loss of fault detection capability of the generated tests. For instance, in [9], we have proved that SAT-based generation is able to produce test suites smaller of around 75% than those produced by classical Minimal-MUMCUT and smaller of around 17% than those produced by MCDC. Having compact test suites with an assured fault detection capability is of extreme importance, especially in testing safety critical systems, when both the cost of executing every single test and of missing a fault are very high. Moreover, this approach does not require the specifications under test to be expressed in a particular normal form, so avoiding both overhead due to the formula transformation and missing faults due to the transformation to normal form [16]. Finally, it generates test cases directly targeting specific fault classes.

However, the use of SAT/SMT solvers for test generation purposes requires more time and memory than standard generation algorithms and this fact limits its use in practice, unless numerous optimizations, as initially sketched in [17], are devised.

In this paper, we formalize an optimized SAT/SMT-based process, and explains in details and with preciseness all its sub-procedures, especially the collecting algorithm that is responsible for producing compact test suites. Moreover, we deal with constraints over the inputs of the Boolean expressions, and we consider, among the testing criteria, also MCDC. A broad range of options and optimizations are presented. Some regard the actual use of the tools (e.g., avoiding the exchange of files with the external solver, and using native libraries instead). Other optimizations improve the SAT/SMT-based process of automatic test generation, independently from a specific input specification and selected testing criterion. Others are specific to the process for testing Boolean expressions. Some optimizations take advantage of the interface provided by the solvers.

We also propose a comparison of different off-the-shelf SAT and SMT solvers that can be used in the test generation process and that are able to support (not necessarily all) the proposed optimizations. On the base of our experiments, a good SAT solver performs better than SMT solvers when no optimization is used, although SAT solvers pay a price for accepting only CNF (like SAT4J [18]) or for having only a command line version (like NFLSAT [19]). On the other hand, SMT solvers overcome SAT solver limitations by accepting any form of formula and by offering a richer command interface that makes possible the application of all the optimizations. This is why, although we do not exploit any external theory supported by SMT solving, we still consider SMT solvers in our work, and, on the base of the best SMT tool, we show evidence that the proposed optimizations are effective.

Overall, we show that SAT/SMT solvers can be successfully applied to Boolean testing and that a well engineered process for test generation based on SMT solvers is capable of generating tests for complex criteria (fault-based and MCDC) in a reasonable time.

The paper is organized as follows. Background on test generation for Boolean expressions is given in Sect. 2 and the general process of test generation by using SAT/SMT solvers is described in Sect. 3. Process options are presented in Sect. 4, while optimizations of the test generation process are presented in Sect. 5. Experimental results which bring evidence of the improvements due to the proposed options and optimizations are reported in Sect. 6 where several experiments, conducted by means of different SAT/SMT solvers and on diverse Boolean specifications, are presented. Sect. 7 discusses possible threats to the validity of our approach. Sect. 8 presents some related work, while Sect. 9 concludes the paper.

2. TEST GENERATION FOR BOOLEAN EXPRESSIONS

In this section we introduce some basic definitions regarding model-based test generation in case specifications under test are Boolean expressions. Note, however, that our approach may be extended to other types of specifications.

A Boolean expression contains as operands several Boolean subexpressions, including *atomic* Boolean terms, that cannot be further decomposed in simpler Boolean expressions. We call the atomic terms *inputs* or *variables* and use symbols like $x_1, x_2, \ldots$. The occurrence of an input in a expression 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) and three conditions (two x_1's and one x_2). If the expression is not normalized, i.e., no restrictions exist on how operators and conditions are joined together, we say that the expression is in *general form* (GF). In this paper we consider GF Boolean expressions.

A Boolean expression under test φ comes with constraints that can reduce the input space of φ. The set of constraints can be represented by a predicate δ over the variables of φ. Therefore, because of the constraints δ, the input space of φ is not *complete*, i.e., inputs can take only logical values that satisfy δ. Furthermore, we assume that the constraints do not contradict one with another and therefore there exists at least a model that satisfies δ.

We provide the following definitions for the testing of Boolean predicates.

Definition 2.1. *Given a boolean expression φ with constraints δ, a* test case *is a value assignment to every Boolean variable in φ that satisfies δ. A* test suite *is a set of test cases; the* size *of a test suite is the number of its tests.*

Test generation is usually driven by some testing criteria, which can be formalized as follows.

Definition 2.2. *A* testing criterion TC *is a function that, given a Boolean expression φ, returns a set of predicates which must be satisfied (or covered). Each predicate represents a test goal and is called* test predicate *(*tp*).*

Definition 2.3. *Given a Boolean expression φ and its constraints δ, we say that a test* t *satisfies or* covers *a test predicate* tp *of φ iff* t *is a model of* tp *and satisfies δ, i.e.,* $\mathtt{t} \models \delta \wedge \mathtt{tp}$. *A test predicate* tp *is said* infeasible[3] *if there is no test that satisfies it and the constraints, i.e.,* $\models \neg(\delta \wedge \mathtt{tp})$ *or, equivalently,* $\models \delta \rightarrow \neg\mathtt{tp}$.

Definition 2.4. *Given a testing criterion TC, a Boolean expression φ and its constraints δ, we say that a test suite TS is* adequate *to test φ according to TC, iff every test predicate of TC either it is unfeasible or it is feasible and covered, i.e.,* $\forall \mathtt{tp} \in TC(\varphi)\colon \models \neg(\delta \wedge \mathtt{tp}) \vee (\exists t \in TS\colon t \models \delta \wedge \mathtt{tp})$.

Note that including the constraints in the test generation process is important, and it has been discussed in [15].

Example 1 (Condition Coverage of example c of Fig. 1). Consider, for instance, as testing criterion TC the *condition coverage*, which requires that each atomic condition in the expression is evaluated at least once to true and at least once to false [20]. The set of test predicate for TC is simply the set of all the inputs of the expression under test in positive and negative form. Let the Boolean expression $\varphi = a \vee b$ with the constraint $\delta = \neg(a \wedge b)$ be the expression under test. The test predicates for the coverage of all the conditions in φ are $\{a, \neg a, b, \neg b\}$. The test suite containing the two tests $\{(a = \top, b = \bot), (a = \bot, b = \top)\}$ covers all the test predicates, satisfies the constraint δ, hence it is adequate to test φ according to the condition coverage criterion.

2.1. Two Selected Testing Criteria

There exist many testing criteria proposed in literature for Boolean expressions. Kaminski et al. [21] counted 16 testing criteria, that can be grouped in two families: semantic and syntactic criteria. Semantic criteria like condition coverage, decision coverage, and the Modified Condition Decision Coverage (MCDC) Criterion [6] focus on the meaning of the Boolean expressions and do not require the expression to be in a particular form. Syntactic criteria (like MUMCUT [7]) assume that the expressions are in a standard form (generally DNF or CNF) and define the tests and their fault detection capability by considering the structure of the expression

[3]For it represents an infeasible test requirement [20].

under test. New criteria have been since introduced; for instance, new fault-based criteria directly related to Boolean fault classes are defined in [9]. In this paper, we consider two criteria, the first one is a fault-based criterion and the second one is the MCDC.

Fault-based Testing Criteria In [9] we have presented fault based testing criteria which generate test predicates from Boolean expressions in disjunctive normal form and whose tests are guaranteed to detect faults in specific fault classes (as in [22, 23]). This approach has been extended to GF Boolean expressions and it intends to go beyond classical syntactic criteria like MUMCUT. Although its detailed description is outside the scope of this paper, it can be summarized as follows.

Given a Boolean specification φ, a fault class F identifies all the possible faulty versions φ'_i of φ, due to a particular kind of fault. Several fault classes for Boolean expressions have been defined and a hierarchy among them has been established [4, 24]. According to the fault-based testing criteria approach, given an expression φ and ranging over all the fault classes, the testing criteria compute all the *test predicates* $\mathsf{tp}_i = \varphi \oplus \varphi'_i$ (called *detection conditions*), where $\oplus$ denotes the exclusive or (xor). A model of tp_i makes φ true and φ'_i false, or the other way around.

Example 2 (Tps for the Stack-at-0 fault). The Stack-at-0 fault (SA0) replaces one occurrence of a condition with false [24]. Let $\varphi = a \vee b$ be the Boolean expression of example c of Fig. 1. The test predicates obtained from φ by applying the SA0 are $\mathsf{TPS} = \{(a \vee b) \oplus (\mathsf{false} \vee b), (a \vee b) \oplus (a \vee \mathsf{false})\}$.

MCDC testing criterion The MCDC criterion [6] is one of the most widely used coverage criteria for software testing for Boolean expressions, largely because MCDC is required by the Federal Aviation Agency (FAA) for all software on moderate-size U.S. commercial aircraft. Briefly, it requires that every variable in the expression has been shown to independently affect the final value of the expression. The original MCDC definition has been interpreted in different ways, originating several methods to generate test predicates. In this paper, we use the following two flavors of MCDC.

1. The first one adopts a weak form of MCDC, called General Active Clause Coverage (GACC) [25, 21] and the Boolean derivative to generate test predicates [26]. Given a Boolean variable v in an expression P, let $P_{v \leftarrow \mathsf{true}}$ represent the expression P with every occurrence of v replaced by true, and $P_{v \leftarrow \mathsf{false}}$ the expression P with every occurrence of v replaced by false. The condition under which the value of v determines the value of P is $C_v \equiv P_{v \leftarrow \mathsf{true}} \oplus P_{v \leftarrow \mathsf{false}}$. Then, for each variable v in P, the two test predicates are $v \wedge C_v$ and $\neg v \wedge C_v$.
2. The second one uses the masking variation of the MCDC, also known as Correlated Active Clause Coverage (CACC) [25, 21]. The generation of test predicates is based on the visit of the expression P under test by considering it as a tree [27]. Starting from the leaf representing the condition c under test, the tree is visited up to the root. Every sibling node is marked false if the parent node is an $\vee$, while it is marked true if the parent node is an $\wedge$. If the node is the operator $\neg$, the marking is switched (false for $\wedge$ and true for $\vee$). Once the root is reached, the tree is traversed again in the inverse direction and the test predicate is built considering the marking given to each sibling node. The condition c is then added as c and as $\neg c$ in order to obtain two test predicates. The complete generation algorithm for all the variables in an expression is shown in Alg. 1. Note that the CACC subsumes other testing criteria as decision and condition coverage [20].

Example 3 (Tps for MCDC). Let $\varphi = a \vee b$ be the Boolean expression of example c of Fig. 1 and let a be the variable under test. With the first method, $C_a \equiv \varphi_{a \leftarrow \mathsf{true}} \oplus \varphi_{a \leftarrow \mathsf{false}} \equiv (\mathsf{true} \vee b) \oplus (\mathsf{false} \vee b) \equiv \neg b$ and the two test predicates are $\mathsf{TPS} = \{a \wedge C_a, \neg a \wedge C_a\}$. Instead, the tree method visits the expression tree, it marks the node b to false, and computes the test predicates $\mathsf{TPS} = \{a \wedge \neg b, \neg a \wedge \neg b\}$.

3. TEST GENERATION BY SAT/SMT SOLVERS

In this section, we explain and formalize an approach for test generation by means of SAT and SMT solvers extending the technique originally presented in [9, 17].

Upon the assumption of having an algorithmic way for generating the test predicates, given a specification (i.e., a Boolean expression φ and a set of constraints δ over the variables of φ) and a testing criterion, a SAT/SMT solver can be used in order to find a *test* that covers a test predicate or to discover if it is infeasible (assuming that the solver terminates, otherwise it is not known whether the test predicate is infeasible or not).

The proposed overall test generation process by SAT/SMT solvers is depicted in Fig. 2 and shown in Alg. 2. Given some testing criteria, a set TPS of test predicates is generated from a specification (① in Fig. 2 and line 1 in Alg. 2) according to Def. 2.2.

The generation of the complete test suite (② in Fig. 2) can be performed by taking a test predicate from TPS and trying to generate a test that covers it (according to Def. 2.3). In order to obtain an adequate test suite (according to Def. 2.4), the process should be iterated until one test is generated for every feasible test predicate. However, this naïve approach requires a high amount of time and generates huge test suites. It can be improved by several techniques: here we

Algorithm 1 Computing MCDC test predicates for CACC

```
function mMcdcTP(BooleanExpression $a$)
    if $a = a_1 \wedge a_2$ then
        return Prod( mMcdcTP($a_1$), $a_2$) $\cup$ Prod( mMcdcTP($a_2$), $a_1$)
    else if $a = a_1 \vee a_2$ then
        return Prod( mMcdcTP($a_1$), $\neg a_2$) $\cup$ Prod( mMcdcTP($a_2$), $\neg a_1$)
    else if $a = \neg a_1$ then
        return mMcdcTP($a_1$)
    else                          // a is a variable
        return { $a$, $\neg a$ }
    end if
end function

function Prod(SetOf[LogicalExpression] $\Sigma$, LogicalExpression $x$)$\rightarrow$SetOf[LogicalExpression]
    result $\leftarrow$ {}
    for all $\sigma \in \Sigma$ do
        result := result $\cup$ {$\sigma \wedge x$}
    end for
    return result
end function
```

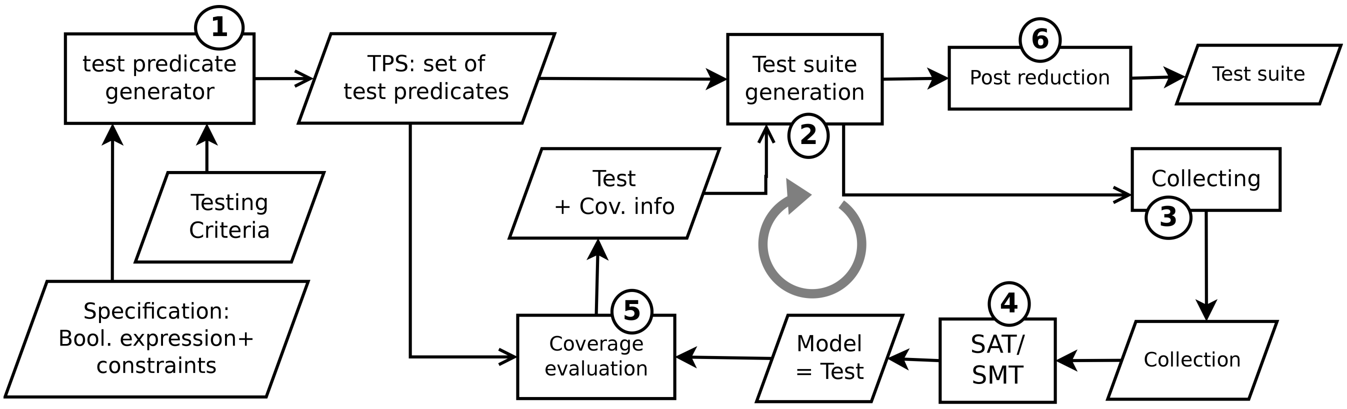


FIGURE 2: Test generation process

consider collecting (③ in Fig. 2), coverage evaluation (⑤ in Fig. 2), and post reduction (⑥ in Fig. 2)[4].

During the test generation process, the test predicates in set TPS are moved to three other sets: COLLECTED contains all the test predicates that have been already collected and which a test has been generated for, INFEASIBLE contains all the test predicates that have been found infeasible, and TBC_FEASIBLE contains all the test predicates that have been tested for feasibility and, although they could not be collected yet, they are feasible (this set is useful in order to avoid to repeatedly check the feasibility of the test predicates). The resulting test suite generator cycle (② in Fig. 2) corresponds to the **while** loop in Alg. 2. The loop continues until every test predicate has been collected or has been found infeasible; in each iteration, the algorithm collects several test predicates together (line 7), finds a test t that covers all the collected test predicates (line 8), computes the coverage achieved by t (line 9), and adds t to the test suite (line 10).

In the following, sets TPS, COLLECTED, INFEASIBLE, and TBC_FEASIBLE must be considered as global variables.

[4] *Ordering* of test predicates is not considered in this work and we assume random ordering.

3.1. Collecting

A core optimization employed in this paper consists in finding tests that cover as many test predicates as possible, instead of single test predicates. We call *collection* the set of test predicates sharing the same model, and *collecting* the process of grouping test predicates. The collecting phase is shown in Fig. 2 as ③ and it is performed in line 7 of Alg. 2.

From a theoretical point of view, the problem of collecting consists in partitioning the set of feasible test predicates with the minimal number of partition classes. Each class contains only test predicates sharing the same model. The number of possible partitions of a

Algorithm 2 Test suite generation

```
Input: $\varphi$: Boolean expression
Input: $\delta$: Constraints
Input: $CRITS$: Testing criteria
 1: TPS $\leftarrow$ getAllTPs($\varphi$, $CRITS$) // all the test predicates to cover
 2: TBC_FEASIBLE $\leftarrow \{\}$      // test predicates to be covered and known to be feasible
 3: COLLECTED $\leftarrow \{\}$         // collected test predicates
 4: INFEASIBLE $\leftarrow \{\}$        // infeasible test predicates
 5: $TS \leftarrow \{\}$                // test suite
 6: while TPS $\cup$ TBC_FEASIBLE $\neq \emptyset$ do
 7:     $C \leftarrow$ COLLECT          // collecting
 8:     $t \leftarrow$ modelSat($\delta \wedge \bigwedge_{c \in C} c$) // test computation
 9:     coverageEval($t$, $C$) // coverage evaluation
10:     $TS \leftarrow TS \cup \{t\}$
11: end while
12: postReduction
13: return $TS$
```

set of size n is given by the Bell number B_n which grows exponentially with n [28]. For this reason, in order to keep the search of the optimal solution still feasible in practice, we accomplish such partition by using the greedy algorithm reported in Alg. 3 and explained in the following.

The algorithm starts with an empty collection C (line 1). Then, it tries to add every test predicate tp to C. This is possible only if tp does not invalidate the collection, i.e. it shares at least one model with the other already collected test predicates (line 3). This is checked by calling an auxiliary function canCollect(tp, C). In the basic version of the algorithm, canCollect is simply defined as **return** sat($\delta \wedge \mathsf{tp} \wedge \bigwedge_{c \in C} c$), i.e., it calls a SAT/SMT solver to check if the conjunction of the constraint δ, the test predicate tp, and all the test predicates already in C is satisfiable. In Sect. 5, we introduce several optimizations for checking whether a test predicate can be inserted in the collection C, i.e., we provide advanced implementations of canCollect.

If the test predicate is not collected and it is not known for being satisfiable (line 6) (i.e., it belongs to TPS and not to TBC_FEASIBLE), the SAT/SMT solver is called for assessing the test predicate satisfiability (line 7). According to the result, the test predicate is moved either to TBC_FEASIBLE (in this way the checking for satisfiability in other runs of the algorithm is avoided), or to INFEASIBLE (in this way, it is no more considered as a collectable test predicate). Note that recording the satisfiability of a test predicate is very important, because infeasible test predicates consume computing resources without producing usable tests.

The collecting process is able to produce very compact test suites [9], but it is very expensive in terms of solver calls. For this reason, the algorithm may decide by calling the function quitCollecting to quit collecting, even before all the remaining (not collected yet) tps are considered.

In Fig. 3 the evolution of the classification of a set of test predicates is shown. At the beginning, all the test predicates have to be evaluated. After the first run of the collecting algorithm, some of the infeasible test predicates have been identified, some test predicates have been collected in the collection C_1, and some other test predicates have been moved to TBC_FEASIBLE. In the following runs of the collecting algorithm, the test predicates in TPS and in TBC_FEASIBLE are all gradually collected in the collections $C_2, \ldots, C_n$ (or found infeasible).

In the basic version of the algorithm, quitCollecting is simply defined as **return false**, namely we consider all the test predicates. In this case, during the first run all the test predicates in TPS are considered for collection and, therefore, TPS is emptied. Sect. 5.3 will introduce several means of efficiently limiting collecting.

Example 4 (Collecting tps of Example 1). Let's collect with Alg. 3 the test predicates TPS $= \{a, b, \neg a, \neg b\}$ of Example 1, considering them in the order in which they are reported here. The algorithm collects them in two consecutive runs. In the first run, it builds the collection $C_1 = \{a, \neg b\}$, and it puts b and $\neg a$ in TBC_FEASIBLE; note that b has not been collected because of the constraint $\delta = \neg(a \wedge b)$. In the second run, the algorithm collects in C_2 both b and $\neg a$.

3.2. Test computation

Once a collection C_i is built, the SAT/SMT solver is invoked (④ in Fig. 2 and line 8 of Alg. 2) to find a test that covers all the test predicates collected in C_i by searching a model for $\delta \wedge \bigwedge_{c \in C_i} c$.

3.3. Coverage evaluation

After the computation of a collection model, the *coverage evaluation* is performed (⑤ in Fig. 2 and

Algorithm 3 COLLECT: Collecting process

```
Output: $C$ : collection of test predicates
 1: $C \leftarrow \{\}$
 2: for tp $\in$ TPS $\cup$ TBC_FEASIBLE do
 3:     if canCollect(tp,C) then
 4:         $C \leftarrow C \cup \{$tp$\}$        // tp is added to the collection $C$
 5:         moveTo(tp,COLLECTED)
 6:     else if tp $\in$ TPS then // tp is not collected and not known for being satisfiable
 7:         if sat($\delta \wedge$ tp) then
 8:             moveTo(tp,TBC_FEASIBLE)// tp is feasible but it cannot be collected
 9:         else
10:             moveTo(tp,INFEASIBLE)
11:         end if
12:     end if
13:     if quitCollecting($C$) then
14:         break
15:     end if
16: end for
17: return $C$

  procedure moveTo(TestPredicate tp, SetOf[TestPredicate] DEST)
      ORIG $\leftarrow$ getSet(tp)     // tp belongs to only one set (TPS, COLLECTED, INFEASIBLE, or TBC_FEASIBLE)
      DEST $\leftarrow$ DEST $\cup \{$tp$\}$
      ORIG $\leftarrow$ ORIG $\setminus \{$tp$\}$
  end procedure
```

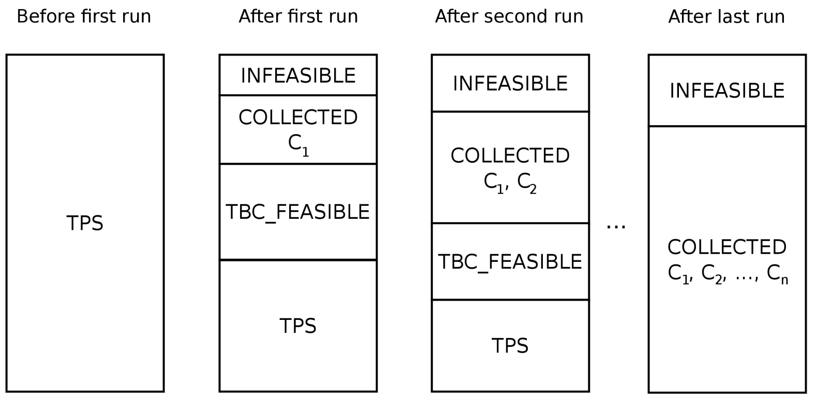


FIGURE 3: Evolution of the test predicate collection process

line 9 of Alg. 2); it checks if the newly generated test also covers other test predicates (excluding those just collected and those infeasible). Alg. 4 shows the coverage evaluation in details; if a test predicate tp is covered by the test t, it is added to C and considered collected (i.e., if necessary, moved to COLLECTED) in order to skip it for further considerations. Because of coverage evaluation, collections $C_1, \dots, C_n$ do not constitute a partition of COLLECTED, since a test predicate could belong to more than one collection. Moreover, coverage evaluation is also needed for post reduction (see Sect. 3.4).

Remark on coverage evaluation vs collecting
Both coverage evaluation and collecting contribute to produce compact test suites. Using collecting reduces the number of the generated tests more than coverage evaluation [9], but, while coverage evaluation is not expensive (checking if a test is a model for a formula is linear with the size of the formula) and no solver is called, collecting is very computationally expensive and requires several calls of the SAT/SMT solver in the algorithm COLLECT. However, while in coverage evaluation a test generated for a test predicate is evaluated *a posteriori* w.r.t. the test generation to check if it incidentally covers also other test predicates, collecting aims *a priori* at generating a test which covers many uncovered test predicates.

Algorithm 4 coverageEval: Coverage evaluation

```
Input: C: a collection of test predicates
Input: t: a test that covers the collection C
  for tp ∈ (TPS ∪ TBC_FEASIBLE ∪ COLLECTED \ C) do
      if isModelFor(t, tp) then
          C ← C ∪ {tp}
          if tp ∉ COLLECTED then
              moveTo(tp, COLLECTED)
          end if
      end if
  end for
```

3.4. Post reduction

A test predicate could be covered by several tests, and this is discovered by coverage evaluation. So, after the test generation process, the resulting test suite may contain tests which are *unnecessary*, i.e., removing them from the test suite leads to a situation in which test predicates are all still covered (as in [29]). The problem of finding the optimal subset of the original test suite that still covers all the test predicates is NP-hard, but can be efficiently solved by a simple greedy heuristic [30]. This task is accomplished by *Post reduction* (⑥ in Fig. 2 and line 12 in Alg. 2), the last step of the process; it removes unnecessary tests (if any) and it can be performed in a negligible amount of time.

4. PROCESS BASIC OPTIONS

The test generation process requires the user to select some *options* that may optimize the performances of the process. Referring to the process in Fig. 2, some options regard the structure of the test predicates generated at step ① (in Sect. 4.1), another concerns the way the solvers are invoked at step ④ (in Sect. 4.2), while others regard the form in which the specification is given (in Sect. 4.3).

4.1. Test Predicate Generation Options

O.MCDC
As explained in Sect. 2.1, there are two ways to generate the test predicates for MCDC, and they differ in the structure (and therefore complexity) of the test predicates. The two possible options are using either GACC or CACC.

O.X *Simplification of the test predicates*
Considering that most of our test predicates (those produced from fault classes and from MCDC using the Boolean derivative) have form $\varphi \oplus \varphi'$ and that φ and φ' often have a common subexpression, it may be useful to apply some kind of simplification of a test predicate before running the solver in order to reduce the number of conditions (i.e., occurrences of literals). We have used the following two equivalences that allow to factor a part of the formula and to push the $\oplus$ operator near the literals:

$$(a \wedge b) \oplus (a \wedge c) \equiv a \wedge (b \oplus c)$$
$$(a \vee b) \oplus (a \vee c) \equiv \neg a \wedge (b \oplus c)$$

where a, b, and c are predicates.

Example 5 (Simplification of xor expressions). Consider, for instance, the expression $\varphi = a \vee b$ and apply the SA0 to b, obtaining $\varphi' = a \vee \mathsf{false}$. The test predicate $\varphi \oplus \varphi'$ becomes:

$$(a \vee b) \oplus (a \vee \mathsf{false}) \equiv \neg a \wedge (b \oplus \mathsf{false}) \equiv \neg a \wedge b$$

While the original test predicate has 3 conditions, the simplified version contains only two condition.

4.2. Solver Invocation Option

O.API *Using the API and avoiding the exchange of files*
A simple optimization regards the way the solvers are invoked. The previous version of our prototype tool runs solvers by command line interface. Each invocation of the solver is done by creating a new external process, and the interaction with it is performed by means of files and command line strings. That solution requires that the solver is fed with input files; in this way, the read/write speeds of the hard disk can increase the time taken just to invoke the tool. Since in the collecting process (see Sect. 3.1) the SAT/SMT solver is repetitively called, the number of invocations of the solver rapidly increases and the time taken just to invoke the tool becomes a critical factor. A simple yet critical optimization consists in avoiding this use and embedding the solver in the process itself.

4.3. Specification form

O.CNF *Choosing the best transformation to CNF*
Almost all SAT solvers require CNF input formulae, while Boolean expressions we consider and their test predicates have *general* form. Efficient transformation to CNF is still a research topic [31]. There are at least two classical possible alternatives: one that preserves equivalence and consists in applying several logical equivalences (double negative law, De Morgan's laws, distributive law), and the classical transformation proposed by Tseitin [32] that preserves satisfiability, avoids the size explosion of the resulting CNF, but introduces a linear number of new variables.

O.GF *Avoiding the transformation to clausal form*
As argued by Jain and Clarke [19], converting a non-clausal formula to CNF requires a great effort (it can grow exponentially in length) and it may destroy the initial structure of the formula, which could be used for efficient satisfiability checking. Therefore, SAT/SMT solvers taking Boolean expressions in general form (GF) may perform better.

Algorithm 5 canCollect – Checking if the witness is a model

```
function canCollect(tp, $C$)
    if isModelFor($C$.witness, tp) then
        return true
    else if sat($\delta \wedge$ tp $\wedge \bigwedge_{c \in C} c$) then
        $C$.witness ← modelSat($\delta \wedge$ tp $\wedge \bigwedge_{c \in C} c$)
        return true
    end if
    return false
end function
```

5. COLLECTING OPTIMIZATIONS

Since collecting (step ③ in Fig. 2) mostly contributes in generating small test suites, but it is also the most expensive [9], a great effort should be spent to improve this part of the generation process. In this section, we devise several techniques to speed up the complete collecting process (see Sect. 5.1 and Sect. 5.2) and how to quit the collecting in advance (see Sect. 5.3).

5.1. Witness exploitation

O.C_ W *Collection with witness*
The collecting algorithm (Alg. 3) returns the collected test predicates and the SAT/SMT solver is called after the collecting process to find a model for the conjunction of the collected test predicates (④ in Fig. 2 and line 8 in Alg. 2). Indeed, the collecting algorithm simply checks that the collection is satisfiable, but it does not ask for a model.

An optimization could be avoiding a further call to the SAT/SMT solver: The collecting algorithm should always ask to the solver also the model of the collection when this is satisfiable. Alg. 3 can be modified in a way that it returns the collected test predicates together with the model found for the collection. This model is a *witness*, since it is the proof that the collected test predicates can be actually collected together.

This optimization assumes that the computational overhead introduced in the collecting algorithm by the multiple model requests is negligible with respect to a further call of the SAT/SMT solver over the collection.

O.C_ WM *Checking if the witness is a model*
When trying to add the current test predicate tp to the collection C (line 3 in Alg. 3), one could check if the witness for the collection C is already a model for tp. In this case, tp can be added to C without any further call of the SMT solver. Note that this optimization requires that O.C_W is applied. Alg. 5 shows the modified canCollect function.

5.2. Incremental collecting

O.C_I *Collecting incrementally*
Most modern SAT/SMT solvers maintain the logical context of a given problem and allow incremental satisfiability checking. Although this concept is not uniquely defined, it means that it is possible to incrementally add expressions to the current context[5] and that satisfiability is checked after each addition. If the expression is a conjunction $e_1 \wedge \ldots \wedge e_n$, the sat predicate can be computed in the following way:

```
function sat($e_1 \wedge \ldots \wedge e_n$)
    $ctx$ ← makeNewContext
    for $i = 1, \ldots, n$ do
        $ctx$.add($e_i$)
        if $\neg ctx$.solve() then
            return false
        end if
    end for
    return true
end function
```

In SAT solvers that require the formulae to be in CNF, each e_i must be a clause. In SMT solvers, instead, each e_i can be any Boolean expression in general form.

O.C_IB *Collecting incrementally with backtracking*
Besides the incremental satisfiability, most SMT solvers, like Yices [34] and Z3 [35], have the further feature of removing an added formula from the context. Therefore, SMT solvers allow incremental collecting (O.C_I), but also backtracking of assertions.

Normally, to prove that a collection of test predicates has a model, in the canCollect function (line 3 in Alg. 3), we build a new context and we search for a model of the formula $\delta \wedge$ tp $\wedge \bigwedge_{c \in C} c$. Instead, thanks to the capability of adding and removing assertions to the context, we could incrementally collect all the test predicates having a common model. Alg. 6 exploits the SMT operations push, saving the current context, and pop, restoring the previous saved context. At the beginning of every collecting process, the context contains only the constraint δ. Before adding a single candidate test predicate tp to the context by an assert instruction, the context is saved by a push. If the context has still a model (solve returns true), then tp can be collected (the function returns true), otherwise the context is restored by a pop.

O.C_DI *Double incremental collecting*
In case all the test predicates have the form $\varphi \oplus \varphi'_i$ (if the $\oplus$ is not pushed), one can use the following *Xor elimination* logical equivalence:

$$\delta \wedge \bigwedge_{i=1}^{n} (\varphi \oplus \varphi'_i) \equiv \left(\delta \wedge \varphi \wedge \bigwedge_{i=1}^{n} \neg\varphi'_i \right) \vee \left(\delta \wedge \neg\varphi \wedge \bigwedge_{i=1}^{n} \varphi'_i \right)$$

[5]For instance, MiniSat [33] provides the method addClause, Yices [34] and Z3 [35] have the instruction assert.

Algorithm 6 canCollect – Incrementally collecting with backtracking

```
function canCollect(tp, $C$)
    if $C = \emptyset$ then
        $ctx \leftarrow$ makeNewContext
        $ctx$.assert($\delta$)
    else
        $ctx \leftarrow$ getContext($C$)
    end if
    $ctx$.push() // save current context
    $ctx$.assert(tp) // add tp to the current context ctx
    if $ctx$.solve() then
        return true
    else
        $ctx$.pop() // restore previous context
        return false
    end if
end function
```

to simplify the collecting process. Thanks to this equivalence, one can start with two contexts: $c_\top$ initially containing only $\delta \wedge \varphi$, and $c_\bot$ containing $\delta \wedge \neg\varphi$. When a test predicate $tp_i = \varphi \oplus \varphi'_i$ must be checked for compatibility with all the test predicates already collected, $\neg\varphi'_i$ is added to $c_\top$ (if still valid), while φ'_i is added to $c_\bot$(if still valid). After the addition, we can have one of the following three cases:

1) if both contexts are still satisfiable, then tp_i is accepted;
2) if only one context is satisfiable, then tp_i is still accepted, but the context without model is invalidated and no longer considered for the collection until the next new collecting process;
3) if no valid context is satisfiable, then tp_i is refused and the valid contexts are restored.

5.3. Limiting Collecting

To make the collecting process of test predicates faster, another approach consists in limiting the test predicates that can be possibly collected, modifying the quitCollecting function (line 13 in Alg. 3). The collection would not contain all the uncovered test predicates which could be possibly collected (we can say that it is a *partial* collection), and this may reduce the effectiveness of the collecting process itself. However, this negative effect should be reduced by the coverage evaluation: if the test generated for a collection covers also test predicates which could have been collected, then these test predicates are marked as covered by the coverage evaluation (⑤ in Fig. 2 and line 9 in Alg. 2) and no longer considered.

O.C_ QN *Quit after N*
A first limitation regards the maximum *number* of test predicates to be possibly added to a collection. Once that the collection contains N test predicates, the collecting process stops. This policy is very easy to implement; the modified quitCollecting function is shown in Alg. 7. With very small N, it makes the collecting process very fast, but it may produce bigger test suites. With bigger N, it behaves similarly to the unlimited collection, but also the time may increase.

Algorithm 7 Quit collecting with bound N on the size of the collection

```
function quitCollecting($C$)
    return $|C| = N$
end function
```

O.C_ UU *Collecting until useful*
Another policy consists in performing the collecting until it is useful, but stopping it as soon as it becomes useless. Indeed, when the model of the collected test predicates is the only one, collecting could be stopped without losing anything: any new test predicate that could be added to the collection would be covered in any case in the coverage evaluation step (⑤ in Fig. 2), by the test produced by the SAT/SMT solver for the collection. We devise the following technique in order to discover if a model of a predicate is unique.

Let *asExpr* be a function that, given a model m, returns a Boolean predicate having m as unique model. The simplest *asExpr* is the function that returns the conjunction of the variables having value *true* in m and the negation of the variables having value *false* in m.

Example 6. If $m = \{a = \top, b = \bot\}$ is the model, then $asExpr(m) = a \wedge \neg b$.

The following proposition indicates how to check if a model of a Boolean predicate is unique.

Proposition 5.1. *Let ψ be a predicate and m be a model of ψ. m is the unique model of ψ if $\psi \wedge \neg asExpr(m)$ is not satisfiable.*

We can use Prop. 5.1 to check if the model of the collection is unique, right after a tp is added to the collection. If the model is unique, the collection process can be stopped.

Example 7 (Checking if a model is unique). Let $C = \{a, \neg b\}$ be a collection of test predicates and $m = \{a = \top, b = \bot\}$ be one of its models. The predicate $a \wedge \neg b \wedge \neg asExpr(m) \equiv a \wedge \neg b \wedge \neg(a \wedge \neg b)$ is unsatisfiable, therefore m is the unique model of C.

Note that, while optimization O.C_QN may produce bigger test suites because it may leave out from the collection some uncovered test predicates that should be collected, this optimization collects all the useful test predicates and, therefore, it does not impact on the test suite size.

The modified quitCollecting function is shown in Alg. 8.

Algorithm 8 Quit collecting with uniqueness checking

```
1: function quitCollecting($C$)
2:     $m \leftarrow$ modelSat($\delta \wedge \bigwedge_{c \in C} c$)
3:     return $\neg$sat($\delta \wedge \bigwedge_{c \in C} c \wedge \neg asExpr(m)$)
4: end function
```

Algorithm 9 Collecting process with uniqueness checking after N

```
1: function quitCollecting($C$)
2:     if $|C| \geq N$ then
3:         $m \leftarrow$ modelSat($\delta \wedge \bigwedge_{c \in C} c$)
4:         return $\neg$sat($\delta \wedge \bigwedge_{c \in C} c \wedge \neg asExpr(m)$)
5:     end if
6:     return false
7: end function
```

O.C_ UAN *Checking uniqueness after N*
This optimization starts checking the uniqueness of the model only after N test predicates have been added to the collection. The first N predicates are added (if possible) in the classic way. Afterwards, the collecting process checks if the *witness* of the collection is unique and, in this case, it quits.

The modified quitCollecting function is shown in Alg. 9.

6. EXPERIMENTAL RESULTS

To evaluate the approach described in this paper, we perform a set of experiments, trying to measure the effects of the described options and optimizations in terms of test generation time.

Note that the model obtained for a given test predicate is independent of the solver used (indeed, we assume that each solver returns with equal probability one of the possible models of the test predicate). Moreover, all the proposed options and optimizations (except for the limiting optimizations) do not affect the set of models of a formula, since they either modify the formula into an equivalent form (e.g., O.X) or speed up the test generation (e.g., O.C_W) by keeping the formula unchanged. The model of a test predicate and its individual coverage only depend on the order in which test predicates are collected. Therefore, the size of the final test suite, when collection is not limited, only depends on the order in which test predicates are collected. In conclusion, options, optimizations, and the used solver influence the time to find a model of every collection but not the model itself (i.e., the value assignments). They do not affect the coverage of every test and, therefore, the final test suite size.

We first present a set of benchmarks and introduce the considered SAT and SMT solvers. In Sect. 6.1, we describe the evaluation of the options regarding the generation of the test predicates (Sect. 6.1.1), the solver option (Sect. 6.1.2), and the specification form options (Sect. 6.1.3). Sect. 6.2 reports a comparison among all the solvers we consider. In Sect. 6.3, we evaluate the basic optimizations O.C_W, O.C_WM, O.C_IB and O.C_DI regarding the collecting process, using the solver that had the best performances in the experiment described in Sect. 6.2 and that supports the tested optimizations. In Sect. 6.4, we evaluate the optimizations O.C_QN, O.C_UU and O.C_UAN regarding the limiting of the collecting process. Finally, in Sect. 6.5, we evaluate how good is our collecting algorithm compared to the minimal solution.

	Specification source			
	Source code	ISCAS	NuSMV	All
#original specs	89332	99507	131	188970
#simplified specs	461	372	79	801

TABLE 1: Number of selected specifications

We run all the experiments on a Linux PC with 24 Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz and 64 GB of RAM. Since our algorithms have some sources of nondeterminism (mainly the order in which the test predicates are considered), they may return slightly different results in different executions over the same inputs; for this reason, each experiment has been executed 50 times and this explains the error bars in the graphs.

Benchmarks For experimentation, several specifications have been taken from different sources (as shown in Table 1). 89332 specifications have been extracted from source code of projects hosted on SIR[6]. We have considered libraries (including apache-ant-1.9.1 and jakarta, for instance) and safety critical software (like JTCAS and Elevator, for instance); totally, we considered around 2 MLOC. 99507 specifications have been extracted from circuit models in ISCAS format [36]. Finally, we extracted 131 specifications from the guards of models of the NuSMV model checker; we considered all the models hosted on the website of NuSMV, and all the NuSMV models we could find on the Internet.

The specifications and the constraints have been simplified in the following way. In a constraint, the subexpressions not related to the expression (i.e., not containing any variable of the expression or any variable that may influence the model of the expression) have been removed; for example, given the expression $a \vee b$ and the constraint $\neg a \wedge c$, the constraint can be simplified as $\neg a$.

Then, the number of simplified expressions has been reduced by identifying isomorphic specifications (i.e., specifications that can be transformed one in the other by a simple renaming of the inputs); for example, specifications $a \wedge b$ and $c \wedge d$ are isomorphic. For each group of isomorphic specifications, only one

[6]Software-artifact Infrastructure Repository http://sir.unl.edu

representative specification has been kept, associated with the size of the group.

In the end, we obtained 801 single specifications (Table 1 reports also the number of simplified specifications from the different sources); in order to keep the execution times reasonable, and to avoid very rare expressions that could invalidate the results, we selected the more common specifications. We have sorted the singular specifications in decreasing order of representativeness, and we have selected the first 119 singular specifications, that are representative of the 99% of the specifications originally retrieved[7]. On average, each selected specification contains 9 variables and 34 Boolean operators; the maximum number of variables in an expression is 76, and the maximum number of Boolean operators is 804. We believe that the selected specifications are a meaningful sample of the complexity of Boolean conditions usually occurring in actual software.

SAT/SMT solvers As SAT solvers, we select SAT4J [18], MiniSat [33], CryptoMiniSat [37], PicoSat [38] and NFLSAT [19]. SAT4J [18] is a mature, open source library of SAT-based solvers for Java, SAT4J can be embedded in Java and does not require any exchange of files. However, it requires that the test predicates are transformed into CNF. SAT4J (partially) supports incremental solving of SAT problems. MiniSat is a minimalistic, open-source SAT solver, which proved to be very effective in all the SAT competitions over the past years. CryptoMiniSat natively supports xor clauses, so it seems suitable for dealing with our test predicates. We also add PicoSat since it claims to support incremental SAT checking. NFLSAT is a SAT solver for non-clausal formulae; the input to NFLSAT is a Boolean circuit in *And Inverter Graph* (AIG) format or *ISCAS* format.

As SMT solvers, we use Yices [34] and Z3 [35]. Yices includes a very efficient SAT solver; it claims to be "*competitive as an ordinary SAT and MaxSAT solver*" [34]. Z3 is a high-performance SMT solver, developed at Microsoft research.

To implement optimization O.API we use the solvers (if possible) together with the Java Native Access (JNA) libraries, which simply require native shared libraries.

A brief comparison of the capabilities of the solvers is reported in Table 2 (Y? means that the feature is partially supported). MiniSat and PicoSat support a limited form of backtracking when all the added constraints are unit clauses and this is not enough for implementing O.C_I. CryptoMiniSat does not provide suitable APIs, and so incremental collecting and backtracking are not applicable. The incremental version of SAT4J does not work as expected[8]. NFLSAT does not require inputs as CNF, but it cannot work with JNA since it comes as executable binary. Yices and Z3 have very rich APIs, accept GF Boolean expressions, and support a very efficient backtracking technique.

Solver	**O.API**	**O.GF**	**O.C_I**	**O.C_IB**
	API	GF (vs CNF)	incremental	backtracking
SAT4J	Y	N	Y?	N
MiniSat	Y/N	N	Y?	N
CryptoMiniSat	N	N	N	N
NFLSAT	N	Y	N	N
PicoSat	Y/N	N	Y?	N
Yices	Y/N	Y	Y	Y
Z3	Y/N	Y	Y	Y

TABLE 2: Solver features

6.1. Options Evaluation

In this section we evaluate the effect of applying the options described in Section 4. Table 3 reports, for each option, the improvement (average, minimal and maximal) due to different option values. The improvement of time t due to the use of option value **A** w.r.t. option value **B** is defined as $1 - t_A/t_B$.

6.1.1. Test predicates generation options

MCDC Test predicates We compare the two possible ways of generating the test predicates from MCDC (using solver SAT4J). The experiment shows that CACC is, on average, 13.4% faster than GACC (as shown in Table 3). *GACC, using the Boolean derivative, produces test predicates more difficult to solve than those produced by CACC.*

XOR Simplification Now, we want to assess the influence of O.X option. We run all the solvers (using O.API whenever possible) with and without the O.X option. We discovered that the simplification of the xor operator produces, on average, an *improvement of around 37%* of the test generation time (as shown in Table 3). This optimization could be directly embedded in the SMT solvers (that accept formulae in general form), which probably do not apply this form of simplification because it is rarely applicable for generic Boolean expressions.

6.1.2. Solver Invocations Option

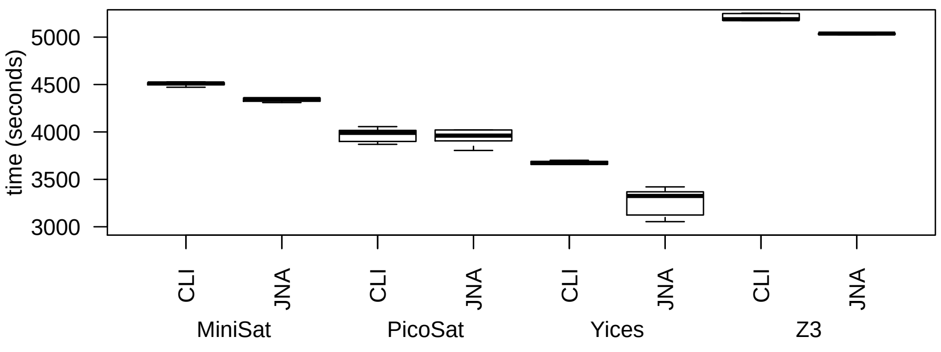
API We want to determine whether the application of option O.API has some benefits. We consider only solvers MiniSat, PicoSat, and Yices, that allow both versions, one at command line (CLI) and the other one using JNA. Fig. 4a reports the time taken to generate the complete test suite for the selected 119 specifications for 50 runs. We see that *using the JNA version is always worthwhile*; Table 3 reports that using the JNA version decreases, on average, the test generation time of 6.2%.

[7]All the used specifications and a jar of the test generation tool can be downloaded from http://fmse.di.unimi.it/atgtBoolean.html.

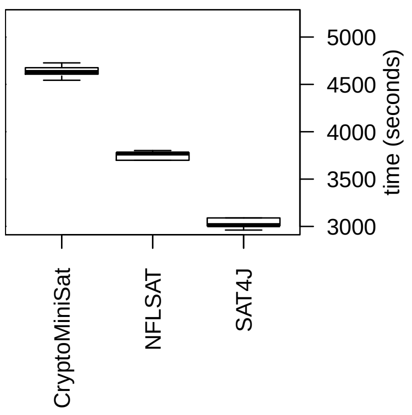
[8]The SAT4J command `removeConstr` accepts only simple clauses and, as stated in the official documentation, it is only partially implemented.

Option	Description (A vs B)	Improvement		
		Average	Min	Max
O.MCDC	using CACC vs using GACC	13.4	7.4%	18.7%
O.X	applying O.X vs not applying O.X	37.1%	30.8%	42.1%
O.API	JNA vs CLI	6.1%	4.8%	7.2%
O.CNF	using Tseitin transformation vs using equivalence preserving transformation	69.7%	42.9%	85.1%
O.GF	General form vs CNF	16.3	11.9%	28.2%

TABLE 3: Options evaluation



(a) O.API optimization



(b) Other solvers

FIGURE 4: Solvers

6.1.3. Specification Form Options

CNF Conversion We want to assess the influence of the transformations to CNF (O.CNF). We test two CNF transformations, one that preserves equivalence and the Tseitin transformation [31]. As solvers, we consider those accepting only the CNF format (MiniSat, PicoSat and SAT4J). The experiment confirms that the *Tseitin algorithm is more efficient* also for test generation, even if it increases the number of literals (and therefore increases the model size). As shown in Table 3, using the Tseitin algorithm speeds up the test generation time, on average, of 69.7%.

General form We now evaluate whether the optimization O.GF is effective. We take all the solvers accepting formulae in general form (NFLSAT, Yices and Z3) and we compare the use of test predicates in general form and that in CNF. As shown in Table 3, *using the formulae in general form speeds up, on average, the test generation process of 16.3%.*

Remark. In the experiments that follow, all the options proved to be effective are applied, unless stated otherwise.

6.2. Solver comparison

We here compare all the solvers and Fig. 4 shows their test generation times. SAT4J is the best solver, however, it does not support optimizations O.C_I and O.C_IB. Yices is the second best and supports all the collection optimizations. For this reason, we choose Yices to perform the remaining experiments in which we compare more deeply the optimizations regarding the collecting algorithm.

6.3. Collecting Optimizations Evaluation

We run the test generation algorithm over the 119 specifications for 50 runs, applying the optimizations regarding collecting without limits, namely the use of witnesses in three variants (not used, O.C_W, and O.C_WM of Sect. 5.1), and incremental collecting in three variants (not applied, single incremental collecting with backtracking O.C_IB, and double incremental collecting O.C_DI, as explained in Sect. 5.2). We do not experiment O.C_I since it is superseded by O.C_IB and because it has proved to be ineffective in experiments not reported here.

Totally, there are 9 possible combinations of optimizations, but only 7 are feasible when generating tests for fault-based criteria: optimization O.C_WM cannot be applied with incremental collecting (both single and double) because a test predicate must be added in any case to the logical context in order to allow incremental construction. When generating tests for the MCDC criterion with the CACC method (the method we have chosen in the experiment in Sect. 6.1.1), only 5 combination of optimizations, among those used for fault-based criteria, are feasible: indeed double incremental collecting (O.C_DI) cannot be applied to test predicates generated with CACC, since they do not contain the *xor* operator.

The optimization O.X is applied except when it is not possible: it cannot be applied together with the double collecting O.C_DI because double collecting requires the test predicates in the *xor* form.

First, we analyze the data in order to assess the effect of every single optimization. Table 4 reports the observed improvements due to optimization O.*i*. It compares the time (as average and deviation) required to complete the test generation for all the specifications when O.*i* is not applied (column *without*) with the time when O.*i* is applied (column *with*), considering all the combinations of the other optimizations where O.*i* is applicable. We can draw the following conclusions.

The only ineffective optimization is O.C_W: it seems that an extra call of the SMT solver after the collecting phase does not impact over the final time. The time spent by an extra call to the solver to compute the final collection model is comparable with the time necessary to read the model every time after a test predicate is added to the collection.

The other options are effective. Checking if the witness of the collection is also a valid model for the test predicate being added to the collection (O.C_WM) is rather effective. The *incremental collecting (O.C_IB and O.C_DI) boosts the performances* by significantly reducing the time. Double incremental collecting is slightly more efficient than single incremental collecting. This confirms the importance of providing incremental solving interfaces also for test generation.

Second, we analyze the data in order to compare the combinations of optimizations and to find the best ones. Fig. 5 reports the time required to complete the test generation for all the considered specifications depending on the optimizations used. It is apparent that the incremental collecting (both single and double) significantly improves the performances. The best combination for fault-based is O.C_DI, and for MCDC is O.C_IB with O.X. Table 5 reports the generation time of these two combinations. However, double collecting cannot be used for the MCDC test predicates and, therefore, single incremental collecting with the *xor* simplification (O.C_IB and O.X) is the best optimization over both the criteria. Such setting requires only 105.79 secs for the whole generation, with an average of 1.03 secs for specification. This proves that *a well engineered and optimized SMT-based test generation process can be used in practice for Boolean specifications instead of the classical algorithms*, which are fast as well but produce much larger test suites [9].

6.4. Limiting Collecting Evaluation

In this experiment, we test how all the limiting policies introduced in Sect. 5.3 may affect the test generation process (used together with incremental collecting with backtracking O.C_IB). We run the test generation for all the selected specifications for 50 runs with

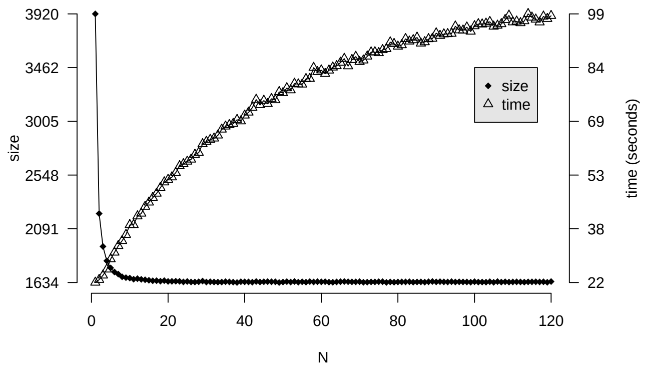


FIGURE 6: Quit collecting after N (O.C_QN)

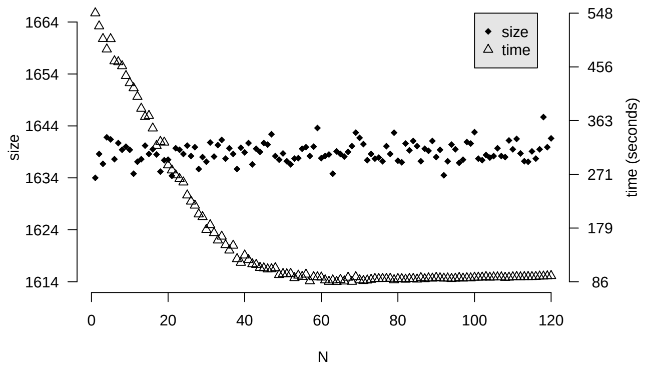


FIGURE 7: Checking uniqueness after N (O.C_UAN)

O.C_QN, O.C_UU, and O.C_UAN; for O.C_QN and O.C_UAN we experimented N from 1 to 120. Table 6 reports the test suite size and the time (average and deviation). These policies are compared (column ± wrt §) with the best result obtained by single incremental collecting (reported again in the first row § in the table).

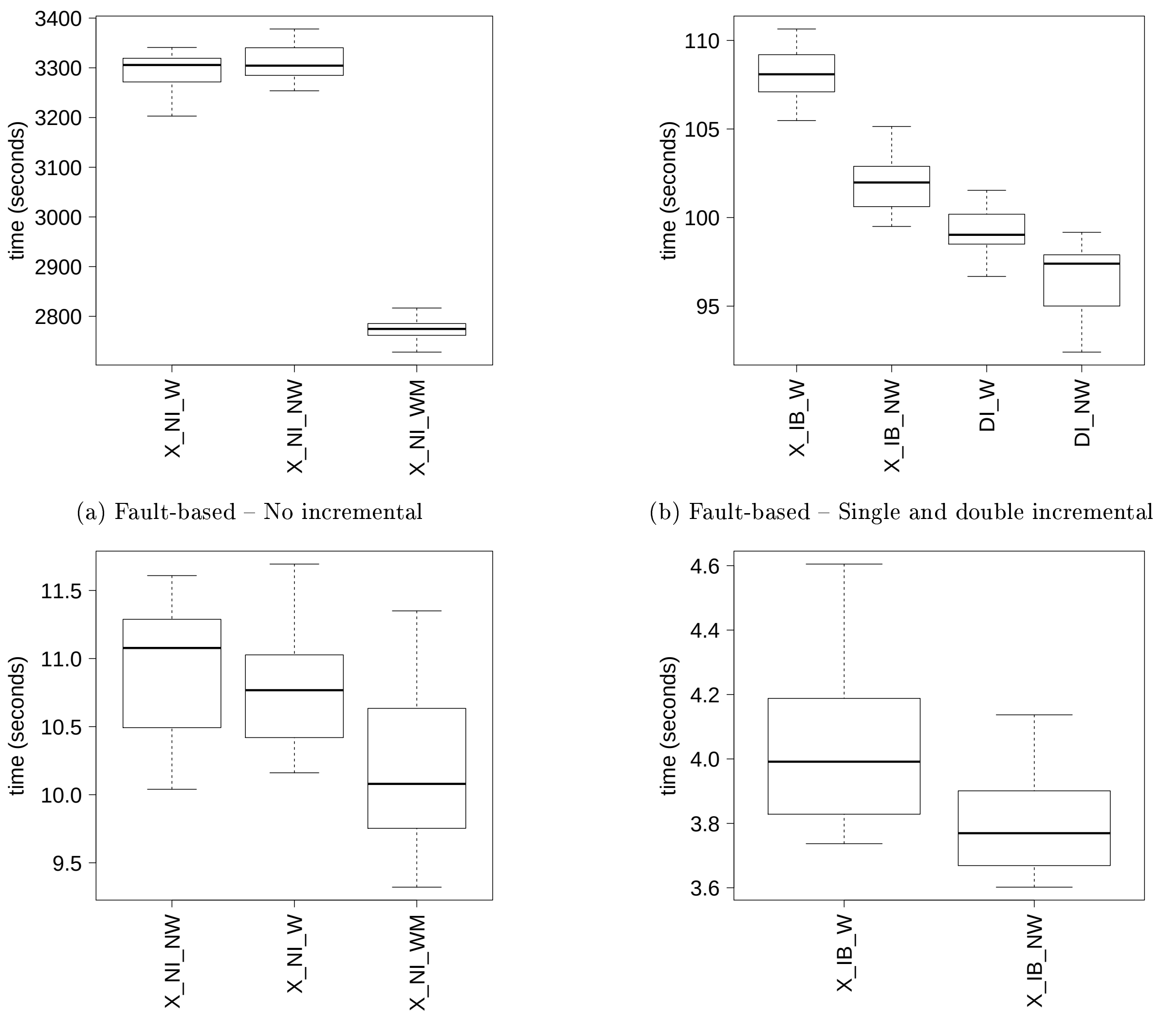
Fig. 6 depicts the effect of quitting the collecting after N iterations (O.C_QN). As the figure shows and as expected, the size of the test suite decreases with increasing N, but the time required increases as well. For small N the size rapidly decreases, but after a threshold (around 15) the test suite size is reduced only marginally. This option gives the user more control over the collecting process: a suitable value of N can be chosen to balance between test suite compactness and test generation time. Note that the test suite becomes of comparable size w.r.t. the case without limiting (§) when N approaches 30, but the generation time still remains smaller (see Table 6).

Collecting until the uniqueness of the model is reached (O.C_UU) can find a test suite as small as the best combination §, but the time required is quintupled.

Fig. 7 depicts the effects of starting checking the uniqueness after N (O.C_UAN), which is capable of finding test suites as small as the best combination §; for values of N lower than around 50, the time is greater, while, for values of N greater than 50, the time is lower. We can argue that the collections contain, on

Optimization	Faults				MCDC				Overall	
	Avg. time (sec)		Improvement		Avg. time (sec)		Improvement		Improvement	
	without	with	avg.	dev.	without	with	avg.	dev.	avg.	dev.
O.C_W	8023.6	8032.02	-0.11%	1.09%	30.02	29.84	0.53%	4.14%	-0.11%	1.08%
O.C_WM	7696.47	6482	15.78%	0.79%	21.63	20.19	6.58%	4.03%	15.75%	0.79%
O.C_IB	3849.35	115.54	97%	0.02%	10.89	4.07	62.6%	1.49%	96.9%	0.02%
O.C_DI	4396.5	98.04	97.77%	0.03%	N/A	N/A	N/A	N/A	97.77%	0.03%

TABLE 4: Optimization improvements



(a) Fault-based – No incremental

(b) Fault-based – Single and double incremental

(c) MCDC – No incremental

(d) MCDC – Single and double incremental

FIGURE 5: Optimization evaluation with Yices – X: xor simplification (O.X), NI: no incremental collecting, IB: single incremental collecting with backtracking (O.C_IB), DI: double incremental collecting (O.C_DI), W: collection with witness (O.C_W), WM: checking if the witness is a model (O.C_WM)

average, about 50 test predicates. Indeed, for values of N lower than 50, the times for O.C_UAN are greater than the time of the best combination §: this means that, most of the times, the uniqueness check fails (the model is not unique) and the time taken by the check is wasted. For values of N greater than 50, instead, the times for O.C_UAN are lower than the time of the best combination §: this means that, most of the times, the uniqueness check succeeds (the model is unique) and the collecting can be interrupted, while in the best combination § the collecting must be executed on all the test predicates.

Optimizations		Faults			MCDC			Overall		
		Time (secs)			Time (secs)			Time (secs)		
Collecting	others	avg.	min	dev.	avg.	min	dev.	avg.	min	dev.
Single (O.C_IB)	O.X	101.92	99.5	± 1.54	3.87	3.6	± 0.34	105.79	103.88	± 1.55
Double (O.C_DI)		96.61	92.41	± 1.85	N/A	N/A	N/A	96.61	92.41	± 1.85

TABLE 5: Time for the best optimizations

Policy	Test suite Size			Time (secs)		
	average	dev.	± wrt §	average	dev.	± wrt §
§ no limits (O.C_IB)	1638.6	5.12		105.79	1.55	
Fixed N (O.C_QN)						
1	3919.6	13.97	139.2%	22.48	2.33	-78.75%
10	1671.3	5.76	2%	38.94	3.02	-63.19%
15	1652.6	5.04	0.85%	45.39	1.81	-57.09%
20	1643	4.78	0.27%	51.9	1.8	-50.94%
30	1637.4	5.62	-0.07%	62.76	1.86	-40.67%
40	1639.4	6.26	0.05%	70.27	1.37	-33.58%
60	1639.2	6.41	0.04%	83.17	1.56	-21.38%
80	1638.3	4.5	-0.02%	89.96	2.25	-14.96%
100	1640.5	9.25	0.12%	95.8	1.92	-9.44%
120	1641.9	5.02	0.2%	98.74	1.77	-6.67%
Until unique (O.C_UU)	1641	5.16	0.15%	548.2	28.83	351.1%
Mixed (O.C_UAN)						
1	1634.5	6.31	-0.25%	547.94	55.4	417.96%
10	1639.9	6.06	0.08%	427.79	31.63	304.39%
20	1638	5.16	-0.04%	286.63	18.55	170.95%
30	1637.6	5.95	-0.06%	175.76	17.29	66.14%
40	1639.4	7.21	0.05%	131.64	16.01	24.44%
45	1641.2	4.34	0.16%	109.54	8.88	3.55%
50	1639.2	5.53	0.04%	99.68	8.46	-5.78%
60	1638.3	5.46	-0.02%	93.71	5.51	-11.41%
80	1637.8	6.48	-0.05%	91.26	8.19	-13.74%
100	1643.3	8.22	0.29%	93.5	1.77	-11.62%
120	1642.1	5.61	0.21%	96.37	1.6	-8.9%

TABLE 6: Comparison of limiting policies

Overall, we can state that *limiting the collecting is effective in reducing the time for test generation* with possible no negative effects over the test suite size.

6.5. Optimality of the Collecting Process

As mentioned in Sect. 3.1, the optimal partition of the test predicate set would guarantee the minimal number of partitions and this would ensure the minimality of the final test suite. However, the proposed greedy collecting process depends on the order in which test predicates are considered and it may not find the optimal partition of the test predicates. We are interested in measuring how much the solutions computed by such greedy algorithm differ from the optimal one.

With this goal, we run the test generation process 2000 times for all the specifications. For each specification we identify, among the 2000 runs, the smallest value for the test suite size which is likely very close or equal to the minimal optimal value. Then we check how much, in all the runs, the values obtained are bigger than the minimum. Fig. 8 shows the cumulative distribution of the difference of the test suite size for every run and for every specification w.r.t. its minimum. As shown by Fig. 8, for around 70% of the cases we obtain the minimum, for almost 95% of the cases the size of the test suite is maximum 10% bigger than the (optimal) smallest test suite. The difference is never greater than 30%. However, in particular cases, it may be necessary to use some heuristics in the test predicate ordering to guarantee the best results.

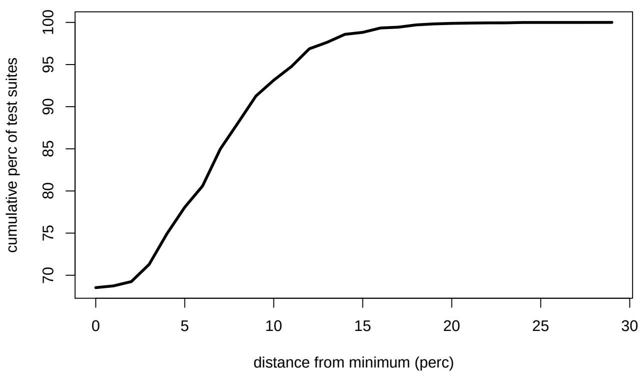


FIGURE 8: Optimality of the Collecting Process

7. THREATS TO VALIDITY

This section discusses the possible threats to validity of the empirical evaluation and explains how the evaluation addressed each threat.

Benchmarks completeness The benchmark set has been obtained by analyzing a large number of case studies, ranging from software to hardware, from commercial software to safety critical systems. The experiments were conducted on a large number of Boolean expressions representing the 99% of the occurrences of the gathered Boolean expressions, while Boolean expressions that rarely occur were ignored. We do not consider expressions that may be not representative of the application domain of our technique, for instance those coming from SAT and SMT competitions. Although different results for a particular expression cannot be excluded, the improvements brought by some techniques are so strong that a small variation cannot change the conclusions.

SAT and SMT solvers There may exist a SAT or SMT solver which performs even better than those experimented. However, the set of several SAT and SMT solvers has been built with particular regard to tools that have performed well in official competitions but also that have particular features helping the test generation (xor native support of CryptoMiniSat, general form inputs for NFLSAT, Java native implementation of SAT4J, and backtracking for SMT solvers). We have carefully reviewed the existing literature and tried to cover all the suitable tools.

Option Interaction In the evaluation of the options, we have evaluated each option singularly, without considering all the possible combinations. Therefore, there could be a combination (not considered in our experiments) that gives better results. However, the improvements are strong enough and the options are independent enough that a possible meaningful further improvement due to the unforeseen interaction of options seems unlikely.

Internal validity The use of a large source for Boolean expressions and a numerous set of solvers reduces the threats to internal validity, i.e., the fact that the final results may strongly depend on the choices done at the input level.

External validity A major external threat to validity is that the obtained results may not be easily extended to other types of testing. The assumption that complex expressions are abstracted in order to obtain Boolean expressions requires that the tests are eventually concretized to real values, and this may threat the actual application of our technique when testing non-Boolean expressions. Especially, when Boolean expressions are embedded in code (for programs) or in guards of events (for models), the concretization process may require to solve complex (path) constraints in order to make the concrete test reach the Boolean expression under test. However, our approach deals with constraints and tries to produce very compact test suites, and this facilitates the process of concretization. Moreover, the proposed use of SMT solvers may make our technique more (than the use of pure SATs) suitable to directly test expressions containing integer arithmetic without the need of any abstraction.

8. RELATED WORK

Regarding test generation for Boolean expressions, there exist many testing criteria for Boolean expressions. A survey counts at least 12 testing criteria [21], including (using the nomenclature common in white box testing literature) decision coverage, condition coverage, multiple condition coverage, and the MCDC [6]. Several syntactic testing criteria have been developed to directly target common faults in Boolean expressions. For instance, Weyuker et al. introduced a family of strategies (MAX-A and MAX-B) for automatically generating test cases from Boolean expressions [5]. These criteria were later improved by the MUMCUT criterion which comes in several versions [7]. The test generation of these criteria is defined by means of suitable algorithms which consider the internal structure of the expressions and generate an adequate test suite. With respect to these algorithms, our SAT/SMT-based technique is much slower but it offers several advantages in terms of applicability (including dealing with constraints) and produces smaller test suites. For instance, in [9], we were able to generate test suites 58% smaller and with a fault detection capability 56% greater than the test suites produced by Minimal-MUMCUT.

Regarding the use of SAT and SMT solvers for test generation, there are many approaches trying to use such tools for test generation from code [39]. In several works, symbolic execution, constraint generation, and constraint solving are combined with test generation and SMT solving to satisfy classical code coverage criteria. For example, solvers are used in concolic testing, in which symbolic execution is combined with concrete execution; Godefroid et al. developed the DART tool [40], Sen et al. developed the CUTE tool [41], Cadar et al. developed EXE [42] and then redesigned it with KLEE tool [43]. The Pex tool uses dynamic symbolic execution and SMT solving to determine test inputs for .NET code [13]. A similar tool is SAGE [12]. We believe that these tools could reuse some optimizations here proposed, and some optimizations of theirs could be reused in our approach as well. For instance, SAGE uses a simple syntactical checker to simplify the constraints by eliminating constraints logically implied by other constraints injected at the same program branch. We could use a similar check when collecting a test predicate: if it is implied by the test predicates, then it can be discarded. Our experiments confirm that simple syntactical optimizations like O.X can greatly improve test generation performances. The EXE tool could extend its constraint caching optimization, which caches constraints and satisfability results using hashes, with our optimization O.C_W which avoids the call of the solver in more cases than simple syntactical caching.

For model-based testing, such solvers are not so popular, but they have been recently used in several works. For instance, a SAT solver has been used to generate tests from the specification of pre-post conditions of Java programs [44]. In [45], an SMT solver has been used for bounded reachability analysis of model programs. Given a model program in terms of an abstract state machine, the Z3 SMT solver is used to find paths and verify invariants of the models. Although the goal is not to generate tests, the approach is similar to a test generation process and also the solver is used in an incremental way, similar to what proposed by optimization O.C_IB. An approach using SAT solving for test generation is presented in [46]. In that paper, specifications are given in terms of Timed Moore Automata (TMA), a temporal extension of finite state machines. A comparison between model checking and SAT solving for test generation is presented. Representing TMAs as SAT problems requires a complex translation which may be simplified if an SMT solver is used instead.

In general, the use of SMT solvers may make the process of abstraction and concretization superfluous, since SMT solvers can directly deal with non-Boolean variables. This capability has been exploited in [47, 48].

Abstract interpretation techniques can be combined with SMT solving in an efficient way as proposed in [47]. Their approach already exploits incremental calls to the SMT solver (SONOLAR), which allows them to add constraints between solver runs and to add constraints that are only valid for one run (so-called assumptions). It could be combined with backtracking to further speed up the test generation process. However, they generate test sequences and they do not apply any collection algorithm. A future work is to exploit the collection algorithm presented here for test sequence generation in order to reduce the length of test traces.

The general approach presented in Sect. 3 has been applied to combinatorial test generation in [49], however, without any optimization here proposed. Combinatorial testing with the aim of path coverage of a state machine is studied in [50] which uses the Z3 SMT solver. The use of the SMT solver allowed the authors to solve test requirements about parameter combinations and path conditions together with constraints over the actions. The incremental construction is applied to the construction of the exclusion constraints in a similar way as our O.C_I.

In hardware testing, SAT solving is increasingly used for test generation, which is classically called automatic test pattern generation (ATPG). However, hardware has different fault models (the most common is the Single Stuck-at Fault - SSF), and tests are generally a sequence of inputs possibly containing also timing information. SAT can be efficiently used to detect SSF [51]. Moreover, also for other typical hardware faults, SAT-based ATGP can be used. For instance, in [52] the authors show how to generate tests that are able to detect delay faults. Also in hardware testing, there exist some attempts to apply dynamic test compaction, which is similar to our collecting algorithm, combined with the use of SAT solving [53]. The fact that SAT solving is increasingly used in hardware testing strengthens our thesis that SAT and SMT solving is a viable technique also for software test generation.

9. CONCLUSION AND FUTURE WORK

We have presented a set of options and optimizations to improve a process of automatic test generation for Boolean expressions by SAT/SMT techniques. Although we propose and apply the optimizations only for a selected number of SAT and SMT solvers and for fault-based and MCDC testing of Boolean expressions, most techniques are general enough and can be applied to other approaches as well, to speed up the test generation even for non-Boolean expressions. Some optimizations exploit specific features of a SAT or an SMT solver, others require specific forms of the input formulae or are applicable only to fault-based test predicates. However, most of the proposed optimizations modify the test generation process and can be applied regardless the notation and the tool used for test generation.

Experimenting these optimizations through a set of benchmark case studies, we make apparent that a well engineered and optimized SAT/(but better an)SMT-based test generation process can be used in practice for Boolean specifications instead of the classical algorithms like MUMCUT, MCDC.

As future work, we plan to study the application of the proposed optimized test generation method to other testing approaches (e.g., constrained combinatorial interaction testing [49, 54]) and to specifications whose testing may originate more complex forms of test predicates that SMT solvers can still manage.

Moreover, we plan to extract specifications from evolving systems (e.g., FSMs, timed automata) by unfolding the behavior of the system in a *flat* expression, as it is done in bounded model checking. In this work, we have only considered Boolean expressions, obtained through abstraction from general expressions possibly containing arithmetics, strings, and so on; tests for Boolean expressions should be transformed in *concrete* tests for the original expressions through concretization. As future work, we want to compare this abstraction-concretization approach with the direct generation of tests from general expressions.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their suggestions and comments.

REFERENCES

[1] Batory, D. (2005) Feature models, grammars, and propositional formulas. In Obbink, H. and Pohl, K. (eds.), *Proceedings of the 9th International Conference on Software Product Lines*, Lecture Notes in Computer Science, **3714**, pp. 7–20. Springer-Verlag, Berlin, Heidelberg.

[2] Rajan, A., Whalen, M. W., and Heimdahl, M. P. (2008) The effect of program and model structure on MC/DC test adequacy coverage. *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, May, pp. 161–170. IEEE Computer Society.

[3] Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. (2001) Automatic predicate abstraction of C programs. *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, New York, NY, USA, May PLDI '01, pp. 203–213. ACM.

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

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

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

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

[8] Fraser, G. and Gargantini, A. (2010) Generating minimal fault detecting test suites for boolean expressions. *6th Workshop on Advances in Model Based Testing A-MOST*, April, pp. 37–45. IEEE Computer Society.

[9] Gargantini, A. and Fraser, G. (2011) Generating minimal fault detecting test suites for general boolean specifications. *Information and Software Technology, Elsevier*, **53**, 1263–1273.

[10] Malik, S. and Zhang, L. (2009) Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, **52**, 76–82.

[11] De Moura, L. and Bjørner, N. (2011) Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, **54**, 69–77.

[12] Godefroid, P., Levin, M. Y., and Molnar, D. (2012) SAGE: Whitebox fuzzing for security testing. *Commun. ACM*, **55**, 40–44.

[13] Tillmann, N. and De Halleux, J. (2008) Pex: white box test generation for .net. *Proceedings of the 2nd international conference on Tests and proofs*, Berlin, Heidelberg, April TAP'08, pp. 134–153. Springer-Verlag.

[14] Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E., and Zahlten, C. (2011) A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In Wolff, B. and Zaïdi, F. (eds.), *Testing Software and Systems*, Lecture Notes in Computer Science, **7019**, pp. 146–161. Springer Berlin Heidelberg.

[15] Gargantini, A. (2011) Dealing with constraints in boolean expression testing. *CSTVA 2011 - 3rd Workshop on Constraints in Software Testing, Verification, and Analysis*, March, pp. 322–327. IEEE Computer Society.

[16] Chen, T., Lau, M., Sim, K., and Sun, C. (2009) On detecting faults for boolean expressions. *Software Quality Journal*, **17**, 245–261.

[17] Arcaini, P., Gargantini, A., and Riccobene, E. (2011) Optimizing the automatic test generation by SAT and SMT solving for boolean expressions. In Alexander, P., Pasareanu, C. S., and Hosking, J. G. (eds.), *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, nov., pp. 388 –391. IEEE.

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

[19] Jain, H. and Clarke, E. M. (2009) Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts. *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, July, pp. 563–568. IEEE Computer Society.

[20] Ammann, P. and Offutt, J. (2008) *Introduction to Software Testing*, 1 edition. Cambridge University Press, New York, NY, USA.

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

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

[23] Kaminski, G. K. and Ammann, P. (2009) Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. *Proceedings of the 2009*

Second International Conference on Software Testing, Verification and Validation, April, pp. 356–365. IEEE Computer Society.

[24] Chen, Z., Chen, T. Y., and Xu, B. (2011) A revisit of fault class hierarchies in general boolean specifications. *ACM Trans. Softw. Eng. Methodol.*, **20**, 13:1–13:11.

[25] Ammann, P., Offutt, J., and Huang, H. (2003) Coverage criteria for logical expressions. *Proceedings of the 14th International Symposium on Software Reliability Engineering*, Washington, DC, USA ISSRE '03, pp. 99–. IEEE Computer Society.

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

[27] Offutt, J., Liu, S., Abdurazik, A., and Ammann, P. (2003) Generating test data from state-based specifications. *Software Testing, Verification, and Reliability*, **13**.

[28] Wallis, W. and George, J. (2010) *Introduction to Combinatorics*. Chapman and Hall/CRC.

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

[30] Chvátal, V. (1979) A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, **4**, 233–235.

[31] Prestwich, S. (2009) Handbook of satisfiability. IOS Press.

[32] Tseitin, G. (1968) On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, **2**, 10–13.

[33] Sörensson, N. and Eén, N. (2009) Minisat 2.1 and minisat++ 1.0-sat race 2008 editions. *Proceedings of SAT 2009*, pp. 31–32.

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

[35] de Moura, L. and Bjørner, N. (2008) Z3: an efficient SMT solver. *TACAS*, Berlin, Heidelberg, pp. 337–340. Springer-Verlag.

[36] Hansen, M. C., Yalcin, H., and Hayes, J. P. (1999) Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering. *Design Test of Computers, IEEE*, **16**, 72–80.

[37] Soos, M. (2009). Cryptominisat – a SAT solver for cryptographic problems. http://www.msoos.org/cryptominisat2/.

[38] Biere, A. (2008) PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, **4**, 75–97.

[39] Nori, A. V., Rajamani, S. K., Tetali, S., and Thakur, A. V. (2009) The Yogi project: Software property checking via static analysis and testing. In Kowalewski, S. and Philippou, A. (eds.), *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Berlin, Heidelberg, Lecture Notes in Computer Science, **5505**, pp. 178–181. Springer-Verlag.

[40] Godefroid, P., Klarlund, N., and Sen, K. (2005) DART: directed automated random testing. In Sarkar, V. and Hall, M. W. (eds.), *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI), Chicago, IL, USA, June 12-15, 2005*, pp. 213–223. ACM.

[41] Sen, K., Marinov, D., and Agha, G. (2005) CUTE: a concolic unit testing engine for C. In Wermelinger, M. and Gall, H. (eds.), *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE), 2005, Lisbon, Portugal, September 5-9, 2005*, Sep, pp. 263–272. ACM.

[42] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. (2006) EXE: automatically generating inputs of death. In Juels, A., Wright, R. N., and De Capitani di Vimercati, S. (eds.), *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pp. 322–335. ACM.

[43] Cadar, C., Dunbar, D., and Engler, D. R. (2008) KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Draves, R. and van Renesse, R. (eds.), *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA*, pp. 209–224. USENIX Association.

[44] Khurshid, S. and Marinov, D. (2004) TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, **11**, 403–434.

[45] Veanes, M., Bjørner, N., and Raschke, A. (2008) An SMT approach to bounded reachability analysis of model programs. *Conference on Formal Techniques for Networked and Distributed Systems*, Berlin, Heidelberg FORTE '08, pp. 53–68. Springer-Verlag.

[46] Löding, H. and Peleska, J. (2010) Timed moore automata: Test data generation and model checking. *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, Washington, DC, USA, April ICST '10, pp. 449–458. IEEE Computer Society.

[47] Peleska, J., Vorobev, E., and Lapschies, F. (2011) Automated test case generation with SMT-solving and abstract interpretation. *Proceedings of the Third international conference on NASA Formal methods*, Berlin, Heidelberg, NFM'11, **6617**, pp. 298–312. Springer-Verlag.

[48] Peleska, J. (2013) Industrial-strength Model-Based Testing - state of the art and current challenges. In Petrenko, A. K. and Schlingloff, H. (eds.), *Proceedings Eighth Workshop on Model-Based Testing, MBT 2013, Rome, Italy, 17th March 2013*, EPTCS, **111**, pp. 3–28.

[49] Calvagna, A. and Gargantini, A. (2010) A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, **45**, 331–358.

[50] Grieskamp, W., Qu, X., Wei, X., Kicillof, N., and Cohen, M. (2009) Interaction coverage meets path coverage by SMT constraint solving. In Núñez, M., Baker, P., and Merayo, M. G. (eds.), *Testing of Software and Communication Systems*, Lecture Notes in Computer Science, **5826**, pp. 97–112. Springer Berlin Heidelberg.

[51] Chen, H. and Marques-Silva, J. (2012) TG-pro: A SAT-based ATPG system. *Journal on Satisfiability, Boolean*

Modeling and Computation (JSAT), **8**, 83–88.

[52] Eggersglüß, S. and Drechsler, R. (2012) *High Quality Test Pattern Generation and Boolean Satisfiability.* Springer.

[53] Czutro, A., Polian, I., Engelke, P., Reddy, S. M., and Becker, B. (2009) Dynamic compaction in SAT-based ATPG. *2009 Asian Test Symposium*, pp. 187–190. IEEE Computer Society.

[54] Cohen, M., Dwyer, M., and Shi, J. (2007) Interaction testing of highly-configurable systems in the presence of constraints. *Proceedings of the 2007 international symposium on Software testing and analysis*, New York, NY, USA ISSTA '07, pp. 129–139. ACM.