AllSAT for Combinational Circuits

Dror Fried
Department of Mathematics and Computer Science, The Open University of Israel, Ra'anana, Israel

Alexander Nadel
Intel Corporation, Haifa, Israel
Faculty of Data and Decision Sciences, Technion, Haifa, Israel

Yogev Shalmon
Intel Corporation, Haifa, Israel
The Open University of Israel, Ra'anana, Israel

Abstract

Motivated by the need to improve the scalability of Intel's in-house Static Timing Analysis (STA) tool, we consider the problem of enumerating all the solutions of a single-output combinational Boolean circuit, called AllSAT-CT. While AllSAT-CT is immediately reducible to enumerating the solutions of a Boolean formula in Conjunctive Normal Form (AllSAT-CNF), our experiments had shown that such a reduction, followed by applying state-of-the-art AllSAT-CNF tools, does not scale well on neither our industrial AllSAT-CT instances nor generic circuits, both when the user requires the solutions to be disjoint or when they can be non-disjoint. We focused on understanding the reasons for this phenomenon for the well-known iterative *blocking* family of AllSAT-CNF algorithms. We realized that existing blocking AllSAT-CNF algorithms fail to generalize efficiently for AllSAT-CT, since they are restricted to Boolean logic. Consequently, we introduce three dedicated AllSAT-CT algorithms that are ternary-logic-aware: a ternary simulation-based algorithm `TALE`, a dual-rail&MaxSAT-based algorithm `MARS`, and their combination. Specifically, we introduce in `MARS` two novel blocking clause generation approaches for the disjoint and non-disjoint cases. We implemented our algorithms in our new tool `HALL`. We show that `HALL` scales substantially better than any reduction to existing AllSAT-CNF tools on our industrial STA instances as well as on publicly available families of combinational circuits for both the disjoint and the non-disjoint cases.

2012 ACM Subject Classification Mathematics of computing → Solvers

Keywords and phrases AllSAT, SAT, Circuits

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.9

Supplementary Material *Software*: `https://github.com/yogevshalmon/allsat-circuits`
archived at `swh:1:dir:38b9a6b1c31218fe227ef1848a74ea0a7f37d86e`

Funding *Yogev Shalmon*: This research was partially supported by The Open University of Israel's Research Fund (grant no. 41662).

Acknowledgements We are grateful to Supratik Chakraborty, Alexander Ivrii, Kuldeep Meel, Roberto Sebastiani, Mate Soos and Moshe Vardi for helpful discussions which played an important role in shaping our research. Also, we thank the anonymous reviewers for their comments and useful suggestions.

1 Introduction

Static Timing Analysis (STA) [42] is a crucial step in circuit design process that validates the timing performance of a circuit by checking all possible paths for timing violations. Given a circuit Δ, Intel's STA flow constructs a single-output combinational circuit Γ over Δ's inputs, such that Γ's output is 1 if and only if the inputs have the potential to trigger a timing violation in Δ. Then, the flow enumerates all the solutions in Γ, where every solution is tested for potential violations in the original circuit Δ[1].

[1] Further details are omitted due to IP considerations.

Motivated by a recent necessity to increase the scalability of Intel's STA flow due to increasing size of the input circuit, we study the so-called AllSAT-CT problem, which is a vital component in the flow. In *AllSAT-CT*, given a combinational circuit Γ with a single output, the objective is to enumerate all the possible inputs for Γ, for which Γ's output is 1.

AllSAT-CT is an instance of the *AllSAT* problem, in which the goal is to enumerate the solutions of a given Boolean formula. Another instance of AllSAT that received substantially more attention, is *AllSAT-CNF*, in which the input formula is provided in Conjunctive Normal Form (CNF). AllSAT-CNF has various applications, including software testing [22], data mining [7] and network verification [26]. AllSAT-CT can be immediately reduced to AllSAT-CNF by translating the circuit to CNF using, e.g., Tseitin encoding [45], and solved by AllSAT-CNF solvers. There are three main families of approaches to AllSAT-CNF, all implemented in state-of-the-art AllSAT-CNF tools, called herein `Toda` *tools (solvers)*, one per each family [44]. The first family, called *blocking* [28], applies an incremental SAT solver [9, 34] to find a solution (satisfying assignment) σ, then generalizes σ to a solution σ' (by replacing Boolean values by don't-cares, whenever possible), blocks σ' with the so-called *blocking clause* (which usually contains the negation of all the literals assigned 0 or 1 in σ') and iterates. The set of generalized solutions is the (compact) description of all solutions to Γ, typically in a form of a Disjunctive Normal Form (DNF) formula, in which every generalized solution is a cube. The second family of *nonblocking* solvers [14] modifies the SAT solver to enumerate the solutions explicitly without using blocking clauses. The third *BDD-based* family [17] is based on reasoning with Boolean Decision Diagrams (BDDs) [4]. Intel's STA flow used to routinely solve AllSAT-CT by using a BDD-based AllSAT-CNF solver until it ceased to scale due to excessive input size. We tried to apply the `Toda` tools, but they failed to scale either. Our investigation showed that, at least for the blocking algorithms, and independently of our specific industrial application, the existing AllSAT-CNF blocking approaches do not scale well for AllSAT-CT, since their solution generalization components are inherently inefficient, being restricted by Boolean logic semantics. Specifically, since one cannot explicitly assign don't-cares to the variables in Boolean logic, solution generalization's efficiency is hindered.

This insight has led us to introduce three dedicated AllSAT-CT blocking-based algorithms–`TALE`, `MARS` and `DUTY`–that utilize *ternary logic* [35] instead of Boolean, either for generalization only as in `TALE`, or throughout the entire algorithm as in `MARS` and `DUTY`. Notably, ternary logic-based approaches are applied by the Property Directed Reachability (PDR) algorithm for model checking [8, 40], but only for the generalization stage. Moreover, as detailed in [46], there is a distinction between AllSAT algorithms that return *disjoint* solutions, in which no two generalized solutions can overlap, and AllSAT algorithms that return *non-disjoint* solutions, in which such an overlap is allowed. We explicitly designed `MARS` to return either disjoint or non-disjoint solutions, per user request, while `TALE` and `DUTY` return non-disjoint solutions only.

Our first method `TALE` reduces AllSAT-CT to AllSAT-CNF by using Tseitin encoding [45] and applies a *ternary simulation*-based generalization algorithm, commonly used by PDR implementations [8]. Specifically, `TALE` generalizes a given solution by simulating whether reassigning a variable from a Boolean value to a don't-care value, under the ternary-logic semantic, would propagate through the circuit and still set the output to 1.

Our second approach, called `MARS`, also reduces AllSAT-CT to AllSAT-CNF but is using the dual-rail encoding [5] that allows one to preserve the ternary logic semantics in the CNF formula by explicitly representing don't-care values. Specifically, every variable v in the original circuit is mapped to two Boolean *dual-rail* variables (v^+, v^-) in the resulting

CNF, where assigning both v^+ and v^- to 0 corresponds to assigning the original v to a don't-care. With the dual-rail encoding, we trust the SAT solver to return a generalized solution by applying anytime MaxSAT-inspired heuristics [31, 32] to increase the number of don't-cares assigned to the circuit inputs (that is, the number of 0's assigned to their respective dual-rail variables). As mentioned before, `MARS` can emit both disjoint and non-disjoint solutions, where, unlike in [46], we achieve this by introducing two different blocking clause generation approaches. Furthermore, while generalization with MaxSAT in an auxiliary dual-rail-encoded CNF instance had been applied in PDR [40], our approach of using a single dual-rail-encoded instance throughout the algorithm, combined with a MaxSAT approximation for generalization and blocking clause generation in dual-rail, is novel.

Finally, since `MARS` uses approximate anytime MaxSAT heuristics, its generalized solutions usually can still be improved. For that, we come up with our third approach called `DUTY` that adds `TALE` on top of the `MARS` to further generalize the solutions obtained by `MARS`.

We have implemented our approaches in an open-source tool `HALL` (**H**aifa **All**SAT). We found that `HALL` scales substantially better than any reduction to existing AllSAT-CNF tools on our industrial STA instances as well as on various publicly available families of combinational circuits. Specifically, `MARS` is the best-performing algorithm in disjoint mode, while `DUTY` and `TALE` are the most scalable ones for industrial and generic instances, respectively, in non-disjoint mode.

The rest of this paper is organized as follows. Sect. 2 presents preliminaries. We review AllSAT in Sect. 3 and present our new algorithms in Sect. 4. Sect. 5 is dedicated to experimental evaluation. Sect. 6 is about related work, while in Sect. 7 we conclude and discuss future work.

2 Preliminaries

We begin by reviewing relevant notions from Boolean and ternary logics. We start with the standard Boolean logic syntax, which, in our context, is common to both logics. Let V be a set of variables. A *literal* l is either a variable $v \in V$, in which case l is *positive*, or a variable's negation $\neg v$, in which case l is *negative*. A formula over V under the standard Boolean logic syntax is in *Conjunctive Normal Form (CNF)* if it is a conjunction of clauses, where a *clause* is a disjunction of literals. Similarly, a formula over V is in *Disjunctive Normal Form (DNF)* if it is a disjunction of cubes, where a *cube* is a conjunction of literals. We assume naturally that no cube and no clause contains both a variable and its negation.

Assuming that circuits are represented in the standard AIGER format [3], a (combinational Boolean single-output) *circuit* Γ with n inputs and m gates is a tuple $\langle I, G, o\rangle$, where $I = \{c_1, \cdots c_n\}$ are *input elements (inputs)*, $G = \{c_{n+1}, \cdots c_{n+m}\}$ are *gate elements (gates)* and $o = c_{m+n+1}$ is a single *output element (output)*. These elements are labeled by a set of variables denoted by $\Gamma(V) = \{v_1, \ldots, v_{n+m+1}\}$. Every input element c_i is labeled by a variable v_i. Every gate element c_k is labeled by a formula $(v_k \leftrightarrow (l_i \wedge l_j))$, where $i, j < k$ and l_i, l_j are literals of variables v_i and v_j respectively. The output element o is labeled with $(v_{n+m+1} \leftrightarrow l_{n+m})$ where l_{n+m} is v_{m+n} or $\neg v_{m+n}$. For simplicity, we identify the elements with their labels (e.g. identify gate c_i as v_i, gate o with v_{n+m+1}, and so on). See Fig. 1a on page 7 for an example of a circuit. Finally, Tseitin encoding [45] generates a CNF from a given circuit Γ, by translating every gate $v \leftrightarrow l_1 \wedge l_2$ to three clauses $(v \vee \neg l_1 \vee \neg l_2) \wedge (\neg v \vee l_1) \wedge (\neg v \vee l_2)$ and adding the unit clause (o) to assert the output.

2.1 Boolean Logic Semantics

In Boolean logic, an assignment $\sigma : V \mapsto \{0, 1\}$ assigns each variable to either 0 or 1. An assignment σ is called *total* if σ is a total function and *partial* otherwise. In Boolean logic, the *cardinality* $|\sigma|$ of an assignment σ is the number of variables assigned under σ. Furthermore, in Boolean logic, we say that an assignment ρ *subsumes* the assignment σ, denoted by $\sigma \subseteq \rho$ if whenever ρ assigns a variable v, it holds that σ assigns v as well and $\rho(v) = \sigma(v)$. We then say that σ *extends* ρ. For example, $\rho \equiv \{x_1 := 1\}$ subsumes $\sigma \equiv \{x_1 := 1, x_2 := 0\}$, whereas σ extends ρ. We denote the set of solutions that extend ρ by $sol(\rho)$. An assignment σ for $V = (v_1, \dots v_n)$ is said to *satisfy* a formula $F(v_1, \dots v_n)$ in Boolean-logic syntax, if the value of $F(\sigma(v_1), \dots, \sigma(v_b))$ is 1 under the standard Boolean logic semantic convention. Specifically if F is in CNF then σ satisfies F if it satisfies at least one literal in every clause of F, and if F is in DNF then σ satisfies F if there is at least one cube in F with all its literals satisfied. A *solution* is a (partial or total) satisfying assignment.

Given an assignment σ, we define a cube D^σ as a conjunction of all positive literals v for which $\sigma(v) = 1$, and all negative literals $\neg v$ for which $\sigma(v) = 0$. Same, given a cube D, we define a (possibly partial) assignment σ^D in which $\sigma^D(v) = 1$ for every positive literal v in D and $\sigma^D(v) = 0$ for every negative literal $\neg v$ in D. We then say that σ *induces* D^σ and that D *induces* σ^D. Naturally, σ satisfies D^σ and σ^D satisfies D. We say that two formulas over the same variables, and with the same sets of solutions are *logically equivalent.*

We next define satisfying assignments for circuits. Given a circuit $\Gamma = \langle I, G, o\rangle$, and an assignment σ for $\Gamma(V)$, we say that σ satisfies a circuit element if it satisfies its label. Specifically, σ satisfies a gate $v_k \leftrightarrow l_i \wedge l_j$, if it satisfies the formula $(v_k \leftrightarrow l_i \wedge l_j)$. We say that σ satisfies the circuit Γ if σ satisfies all Γ's gates and $\sigma(o) = 1$.

One can easily show that, given a circuit $\Gamma = \langle I, G, o\rangle$ and a partial assignment $\sigma : I \mapsto \{0, 1\}$ to *all* the inputs of Γ, σ can be uniquely extended to a total assignment $\tau_\sigma : I \cup G$ that satisfies all the gates in G. Since τ_σ is fully determined by σ, we can define a solution for partial assignments over *all* the inputs only. Specifically, given a circuit $\Gamma = \langle I, G, o\rangle$ and a partial assignment to its inputs $\sigma : I \mapsto \{0, 1\}$, σ is a *solution* if and only if $\tau_\sigma(o) = 1$. Thus, for solving AllSAT-CT, it is sufficient to enumerate solutions defined only over the inputs. This observation holds also for ternary logic, presented next. We say that a circuit and a formula over the circuit's input variables are *logically equivalent* if their sets of solutions is the same. Finally, $\sigma \models T$ denotes that an assignment σ satisfies a formula or a circuit T.

2.2 Ternary Logic Semantics

Ternary logic [35] extends the semantics of Boolean logic with an additional value called *don't-care*, which we denote by X. Formally, in ternary logic, an assignment $\sigma : V \mapsto \{0, 1, X\}$ assigns each variable to one of the ternary values $\{0, 1, X\}$. To define the value of a formula F in Boolean-logic syntax under a ternary logic assignment σ, we use the ternary logic rules that extend the Boolean logic rules in which $(\neg X = X)$, $(X \wedge 1 = X)$, $(X \wedge 0 = 0)$ and $(X \wedge X = X)$ (we still have $\neg 0 = 1$, $1 \wedge 0 = 0$ and so on, as in the Boolean case). We then say that σ *satisfies* F if σ evaluates F to be 1.

We assume that all the assignments in ternary logic are total. Given a ternary assignment σ, let the *support* of σ, denoted $sup(\sigma)$ to be the set of variables v for which either $\sigma(v) = 0$ or $\sigma(v) = 1$. The *cardinality* of σ is then the size of the support of σ. We say that assignment ρ *subsumes* the assignment σ, denoted by $\sigma \subseteq \rho$, if $\sigma(v) = \rho(v)$ for every $v \in sup(\rho)$. We then say that σ *extends* ρ. For example, $\rho \equiv \{x_1 := 1, x_2 := X\}$ subsumes $\sigma \equiv \{x_1 := 1, x_2 := 0\}$, whereas σ extends ρ. Other definitions, such as logical equivalence, are identical to Boolean logic. The same goes for definitions of circuit assignments with the notable exception of how a gate can be satisfied: consider a circuit $\Gamma = \langle I, G, o \rangle$, an assignment σ and a gate v_k labeled $(v_k \leftrightarrow l_i \wedge l_j)$. Then the gate v_k is satisfied, if it either holds that:

1. $\sigma(v_k) = 1$, where $\sigma(l_i) = 1$ and $\sigma(l_j) = 1$, or
2. $\sigma(v_k) = 0$, where $\sigma(l_i) = 0$ or $\sigma(l_j) = 0$, or
3. $\sigma(v_k) = X$, where $(\sigma(l_i) = X$ and $\sigma(l_j) \neq 0)$ or $(\sigma(l_j) = X$ and $\sigma(l_i) \neq 0)$.

Note that a gate can be assigned to a don't-care value. As before, σ satisfies Γ if σ satisfies all Γ's gates and $\sigma(o) = 1$, and $\sigma \models T$ denotes that σ satisfies a CNF or a circuit T.

2.3 Solution Generalization

Solution generalization is a pivotal notion in our context. Given a solution $\sigma \models T$, where T can be a circuit or a CNF formula, any solution $\sigma' \models T$ which subsumes σ is called a *generalization*, or a *generalized solution* of σ. Since σ' can be extended not only to σ, then by generalizing σ we obtain a compact description σ' of more solutions. To explore less solutions and store them compactly, we are interested in generalizations of small cardinality.

A key observation for our context is that in Boolean logic, generalization can be carried out only by *unassigning* variables, whereas in ternary logic, generalization can be done by *reassigning* variables to X. (This is reflected in our formal definition above, since we defined subsumption differently for the two logics). The above-mentioned difference makes ternary-logic-aware generalization substantially more efficient. Indeed, as we show in Sect. 4, one can easily construct a circuit Γ and a solution σ, such that there exists a generalization of σ in ternary logic that is strictly smaller that any possible generalization in Boolean logic.

3 The AllSAT Problem

AllSAT is the problem of enumerating all the solutions for a given Boolean formula. Designing efficient AllSAT algorithms is a challenge. First, finding even a single solution is already NP-complete, hence an efficient SAT oracle is typically required. Second, since the number of possible solutions can be very large (exponential in the number of the formula's variables), a compact description of the solutions is required.

In this paper, we consider two AllSAT flavours: AllSAT-CNF and AllSAT-CT, depending on the input formula type. In AllSAT-CNF, we are given a CNF formula T, and in AllSAT-CT we are given a combinational circuit T. In both cases the solver is expected to return an enumeration of all solutions for T in a form of a DNF Q that is logically equivalent to the original T. To see why Q is indeed such an enumeration, note that since every cube D in Q induces a satisfying partial assignment σ^D, then every solution that extends σ^D also satisfies D, and therefore satisfies Q. Thus, D compactly describes the set of solutions $sol(\sigma^D)$. Then, as every solution to T satisfies at least a single cube D in Q, we have that Q serves as a compact enumeration of exactly all the solutions for T.

Out of the three main families of AllSAT approaches (blocking [28], non-blocking [14] and BDD-based [17]), mentioned in Sect. 1, we focus in this paper on the iterative *blocking* algorithm [28]. Given a CNF or a circuit, the blocking algorithm uses an incremental SAT solver [9, 34] to find solutions iteratively, where every solution is generalized, and then blocked from subsequently reappearing. This blocking is done by using a so-called *blocking clause* that prevents this solution and perhaps other solutions found so far, from being re-discovered by the SAT solver in subsequent iterations. Formally, given a CNF F and a solution $\sigma \models F$, a clause B is *blocking* if and only if $\sigma \not\models B$ and, for any solution $\tau \models F$ such that $\tau \not\subseteq \sigma$, we have $\tau \models B$ (the latter part is required to ensure correctness, that is, in order not to block solutions, not yet reported to the user). By producing generalized solutions, and adding blocking clauses to the CNF formula, the solver gradually generates all the solutions. The process terminates when the SAT solver returns *UNSAT*, which means that no further solutions can be found, indicating that all possible solutions have been enumerated.

A generic *blocking* algorithm framework for both AllSAT-CNF and AllSAT-CT, adopted from [46], is depicted in Alg. 1. Our AllSAT-CT algorithms, described in Sect. 4, follow this framework. Alg. 1 begins by encoding the circuit Γ into a CNF formula F (if required). Then, following some initialization, the algorithm runs in a loop until the current formula F_i is unsatisfiable (line 5), where, for every iteration i, F_i corresponds to the original formula F, updated with the conjunction of all the blocking clauses generated so far. Inside the loop, the algorithm first finds a solution σ_i for F_i by invoking a SAT solver (line 6). Next, it computes a generalized solution σ'_i by using the GeneralizeSol procedure on σ_i (line 7). Then, the algorithm computes the blocking clause b_i by using the ComputeBlockingCls procedure (line 8). Typically, the blocking clause is the negation of the cube $D^{\sigma'_i}$, induced by the current generalized solution σ'_i (this, however is not always the case; see Sect. 4.2.3). Afterwards, we update the DNF Q with $D^{\sigma'_i}$ and construct the next formula F_{i+1} by adding the newly generated blocking clause to F_i (line 9). Once the loop terminates, the algorithm returns the DNF Q. As one can see, many factors may impact the efficiency of a blocking AllSAT solver dramatically, including the choice of the SAT solver, the circuit encoding for AllSAT-CT, as well as solution generalization and blocking clause computation techniques.

Algorithm 1 Blocking AllSAT Algorithm Template.

```
    Input: Circuit $\Gamma$ or CNF $F$
    Output: Q in DNF with exactly the same solutions as $\Gamma$ or $F$
 1: if the input is a circuit (rather than a CNF) then
 2:     $F := \text{EncodeCircuitToCNF}(\Gamma)$      ▷ The input circuit $\Gamma$ is converted to CNF $F$
 3: end if
 4: $i := 1; F_1 := F; Q := \emptyset$
 5: while not UNSAT($F_i$) do
 6:     $\sigma_i := \text{SAT}(F_i)$                    ▷ Get the next solution $\sigma_i$
 7:     $\sigma'_i := \text{GeneralizeSol}(\sigma_i, \Gamma)$    ▷ $\sigma'_i$ generalizes $\sigma_i$; $\Gamma$ provided only if available
 8:     $b_i := \text{ComputeBlockingCls}(\sigma'_i)$       ▷ $b_i$ is disjunction of literals (clause)
 9:     $Q := Q \vee D^{\sigma'_i}, F_{i+1} := F_i \wedge b_i$      ▷ $Q$ is updated by the cube, induced by $\sigma'_i$
10:     $i := i + 1$
11: end while
12: return $Q$                                ▷ $Q$ may be disjoint or non-disjoint
```

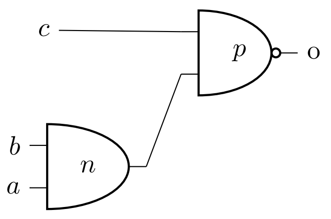
3.1 Disjoint vs. Non-Disjoint Solutions

We next discuss the concepts of disjoint and non-disjoint solutions generation, and their role in the blocking framework. Given a CNF formula or a circuit T and two solutions $\sigma \models T$ and $\tau \models T$, we say that σ and τ are *disjoint*, if there is no solution $\rho \models T$, that extends *both* σ and τ. Otherwise we say that the solutions are *non-disjoint*. Let the DNF Q be a T's AllSAT solution. Two cubes $D_i, D_j \in Q$ are *disjoint* if and only if σ^{D_i} and σ^{D_j} are disjoint. The DNF Q is *disjoint* if all its cubes are pairwise disjoint; otherwise it is *non-disjoint*. One can request an AllSAT solver to generate disjoint or non-disjoint DNFs, where both variants can be of interest, depending on the application [46]. Our industrial application STA does not require the DNF to be disjoint.

By default, the blocking framework Alg. 1 does not guarantee that the resulting DNF Q is disjoint. To produce a disjoint DNF, one can use the following observation from [46]. Let F be the CNF either given as input to Alg. 1 or encoded from a given circuit. We say that a solution τ to F is *blocking-satisfying* if τ also satisfies F_i (comprising F and all the blocking clauses obtained so far). Note that by construction, the solution σ_i, returned by the SAT solver at line 6, is blocking-satisfying, but that does not necessarily mean that the generalized solution σ' is blocking-satisfying as well. It was observed in [46] that, assuming the blocking clauses are constructed as $B := \neg D^{\sigma'_i}$, if every generalized solution σ'_i obtained in line 7 is also blocking-satisfying, then Alg. 1 is guaranteed to return a disjoint DNF. This observation is applicable to TALE, in which generalization does *not* necessarily produce blocking-satisfying solutions, and therefore TALE is not a disjoint solution algorithm. It is not applicable, however, to MARS, since MARS constructs blocking clauses differently; see Sect. 4.2.3 for more details.

4 Ternary Logic-based Algorithms for AllSAT-CT

Towards constructing algorithms that are designated for AllSAT-CT, we first observe that generalizing an existing solution under the Boolean semantics, may result in a solution with larger cardinality (thus less efficient), than when using ternary logic semantics. This is because, in Boolean logic, one cannot explicitly assign don't-care values to the circuit variables. This observation still holds when the circuit is encoded to CNF by using the Tseitin encoding or other encodings under the standard definition that maintains only the Boolean logic semantics (see, e.g. [23]). Thus, using Boolean logic semantics for encoding, may result in missing smaller generalized solutions. We support this observation by the following example.



(a) $\Gamma = \langle I = \{a, b, c\}$, $G = \{n \leftrightarrow a \wedge b, p \leftrightarrow n \wedge c\}, o \equiv \neg p\rangle$.

$$C_1 = (\neg p),$$
$$C_2 = (p \vee \neg n \vee \neg c), C_3 = (\neg p \vee n), C_4 = (\neg p \vee c),$$
$$C_5 = (n \vee \neg a \vee \neg b), C_6 = (\neg n \vee a), C_7 = (\neg n \vee b)$$

(b) Γ's Tseitin encoding into CNF $F = C_1 \wedge \ldots \wedge C_7$.

Figure 1 Illustration for Example 1.

▶ **Example 1.** Consider the circuit Γ in Fig. 1a and the solution $\sigma \equiv \{a := 1, b := 1, c := 0\}$ which satisfies Γ (recall that solutions for circuits are defined over the inputs only). Intuitively, as long as c is assigned 0, the circuit is satisfied, independently from a's and b's values. In ternary logic, σ can be generalized to a solution $\sigma'_t \equiv \{a := X, b := X, c := 0\}$ of cardinality 1 that still satisfies Γ. The corresponding generalized solution in Boolean logic could have been $\tau \equiv \{c := 0\}$. However, note that τ is *not* a solution to Γ, since the gate n cannot remain unassigned. Therefore, in Boolean logic, σ cannot be generalized to a smaller assignment, hence the generalized solution is simply $\sigma'_b = \sigma$ of cardinality 3. To emphasize this point, consider the translation of Γ to a CNF F by Tseitin encoding–in Fig. 1b. One can easily see that unassigning one of a, b or c would render F non-satisfied, since either C_6, C_7 or C_2, would cease to be satisfied, hence generalizing σ in F to a smaller solution is impossible.

To overcome the problem of inefficient generalization in Boolean logic, we propose several blocking AllSAT-CT algorithms that are ternary-logic-aware, as we present next.

4.1 TALE: the Ternary Simulation-based Algorithm

Our first algorithm TALE instantiates the blocking AllSAT algorithm Alg. 1, while borrowing a ternary simulation-based generalization procedure from [8], where it is applied at the generalization stage of the PDR model checking algorithm.

In *ternary simulation* [6, 18], a circuit $\Gamma = \langle I, G, o \rangle$ and an input assignment $\sigma : I \mapsto \{0, 1, X\}$ (which may not satisfy Γ) are given as input. The values of the gates ($v_k \leftrightarrow l_{i<k} \wedge l_{j<k}$) in G, are then computed by iteratively evaluating $l_i \wedge l_j$ based on the ternary semantics outlined in Sect. 2.2. Eventually, ternary simulation returns the evaluation of the circuit, which is the value of o, which can be either $0, 1$ or X.

Given a circuit Γ, the algorithm TALE instantiates Alg. 1 as follows. It first implements EncodeCircuitToCNF by converting Γ into a CNF F using Tseitin encoding. Then, TALE uses the standard incremental SAT-based blocking algorithm to iteratively enumerate and block solutions. The blocking clause computation procedure ComputeBlockingCls simply returns the negation of the cube $D^{\sigma'_i}$, induced by the generalized solution σ'_i. Our focus is on the solution generalization step procedure GeneralizeSol in TALE, which is carried out based on ternary simulation in the original circuit.

Algorithm 2 TALE: GeneralizeSol.

```
Input: current solution $\sigma_i$ (Boolean or ternary); the original circuit $\Gamma = \langle I, G, o \rangle$
Output: generalized solution $\sigma'_i$ (in ternary logic) over the circuit inputs
1: $v \in I$: if $v$ is assigned under $\sigma$ then $\sigma'_i(v) := \sigma_i(v)$ else $\sigma'_i(v) := X$    ▷ Initialize $\sigma'_i$
2: for $v \in I; \sigma'_i(v) \neq X$ do    ▷ $v$ is a circuit input
3:     $\sigma'_i(v) := X$
4:     Simulate $\sigma'_i$ with ternary-simulation on $\Gamma$ to get an evaluation $eval(\Gamma)$
5:     if $eval(\Gamma) = X$ then
6:         $\sigma'_i(v) := \sigma_i(v)$    ▷ Ternary simulation failed, return the original value of $v$
7:     end if
8: end for
9: return $\sigma'_i$
```

In detail, GeneralizeSol receives the current solution σ_i (σ_i is, originally, in Boolean logic, but Alg. 2 interprets it in ternary logic) and the original circuit Γ and returns a solution $\sigma'_i \models \Gamma$, in ternary logic, that generalizes σ_i. First, Alg. 2 initializes σ'_i from σ_i by setting every variable unassigned in σ_i, to X (line 1). Then, the algorithm iterates through all Γ's inputs (line 2), and, for every input v where $\sigma'_i(v)$ is not X, it tentatively assigns X to v in σ'_i (line 3). Alg. 2 then simulates the updated σ'_i on the circuit C by using ternary simulation (line 4). If the simulation renders the output X, then v cannot be converted to a don't-care, thus the algorithm restores the original value of v, that is $\sigma'_i(v)$ (lines 5–6). Otherwise, v remains X in σ'_i. In the end, the algorithm returns the generalized solution σ'_i (line 9).

TALE does not guarantee that the solutions are disjoint, since while generalization guarantees that the resulting solution σ' satisfies the original circuit (thus, the original CNF), it does not guarantee that the blocking clauses in F_i are satisfied (being unaware of them). Hence, σ' is not necessarily blocking-satisfying (recall Sect. 3.1).

4.2 MARS: the Dual-Rail&MaxSAT-based Algorithm

In this section, we introduce our second algorithm, called MARS. Similarly to TALE, MARS fits into the framework of the blocking algorithm Alg. 1. However, instead of applying a dedicated GeneralizeSol procedure to generalize solutions, it relies on the SAT solver to return an already reasonably generalized solution (hence, `MARS` implements GeneralizeSol by simply returning $\sigma' \equiv \sigma$). This is achieved by using the so-called dual-rail encoding [5, 21] to convert the circuit to CNF and applying anytime MaxSAT-inspired heuristics [31, 32] in the underlying SAT solver to heuristically generalize solutions. Here, we introduce two new blocking clause generation algorithms for the dual-rail encoding, designed to have `MARS` return disjoint or non-disjoint solutions, respectively.

Applying the combination of dual-rail encoding and full-blown MaxSAT solving for generalization was proposed in [40], while [38, 10] had previously introduced a closely related method of applying, to the same end, a single SAT invocation that assigns all the decision variables to 0. The latter approach had been proposed in the context of abstraction refinement [38] and minimal model generation for SMT [10], but was also evaluated in the context of PDR in [40]. However, in these works, the main flow creates a separate dual-rail encoded SAT instance for generalization only, while any blocking clauses are created using the standard Tseitin encoding-based technique and added to a Tseitin-encoded CNF instance, maintained by the main flow. Thus, our approach of using a single dual-rail-encoded instance throughout the whole flow, combined with a MaxSAT approximation for generalization and blocking clause generation native to dual-rail encoding, is novel (in addition, our application and the high-level algorithm are completely different from those in [40, 38, 10]). Moreover, our approach of making an AllSAT algorithm return non-disjoint or disjoint solutions, based on different blocking clause generation schemes, is also new.

In what follows, we first describe the dual-rail encoding, followed by the use of a MaxSAT approximation in the generalization process and our blocking clause generation algorithms.

4.2.1 The Dual-Rail Encoding

In dual-rail encoding [5], every variable v in the set of variables $\Gamma(V)$ in a given circuit $\Gamma = \langle I, G, o \rangle$, is mapped to two Boolean *dual-rail* variables (v^+, v^-). This results in a set of variables U that we use to encode the circuit Γ. The dual-rail encoding induces the following one-to-one mapping between a Boolean assignment σ over the dual-rail variables U and a ternary assignment σ over circuit's original variables V (slightly abusing the notation, we reuse σ for both assignments):

1. $\sigma(v) = 1 \iff (\sigma(v^+) = 1 \text{ and } \sigma(v^-) = 0)$
2. $\sigma(v) = 0 \iff (\sigma(v^+) = 0 \text{ and } \sigma(v^-) = 1)$
3. $\sigma(v) = X \iff (\sigma(v^+) = 0 \text{ and } \sigma(v^-) = 0)$
4. the combination $\sigma(v^+) = \sigma(v^-) = 1$ is disallowed.

We now describe the encoding. For a negative literal $l = \neg v$, we use the following notation: $l^+ = v^-$ and $l^- = v^+$. Then, to convert the circuit Γ to CNF, the dual-rail encoding generates the following clauses:

1. For every $v \in V$, we generate the clause $(\neg v^+ \vee \neg v^-)$ to block the disallowed combination.
2. To translate every gate $v \equiv l_i \wedge l_j$, we generate the clauses: $(v^+ \vee \neg l_i^+ \vee \neg l_j^+) \wedge (\neg v^+ \vee l_i^+) \wedge (\neg v^+ \vee l_j^+) \wedge (\neg v^- \vee l_i^- \vee l_j^-) \wedge (v^- \vee \neg l_i^-) \wedge (v^- \vee \neg l_j^-)$.
3. Finally, we generate the unit clause (o^+) to assert the output o.

Note that the dual-rail encoding is heavier than the Tseitin encoding, since it generates twice as many Boolean variables and many more clauses. To summarize, our algorithm `MARS` implements EncodeCircuitToCNF by using the dual-rail encoding.

4.2.2 MaxSAT Approximation for On-the-Fly Generalization

MaxSAT is a widely used extension of SAT to optimize a linear Pseudo-Boolean function [2]. Given a CNF formula F and a *target bit-vector (target)* $T = \{t_n, t_{n-1}, \ldots, t_1\}$, where each *target bit* t_i is a Boolean variable associated with a strictly positive integer weight w_i, MaxSAT finds a model σ to F that minimizes the following objective function: $\Psi(\sigma) = \sum_{i=1}^{n} \sigma(t_i) \times w_i$. A MaxSAT instance is *unweighted* if and only if all the weights are 1; otherwise it is *weighted*.

It was observed in [40] that, given a solution σ to a circuit $\Gamma = \langle I, G, o \rangle$, one can find a *minimal* generalized solution σ' by the following reduction to unweighted MaxSAT. First, Γ is converted into a CNF formula F using the dual-rail encoding. Next, given $I = \left\{v_1, \ldots, v_{|I|}\right\}$, the vector target T is defined to contain both the dual-rail variables for every input of the original circuit: $T = \left\{v_1^+, v_1^- \ldots, v_{|I|}^+, v_{|I|}^-\right\}$. By using this reduction, invoking a MaxSAT solver over F and T guarantees that the resulting solution, when restricted to the circuit inputs, is of a minimal (but not necessarily *minimum*) cardinality. This is because, by definition, the solver maximizes the number of the Boolean dual-rail variables assigned to 0. Then, since we use clauses to block the assignment $(v^+ := 1, v^- := 1)$ for every pair of the dual-rail variables, we have that every solution σ must assign at least one of the dual-rail variables in every pair to 0. Hence, the minimal solution to F maximizes the number of pairs of the dual-rail variables, in which both variables are assigned to 0, therefore maximizing the number of inputs in the original circuit that are assigned to X.

Since, according to our preliminary experiments, obtaining an optimal MaxSAT solution is computationally expensive, in practice our algorithm `MARS` does not apply full-scale MaxSAT solver to generalize the solutions, but rather uses a standard incremental SAT solver, augmented with two heuristics, applied by anytime MaxSAT solvers [31, 32] to heuristically minimize the solution. Specifically, in the beginning of the search, we boost the score of the (would-be) target variables, which describe the dual-rail variables for the circuit inputs (corresponding to the TSB heuristic in [31, 32]). Additionally, whenever a target variable is chosen by the solver's decision heuristic, we assign it to 0 first (corresponding to the optimistic polarity selection heuristic in [31, 32]). These techniques increase the likelihood of generating heuristically generalized solutions.

4.2.3 Generating Blocking Clauses in `MARS`

Fitting into the blocking algorithm Alg. 1, `MARS` uses the dual-rail encoding to encode the given circuit $\Gamma = \langle I, G, o \rangle$ to a CNF formula F, then trusts the SAT solver, described in Sect. 4.2.2, (invoked at line 6 of Alg. 1) to return an assignment σ for F that already heuristically serves as a generalized solution to Γ (whereas GeneralizeSol, invoked next, simply returns σ). Since the semantics of the dual-rail encoding are ternary-based, we have that σ also assigns every circuit input v to 0, 1 or X, as described in Sect. 4.2.1. We finally explain how `MARS` constructs the blocking clauses, that is, how `MARS` implements ComputeBlockingCls. We also show how with this construction, one can make the distinction between the disjoint and non-disjoint modes, which renders Alg. 1 to return a disjoint or a non-disjoint DNF, respectively.

Towards implementing ComputeBlockingCls, we make use of the observation that it is sufficient for the blocking clause B that we construct to force a change in a value, assigned to at least one of the inputs in $sup(\sigma)$, in order to block the SAT solver from finding σ again. In details, for the disjoint case, B needs to force at least one input $v \in sup(\sigma)$ to *flip* to $\neg\sigma(v)$ (that is, either 0 or 1), where merely changing v to X is not enough. In that way, every future solution ρ, found by the solver, will have at least one input in both $sup(\sigma)$ and

$sup(\rho)$ that has different values. Thus, every solution that extends σ will be different from a solution that extends ρ, as required in the disjoint case. For the non-disjoint case, however, it is sufficient to force at least one input $v \in sup(\sigma)$ to *change* (to either $\neg\sigma(v) \in \{0,1\}$ or X). This will guarantee that every future solution ρ will have at least one input from $sup(\sigma)$ assigned differently than under σ, but altogether σ and ρ can still be non-disjoint; for example, ρ might subsume σ.

We are now ready to present our core algorithms for blocking clause generation for both modes. Observe that for every input v in $sup(\sigma)$, one of the dual-rail variables (v^+, v^-) must be satisfied by σ, while the other one must be falsified (since $(1,1)$ is disallowed).

In the disjoint mode, we set B to be the disjunction of all the dual-rail variables of the form v^ϵ, where $\epsilon \in \{+,-\}$ for which $v \in sup(\sigma)$ and $\sigma(v^\epsilon) = 0$. To see why this works, assume, without loss of generality, that σ assigns some input $v \in sup(\sigma)$ to 1, that is, it assigns (v^+, v^-) to $(1,0)$. Then, B forces every subsequent solution, in which every other input $u \neq v \in sup(\sigma)$ is *not* flipped (that is, u retains the same value as in σ or is assigned X), to assign (v^+, v^-) to $(0,1)$, thus flipping v from 1 to 0.

In the non-disjoint mode, we set B to the disjunction of negative literals of the dual-rail variables of the form $\neg v^\epsilon$, where $\epsilon \in \{+,-\}$ for which $v \in sup(\sigma)$ and $\sigma(v^\epsilon) = 1$. Again to see why this works, assume, without loss of generality, that σ assigns some input v to 1, that is, it assigns (v^+, v^-) to $(1,0)$. Then, B forces every subsequent solution, in which all the other inputs $u \neq v \in sup(\sigma)$ are *unchanged* as compared to σ, to assign (v^+, v^-) to either $(0,1)$ or $(0,0)$, thus changing v by either flipping v to 0 or assigning it X.

Observe that, by construction, any solution τ which is not subsumed by σ is guaranteed to satisfy our blocking clause in both modes. To better understand the difference between the non-disjoint and disjoint modes, consider the following example.

▶ **Example 2.** Consider the circuit Γ in Fig. 1a and the solution $\sigma \equiv \{a := X, b := X, c := 0\}$. Recall that, if $\sigma(c) = 0$, then $\sigma(c^+) = 0$ and $\sigma(c^-) = 1$. In the non-disjoint mode, the generated blocking clause would be $B = (\neg c^-)$. Adding the clause B, allows for the following solution $\tau \equiv \{a := 0, b := 0, c := X\}$, where the solution $\rho \equiv \{a := 0, b := 0, c := 0\}$ is subsumed by both σ and τ, thus result in non-disjoint solutions. On the other hand, in the disjoint mode, the generated blocking clause would be $B = (c^+)$, enforcing c to be assigned 1 in every subsequent solution. Adding the clause B would render $\tau \equiv \{a := 0, b := 0, c := 1\}$, disjoint from σ, the only remaining solution.

4.3 DUTY: Combining MARS and TALE

We finally describe our third algorithm `DUTY`, which is similar to `MARS`, except for invoking the ternary simulation-based generalization method of `TALE` at line 7 in Alg. 1. Such a combination makes sense because of the heuristic nature of the on-the-fly generalization carried out by the SAT solver in `MARS`. Note that `DUTY` does not necessarily generate disjoint solutions, since ternary simulation-based generalization does not guarantee that the blocking clauses are satisfied. One may expect `DUTY` to outperform plain `MARS`, because ternary simulation over an *almost* minimized solution is expected to be cheap and, hopefully, efficient. It is unclear, however, how `DUTY` compares to `TALE`. On one hand, `DUTY` incurs the overhead of using the dual-rail encoding which generates more clauses and variables than Tseitin encoding. On the other hand, `DUTY` carries out generalization on-the-fly during SAT solving that leaves substantially less work to ternary simulation.

Another important detail is that, in `DUTY`, `MARS`'s blocking clause generation is applied in disjoint mode because of the substantially better performance than the non-disjoint mode (inside `DUTY`) in our preliminary experiments. Informally, sharing too many subsumed solutions slows down the algorithm as it has to enumerate more generalized solutions.

5 Experimental Results

We implemented our algorithms in a new open-source tool `HALL`[2]. Then, we compared these algorithms within `HALL` with the AllSAT-CNF `Toda` tools [44] on our own industrial STA benchmarks (made publicly available), as well as on circuits from the EPFL combinational benchmark suite [1] and random circuits. We carried out the comparison for both the disjoint and non-disjoint cases. In our experiments, we evaluated two criteria: the runtime and the size, where *size* refers to the number of cubes in the resulting DNF.

5.1 Evaluation Setup

`HALL` is written in C++20 on top of `Intel® SAT Solver` [33]. All the experiments were run on machines with 32Gb of memory with Intel® Xeon® processors with 3Ghz CPU frequency. We set the timeout to 3600 seconds.

Since our work considers circuits with one output, we transformed any multi-output circuits into one-output circuit by applying, over all the outputs, either *or* (disjunction) operator (using the utility *aigor* from the *aiger* library [3][3]) or *xor* operator (using our own *aigxor* utility[4]). We omit the conversion time in the results as it is negligible. Furthermore, to evaluate our AllSAT-CT tool against AllSAT-CNF solvers, we translated each circuit from the AIGER format [3] to CNF using the *aigtocnf* utility[5] from the *aiger* library.

5.1.1 Benchmarks

We used the following three benchmark sets:

- **sta_gen** – Static Timing Analysis (STA) industrial set: we generated the following parametrized benchmark family, which encapsulates a variety of real-world STA instances we had encountered. Given the number of inputs N, each formula $F(N)$ consists of a disjunction of subformulas $F_1(N)$ and $F_2(N)$, where each subformula comprises a DNF, conjuncted with the selector v_N or $\neg v_N$. All the cubes in the DNFs have two variables and are pairwise disjoint. The resulting formula looks as follows, where $j = (N-1)/2$:

$$\begin{aligned} F_1(N) &:= ((v_1 \wedge v_2) \vee \ldots \vee (v_{j-1} \wedge v_j)) \wedge v_N \\ F_2(N) &:= ((v_{j+1} \wedge v_{j+2}) \vee \ldots \vee (v_{(N-2} \wedge v_{N-1})) \wedge \neg v_N \\ F(N) &:= F_1(N) \vee F_2(N) \end{aligned}$$

  To satisfy $F(N)$ with the minimal possible solution σ, it is sufficient to satisfy a pair of consecutive variables and assign the selector so as to satisfy its subformula (e.g., assign $\sigma(v_1) = \sigma(v_2) = 1$ and $\sigma(v_N) = 1$). Note that, for $F(N)$ to be well-defined, N must be odd and $N-1$ divisible by 4. While these formulas may be easy to interpret for humans, we note that they are challenging for the automatic solvers that we inspected.
- EPFL combinational benchmark suite [1]: we used the arithmetic and the random_control sets. We created two instances of each set depending on whether the multiple outputs are combined using the operator *or* or the operator *xor*, resulting in the following four sets: **arithmetic_or**, **arithmetic_xor**, **random_control_or**, **random_control_xor**.

[2] `HALL` and all the benchmarks are available at `https://github.com/yogevshalmon/allsat-circuits`.
[3] The library is available at `https://github.com/arminbiere/aiger`.
[4] *aigxor* is available at `https://github.com/yogevshalmon/aiger`.
[5] We used *aigtocnf* with the **–no-pg** option to enforce the translation to preserve the number of solutions.

- Random combinational circuit: we used the *aigfuzz*[6] utility from the AIGER library [3] for generating large combinational circuits. We report results only for the family **large_cir_or**, where the outputs are combined by the operator *or*, since using the operator *xor* resulted in benchmarks which proved to be too difficult for all the solvers.

5.1.2 Solvers

We compared our tool `HALL` against the three solvers from the `Toda` repository [44]: `BC`, `NBC` and `BDD`. We tried to get access to the more recent solvers BASolver [47] and AllSATCC [25] but, unfortunately, these tools are not available online, and we could not reach the authors. Below, we list all the solvers and algorithms that we have used, separated by whether they are guaranteed to return only disjoint solutions:

- Disjoint solutions guaranteed:
  - `NBC` [44]: a non-blocking AllSAT-CNF solver.
  - `BDD` [44]: a BDD-based AllSAT-CNF solver.
  - `MARS`: our tool `HALL` with `MARS` in disjoint mode.
- Non-Disjoint (that is, disjoint solutions *not* guaranteed):
  - `BC` [44]: blocking clause-based AllSAT-CNF solver. We used the *SIMPLIFY* and *NONDISJOINT* macros to make the solver return (non-disjoint) partial solutions.
  - `TALE`: our tool `HALL` with `TALE`.
  - `MARS`: our tool `HALL` with `MARS` in non-disjoint mode.
  - `DUTY`: our tool `HALL` with `DUTY`.

5.2 Evaluation of the STA Benchmark Sets

In our first experiment, we used our industrial **sta_gen** benchmark set. We separate our analysis to disjoint and non-disjoint solvers.

5.2.1 Results for Disjoint Solvers

Table 1 Comparing disjoint solvers on the **sta_gen** benchmark family. The first column (N) specifies the number of inputs for each instance. Each pair of columns (T,S) shows the run-time in seconds (where TO stands to a time-out, MO stands for a memory-out and < 1 for a run-time lower than 1 second), and the size of the DNF in the number of cubes.

N	*Disjoint*					
	`MARS`		`NBC`		`BDD`	
	T (sec)	**S**	**T** (sec)	**S**	**T** (sec)	**S**
9	< 1	10	< 1	224	< 1	224
17	< 1	59	< 1	89600	< 1	89600
25	**< 1**	253	2.111	27582464	48.26	27582464
33	**< 1**	1315	570.897	7729971200	MO	–
37	**9.808**	59538	TO	–	MO	–
41	TO	–	TO	–	MO	–

[6] We used the following parameters for aigfuzz: **"-a -c -l -1"**.

The comparison between the disjoint solvers is shown in Table 1. One can see that our MARS algorithm is substantially more scalable than the others, but even MARS could only handle small formulas with up to 37 inputs. This is because **sta_gen** is a very hard problem in the disjoint mode, since the structure of the formula implies that the number of total solutions is exponential. Consequently, NBC and BDD that do not iterate over partial solutions, thus return all the total solutions, struggle to scale. MARS, which can return partial solutions, scales somewhat better.

5.2.2 Results for Non-Disjoint Solvers

Table 2 shows our results for the non-disjoint solvers comparison. Notably, our tool HALL, in all its variants, substantially outperforms BC, because of the limitations of generalization in Boolean logic as compared to ternary logic, highlighted in our paper.

Observe also that, for the non-disjoint case, HALL was able to solve instances with over 10000 inputs as compared to 37 inputs only for the disjoint case. This is because, for the non-disjoint case (which is the one required in our industrial usage), solving **sta_gen** becomes substantially simpler. Specifically, for every benchmark with N inputs there exists the following DNF solution with $(N-1)/2$ cubes: $Q = (v_1 \land v_2 \land v_N) \lor \ldots \lor (v_{j-1} \land v_j \land v_N) \lor \ldots \lor (v_{j+1} \land v_{j+2} \land \neg v_N) \lor \ldots \lor (v_{N-2} \land v_{N-1} \land \neg v_N)$.

Table 2 Comparing non-disjoint solvers on **sta_gen**. The first column (N) shows the number of inputs for each instance (inputs from 33 to 2009 are skipped, because of similarity of the results). Each subsequent pair of columns (T,S) shows the solver's run-time in seconds and the size of the resulting DNF in number of cubes.

N	***Non-Disjoint***							
	TALE		MARS		DUTY		BC	
	T (sec)	**S**	**T** (sec)	**S**	**T** (sec)	**S**	**T** (sec)	**S**
9	< 1	4	< 1	4	< 1	4	< 1	90
17	< 1	8	< 1	9	< 1	8	< 1	10538
25	< 1	12	< 1	12	< 1	12	866.53	1026771
33	< 1	16	< 1	20	< 1	16	TO	–
2009	< 1	1004	< 1	1154	< 1	1004	TO	–
3009	$\mathbf{< 1}$	1504	1.874	1990	$\mathbf{< 1}$	1504	TO	–
4009	1.676	2004	2.917	3036	**1.374**	2004	TO	–
5009	3.349	2504	9.83	3629	**2.417**	2504	TO	–
6009	4.403	3004	40.114	20530	**4.181**	3004	TO	–
7009	7.886	3504	10.294	6098	**4.625**	3504	TO	–
9009	28.909	4504	84.817	6889	**11.977**	4504	TO	–
11009	56.877	5504	TO	–	**23.0**	5504	TO	–
13009	96.688	6504	50.361	13704	**29.889**	6504	TO	–

Comparing our algorithms, one can see that DUTY outperforms both TALE and MARS. Notably, TALE and DUTY return DNFs of the same size, which is the optimal $(N-1)/2$, while MARS returns larger DNFs. Apparently, ternary simulation, applied by both TALE and DUTY (but not by MARS), was essential to reduce the size of every cube and also the resulting DNF.

5.3 Evaluation of the EPFL and Random Benchmark Sets

Table 3 summarizes the evaluation results on the EPFL and random families. Specifically, it shows the number of solved instances per each family and solver, where an instance is considered solved, if the solver completed the enumeration all its solutions within the timeout.

Overall, our tool `HALL` solved significantly more instances than the others, where its variant `TALE` is the winner amongst the non-disjoint solver, while `MARS` is the winner amongst the disjoint ones. More specifically, `HALL` is substantially more efficient on instances, where the outputs are joined using the operator *or*. This is because, when the operator *or* is applied over the outputs, the solver can set all the outputs, except for one, to a don't-care. Our dedicated ternary logic-aware algorithms take full take advantage of this property, while other solvers, restricted to Boolean logic, fail to do so. On families, where the outputs are joined using the operator *xor*, the difference is not that significant, where there is one benchmark from the **arithmetic_xor** family, which was solved only by `NBC`.

Table 3 Comparing the number of instances solved from each benchmark set and overall. The first column (Family) shows the benchmark family name. The two subsequent columns provide the number of benchmarks (#Bench) and the average number of inputs for each set (AvgIN). Each of the subsequent columns shows the number of instances solved for the corresponding solver.

Family	**#Bench**	**AvgIN**	***Non-Disjoint***				***Disjoint***		
			`TALE`	`MARS`	`DUTY`	`BC`	`MARS`	`NBC`	`BDD`
arithmetic_or	10	166	**4**	2	3	0	1	1	0
arithmetic_xor	10	166	0	0	0	0	0	**1**	0
random_control_or	10	283	**9**	**9**	**9**	4	7	4	4
random_control_xor	10	283	**5**	**5**	**5**	4	4	4	4
large_cir_or	20	597	**16**	7	**16**	0	7	0	0
Total	60	–	**34**	23	33	8	19	10	8

6 Related Work

AllSAT research has so far been mostly focused on the well studied problem of AllSAT-CNF that is, enumerating all the satisfying assignments of a Boolean formula in CNF [30, 46, 44, 14, 24, 11]; see [44] for an extensive survey. Recent progress in AllSAT-CNF includes [47] that relies on finding backbone variables, and [25] that uses efficient component analysis.

To the best of our knowledge, the only papers that explicitly consider AllSAT-CT are [19, 20, 43]. The first two papers introduce blocking non-disjoint AllSAT-CT algorithms, which utilize a hybrid SAT solver that can carry out conflict analysis and propagation in both CNF formulas and circuits, where the target application is unbounded model checking. The third paper enhances the blocking algorithm with structural analysis. All these algorithms are restricted to Boolean logic in contrast to our ternary logic-aware algorithms. Additionally, the resulting tools are currently unavailable.

A number of solution generalization methods have been proposed in the context of the PDR algorithm for model checking [16, 15, 40, 8, 13]; see [40] for an extensive survey. Finding minimal test cubes in circuit testing is closely related to solution generalization in PDR, where [39, 37] investigate different techniques, including MaxSAT- and MaxQBF-based.

Finally, another closely related problem is one of generating of all the prime implicants, given a Boolean formula, studied in [41, 36, 27]. Informally, in our terminology, an implicant is a cube which implies the original formula (that is, an implicant is a not-necessarily-generalized solution). A prime implicant is a generalized cube, which cannot be generalized further. AllSAT can then be viewed as the problem of finding a (not-necessarily-prime) implicant cover, which is a significantly simpler problem than that of generating all the prime implicants.

7 Conclusion & Future Work

Motivated by the need to improve the scalability of Intel's in-house Static Timing Analysis (STA) tool, we considered the problem of enumerating all the solutions of a single-output combinational Boolean circuit, called AllSAT-CT. We introduced several dedicated ternary logic-based AllSAT-CT algorithms and implemented them in an open-source tool called `HALL`. Our experimental results demonstrated that `HALL` scales substantially better than any reduction to existing AllSAT-CNF tools on our industrial STA instances as well as on various publicly available families of combinational circuits.

For future work, we plan to investigate how to utilize other AllSAT-CNF methods, including the nonblocking technique that modifies the SAT solver to enumerate the solutions explicitly without blocking clauses, for AllSAT-CT. Furthermore, we plan to explore solutions to related problems to utilize the relevant techniques for AllSAT-CT, including dual value propagation in circuits, which is a known method in QBF solving [12] and projected model counting [29]. Specifically, very recently, we became aware of a tool called *dualiza* [29], that, in addition to its main capability of projected model counting, is also capable to enumerate solutions for circuits in the AIGER format. The tool is based on dual calculus. An initial study of ours already suggests promising methods of integrating duality considerations into our algorithms, which we plan to explore.

References

1 Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL combinational benchmark suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.

2 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In *Handbook of Satisfiability – Second Edition*, pages 929–991. IOS Press, 2021.

3 Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

4 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

5 Randal E. Bryant, Derek L. Beatty, Karl S. Brace, Kyeongsoon Cho, and Thomas J. Sheffler. COSMOS: A compiled simulator for MOS circuits. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 9–16, 1987.

6 Randall E Bryant. Race detection in mos circuits by ternary simulation, 1983.

7 Imen Ouled Dlala, Saïd Jabbour, Lakhdar Saïs, and Boutheina Ben Yaghlane. A comparative study of SAT-based itemsets mining. In *Research and Development in Intelligent Systems XXXIII – Incorporating Applications and Innovations in Intelligent Systems XXIV. Proceedings of AI-2016.*, pages 37–52, 2016.

8 Niklas Eén, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134, 2011.

9 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT, Proceedings*, 2003.

10 Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, Italy, 2010.

11 Malay K Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In *IEEE/ACM International Conference on Computer Aided Design,ICCAD.*, pages 510–517, 2004.

12 Alexandra Goultiaeva and Fahiem Bacchus. Exploiting QBF duality on a circuit representation. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1791`.

13 Alberto Griggio and Marco Roveri. Comparing different variants of the ic3 algorithm for hardware model checking. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 35(6):1026–1039, 2016. `doi:10.1109/TCAD.2015.2481869`.

14 Orna Grumberg, Assaf Schuster, and Avi Yadgar. Memory efficient all-solutions SAT solver and its application for reachability analysis. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD, Proceedings*, 2004.

15 Arie Gurfinkel and Alexander Ivrii. Pushing to the top. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 65–72, 2015.

16 Zyad Hassan, Aaron R Bradley, and Fabio Somenzi. Better generalization in ic3. In *2013 Formal Methods in Computer-Aided Design*, pages 157–164, 2013.

17 Jinbo Huang and Adnan Darwiche. The language of search. *J. Artif. Intell. Res.*, 29:191–219, 2007.

18 James S. Jephson, Robert P. McQuarrie, and Robert E. Vogelsberg. A three-value computer design verification system. *IBM Systems Journal*, 8(3):178–188, 1969.

19 HoonSang Jin, HyoJung Han, and Fabio Somenzi. Efficient conflict analysis for finding all satisfying assignments of a boolean circuit. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS, Proceedings*, 2005. `doi:10.1007/978-3-540-31980-1_19`.

20 HoonSang Jin and Fabio Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. In *Proceedings of the 42nd Design Automation Conference, DAC*, 2005.

21 Daher Kaiss, Marcelo Skaba, Ziyad Hanna, and Zurab Khasidashvili. Industrial strength SAT-based alignability algorithm for hardware equivalence verification. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 20–26, 2007.

22 Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT*, 2003.

23 Petr Kucera and Petr Savický. Backdoor decomposable monotone circuits and their propagation complete encodings. *CoRR*, abs/1811.09435, 2018. `arXiv:1811.09435`.

24 Bin Li, Michael S. Hsiao, and Shuo Sheng. A novel SAT all-solutions solver for efficient preimage computation. In *2004 Design, Automation and Test in Europe Conference and Exposition*, pages 272–279, 2004.

25 Jiaxin Liang, Feifei Ma, Junping Zhou, and Minghao Yin. Allsatcc: Boosting AllSAT solving with efficient component analysis. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI*, 2022.

26 Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, and George Varghese. Network verification in the light of program verification. *MSR, Rep*, 2013.

27 Weilin Luo, Hai Wan, Hongzhen Zhong, Ou Wei, Biqing Fang, and Xiaotong Song. An efficient two-phase method for prime compilation of non-clausal boolean formulae. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021*, pages 1–9. IEEE, 2021. `doi:10.1109/ICCAD51958.2021.9643520`.

28 Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification, 14th International Conference, CAV, Proceedings*, 2002.

29 Sibylle Möhle and Armin Biere. Dualizing projected model counting. In *IEEE 30th International Conference on Tools with Artificial Intelligence, ICTAI*, pages 702–709, 2018. doi:10.1109/ICTAI.2018.00111.

30 A. Morgado and J. Marques-Silva. Good learning and implicit model enumeration. *Proceedings – International Conference on Tools with Artificial Intelligence, ICTAI*, pages 131–136, 2005.

31 Alexander Nadel. Anytime weighted MaxSAT with improved polarity selection and bit-vector optimization. In *Formal Methods in Computer Aided Design, FMCAD, Proceedings*, pages 193–202, 2019.

32 Alexander Nadel. Polarity and variable selection heuristics for SAT-based anytime MaxSAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2020.

33 Alexander Nadel. Introducing intel(r) SAT solver. In *25th International Conference on Theory and Applications of Satisfiability Testing, SAT*, 2022.

34 Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In *Theory and Applications of Satisfiability Testing – SAT, Proceedings*, 2012.

35 Emil L. Post. Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, 43, 1921.

36 Alessandro Previti, Alexey Ignatiev, António Morgado, and João Marques-Silva. Prime compilation of non-clausal formulae. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 1980–1988. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/281.

37 Sven Reimer, Matthias Sauer, Paolo Marin, and Bernd Becker. QBF with soft variables. *Electronic Communications of the EASST*, 70, 2014.

38 Jan-Willem Roorda and Koen Claessen. SAT-based assistance in abstraction refinement for symbolic trajectory evaluation. In *Computer Aided Verification, 18th International Conference, CAV, Proceedings*, 2006.

39 Matthias Sauer, Sven Reimer, Ilia Polian, Tobias Schubert, and Bernd Becker. Provably optimal test cube generation using quantified boolean formula solving. In *18th Asia and South Pacific Design Automation Conference, ASP-DAC*, 2013.

40 Tobias Seufert, Felix Winterer, Christoph Scholl, Karsten Scheibler, Tobias Paxian, and Bernd Becker. Everything you always wanted to know about generalization of proof obligations in PDR. under submission. *CoRR*, abs/2105.09169, 2021. arXiv:2105.09169.

41 James R. Slagle, Chin-Liang Chang, and Richard C. T. Lee. A new algorithm for generating prime implicants. *IEEE Trans. Computers*, 1970. doi:10.1109/T-C.1970.222917.

42 Robert B. Hitchcock Sr. Timing verification and the timing analysis program. In *Proceedings of the 19th Design Automation Conference, DAC*, 1982.

43 Abraham Temesgen Tibebu and Görschwin Fey. Augmenting all solution SAT solving for circuits with structural information. In *21st IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS*, pages 117–122. IEEE, 2018. doi:10.1109/DDECS.2018.00028.

44 Takahisa Toda and Takehide Soh. Implementing efficient all solutions SAT solvers. *Journal of Experimental Algorithmics (JEA)*, 2016.

45 Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.

46 Yinlei Yu, Pramod Subramanyan, Nestan Tsiskaridze, and Sharad Malik. All-SAT using minimal blocking clauses. *Proceedings of the IEEE International Conference on VLSI Design*, pages 86–91, 2014.

47 Yueling Zhang, Geguang Pu, and Jun Sun. Accelerating All-SAT computation with short blocking clauses. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE*, 2020.