Enumerating All Boolean Matches

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

Yogev Shalmon
Intel Corporation, Haifa, Israel
Faculty of Data and Decision Sciences, Technion, Haifa, Israel

Abstract

Boolean matching, a fundamental problem in circuit design, determines whether two Boolean circuits are equivalent under input/output permutations and negations. While most works focus on finding a single match or proving its absence, the problem of enumerating all matches remains largely unexplored, with `BooM` being a notable exception. Motivated by timing challenges in Intel's library mapping flow, we introduce `EBat`– an open-source tool for enumerating all matches between single-output circuits. Built from scratch, `EBat` reuses `BooM`'s SAT encoding and introduces novel high-level algorithms and performance-critical subroutines to efficiently identify and block multiple mismatches and matches simultaneously. Experiments demonstrate that `EBat` substantially outperforms `BooM`'s baseline algorithm, solving 3 to 4 times more benchmarks within a given time limit. `EBat` has been productized as part of Intel's library mapping flow, effectively addressing the timing challenges.

2012 ACM Subject Classification Mathematics of computing → Solvers

Keywords and phrases Boolean Matching, All-Boolean-Matching, Enumeration, SAT, Generalization

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.22

Supplementary Material *Software*: `https://github.com/yogevshalmon/ebat`
archived at `swh:1:dir:4b747bd04b637c1e3c938cea66469b2ff0b5d2d9`

Acknowledgements We are grateful to Oded Asulin, Yossi Levani, Aviad Munitz, Pavel Nisanov and Hagay Segal for helpful discussions, which played an important role in shaping our research. We also thank Roland Jiang for providing us with the original code for `BooM`. Additionally, we would like to thank the anonymous reviewers for their valuable comments and insightful suggestions.

1 Introduction

Boolean matching is a pivotal problem of determining whether two Boolean circuits are equivalent under the permutation and negation of inputs and outputs. From the theoretical standpoint, Boolean matching lies between coNP and Σ_2^P [3, 10], which makes it a good candidate for examining the open question about the collapse of the polynomial hierarchy. On the practical front, Boolean matching is widely applied, including in library mapping (also known as library binding or technology mapping) [6, 28], synthesis [16, 43], Engineering Change Order (ECO) [22], equivalence checking [29] and protection against hardware Trojans [40]. Considerable attention has been devoted to solving Boolean matching, highlighted by its inclusion as a CAD contest problem at ICCAD'23 [15]. Existing solving methods can be categorized into canonical-form- [6, 29, 23], signature- [44, 1], and SAT-based [42, 26, 25]. While almost all existing works focus on finding a *single* match or proving its nonexistence, this paper is dedicated to the *all-Boolean-matching* problem of enumerating *all* the matches.

1.1 Motivation

This work was driven by a critical industrial need identified by engineers at Intel.

28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025).
Editors: Jeremias Berg and Jakob Nordström; Article No. 22; pp. 22:1–22:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The modern semiconductor design process relies heavily on the standard-cell methodology, where designers build complex circuits using fundamental building blocks called *standard cells*, organized into *standard cell libraries*. *Library mapping* [6, 28] is the process of transitioning between libraries to re-implement the same logical functionality using a new library. Intel engineers found that standard Boolean matching-based library mapping often failed to meet timing requirements. Specifically, their timing tool revealed significant delay variances in matched pairs, whereas it is crucial to identify matches with minimal delay variance.

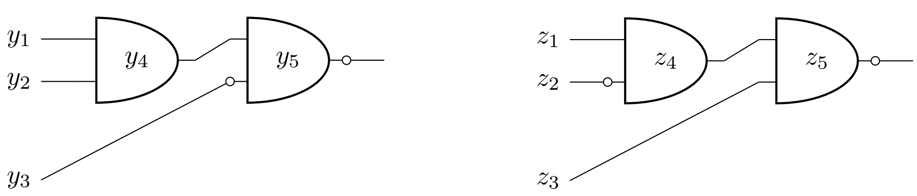
The key issue was that for a given pair of matching cells, multiple input mappings were possible, each resulting in a different delay, yet existing Boolean matching tools returned an arbitrary match. To address this, engineers requested a tool capable of enumerating *all* possible Boolean matches between two cells or proving their absence, so that they could run their timing tool on the results to identify the best match. Notably, the timing tool calculates a complex function that varies according to project specifications, making it infeasible to express the problem using an optimization paradigm like MaxSAT.

This paper presents `EBat`, our novel open-source tool developed to enumerate all possible Boolean matches or prove their absence, fulfilling the engineers' requirement. `EBat` has been productized and is actively used at Intel for library mapping.

We have strong reasons to believe that `EBat` will be valuable to both industrial practitioners and researchers. First, poor timing in library mapping is a common and critical challenge in semiconductor design. Second, a tool that efficiently enumerates all solutions to a widely used decision problem, such as Boolean matching, can open new research directions and practical applications.

1.2 Our Focus

Having clarified our motivation, this paper focuses on algorithmic solutions for all-Boolean-matching. We restrict ourselves to combinational circuits with a *single output*. In line with the literature (see, e.g., [26]), we distinguish between three types of equivalence: Permutation-Equivalence (*P-Equivalence*), where only input permutations are allowed; Negation-Permutation-Equivalence (*NP-Equivalence*), where inputs can also be negated; and Negation-Permutation-Negation-Equivalence (*NPN-Equivalence*), where both inputs and outputs can be negated. In the case of single-output circuits, NPN-equivalence can be reduced to two NP-equivalence checks: the first runs NP-equivalence on Y and Z as-is; the second runs it on Y and Z with Z's output negated. Consequently, this work focuses on *P-equivalence* and *NP-equivalence*. Fig. 1 shows an example of finding all Boolean matches for both P- and NP-equivalence.



(a) Circuit Y implementing $\neg((y_1 \wedge y_2) \wedge \neg y_3)$. **(b)** Circuit Z implementing $\neg((z_1 \wedge \neg z_2) \wedge z_3)$.

Figure 1 For P-equivalence, there are two matches between Y and Z: $[z_1, z_3, z_2]$ (that is, $y_1 \mapsto z_1$; $y_2 \mapsto z_3$; $y_3 \mapsto z_2$) and $[z_3, z_1, z_2]$. For NP-equivalence, there are these two plus four more: $[z_1, \neg z_2, \neg z_3]$, $[\neg z_2, z_1, \neg z_3]$, $[\neg z_2, z_3, \neg z_1]$, and $[z_3, \neg z_2, \neg z_1]$.

1.3 Previous Work: BooM

To our knowledge, BooM [26, 25] is the only Boolean matching approach capable of enumerating all matches. We refer to BooM's all-Boolean-matching algorithm as the *sifter* (BooMS). At a high level, BooMS sifts through a given bucket of mappings using a *mismatch-sifter*, filtering out mismatches and collecting the remaining mappings in a new bucket containing only matches, which are then reported to the user.

Specifically, BooMS constructs two SAT instances: misms, initialized to capture only mismatches, and bucket, initialized to include all possible mappings. BooMS iteratively visits each unvisited mismatch by querying misms for a solution π. It then blocks π and potentially other mismatches from *both* misms and bucket by adding the same *blocking clause* to both instances. For NP-equivalence, before blocking, the mismatch witness (that is, the solution) is further *extended* through *generalization* [35, 21] using *backward ternary simulation (aka justification)* [37, 39] to cover additional mismatches. Once misms returns UNSAT, bucket contains only matches as its solutions. These matches can then be enumerated by repeatedly querying bucket to obtain and block each match.

1.4 Our Contributions: New Algorithms and EBat

Our solution, EBat, implemented from scratch, reuses BooM's SAT encoding while introducing novel all-Boolean-matching algorithms based on the following insights.

The feasibility of all-Boolean-matching algorithms hinges on efficiently identifying and blocking multiple mismatches and matches simultaneously – a capability essential for scaling to real-world instances. We observed a fundamental limitation in the original BooMS algorithm: it enumerates, extends, and blocks only mismatches. In the following, we describe our contributions, starting with a new algorithm that addresses the above limitation:

- **Our first contribution** is a novel algorithm, named the *picker* (EBatP), which enumerates not only mismatches but also matches. Its core capability, which distinguishes it from BooMS, is explicitly visiting and *strengthening* matches using minimal unsatisfiable core extraction [20, 18] to cover and block multiple matches simultaneously, while still leveraging witness extension for efficient mismatch handling, similarly to BooMS. We observed EBatP breaking a performance bottleneck and solving significantly more instances, but only for NP-equivalence. Our analysis suggested this stemmed from the lack of witness extension for P-equivalence, as we followed BooMS, which omits this procedure since applying generalization for P-equivalence compromises correctness (details in Sect. 5). This led us to our second contribution.
- **Our second contribution** is a new witness extension algorithm for P-equivalence, achieved through a dedicated modification of SAT solver heuristics. With this approach, we successfully broke the performance bottleneck for P-equivalence as well. The following three algorithmic contributions further increase the number of solved instances:
- **Our third contribution** is a novel high-level "sift-and-pick" algorithm, EBatC, which combines BooMS and EBatP.
- **Our fourth contribution** is more efficient witness extension for NP-equivalence using a new generalization algorithm, inspired by our recent results in solution enumeration for circuits (AllSAT-CT) [21], which outperforms BooM's witness extension method (i.e., generalization via backward ternary simulation).
- **Our fifth contribution** is a novel, dedicated mismatch-blocking algorithm for P-equivalence.

- Finally, **our sixth contribution** comprises the implementation of all our algorithms and the baseline `BooM` algorithm in a new open-source tool, `EBat`. Despite the kind assistance of `BooM`'s authors, we could not get the original implementation to work, making `EBat` the only publicly available all-Boolean-matching tool.

Experiments show that our algorithms solve 3 to 4 times more benchmarks than the baseline `BooMS` algorithm within `EBat` for both P- and NP-equivalence across a diverse benchmark set.

In what follows, Sect. 2 provides preliminaries, and Sect. 3 reviews prior work. Sect. 4 introduces the `EBat` algorithms, with mismatch handling for P-equivalence detailed in Sect. 5, and correctness discussed in Sect. 6. Experimental results are presented in Sect. 7, and conclusions in Sect. 8.

2 Preliminaries

We establish the relevant notation, assuming familiarity with Boolean logic fundamentals. Let V be a set of Boolean variables. A *literal* l is either a variable $v \in V$ or its negation $\neg v$. The set of literals corresponding to all variables in V is denoted by $\nu^V := \{v \mid v \in V\} \cup \{\neg v \mid v \in V\}$. For a single variable v, we overload the notation by letting ν^v denote either v or $\neg v$, non-deterministically. Given a function F, $Dom(F)$ denotes its domain.

We introduce the semantics used in this paper, omitting the standard Boolean semantics for brevity. *Ternary logic* [34] extends Boolean logic by introducing a third value, *don't-care* (X). Formally, a *ternary assignment* $\tau : V \mapsto \{0, 1, X\}$ assigns each variable one of the *ternary values* $\{0, 1, X\}$. Hereafter, the term *assignment* will refer to a ternary assignment unless otherwise specified, and every assignment is total (i.e., it is defined for all variables in its domain). We omit variables assigned X when listing assignments. For example, $\tau(\{v_1, v_2\}) \equiv \{v_1 := 1\}$ represents $\tau(\{v_1, v_2\}) \equiv \{v_1 := 1, v_2 := X\}$. The *cardinality* $|\tau|$ is the number of variables in τ assigned either 0 or 1. An assignment is *Boolean* if it has maximal cardinality. To evaluate a formula under an assignment τ, standard Boolean semantics is extended with $(\neg X \equiv X)$, $(X \wedge 1 \equiv X)$, $(X \wedge 0 \equiv 0)$, and $(X \wedge X \equiv X)$. An assignment $\rho(V)$ *subsumes* the assignment $\tau(V)$, denoted by $\rho \subseteq \tau$, if $\tau(v) = \rho(v)$ for every v such that $\rho(v) \in \{0, 1\}$. If $\rho(V)$ subsumes $\tau(V)$, then $\tau(V)$ *extends* $\rho(V)$. For example, $\tau_1 \equiv \{v_1 := 1\}$ subsumes $\tau_2 \equiv \{v_1 := 1, v_2 := 0\}$, whereas τ_2 extends τ_1. We now proceed to define a circuit.

▶ **Definition 1** (Circuit). *A combinational Boolean* circuit *with i inputs I^Y, g gates G^Y, and output o^Y is the Boolean structure:*

$$\overset{g}{\underset{i}{Y}}(I^Y = \{y_1^Y, \dots, y_i^Y\}) = \langle G^Y = \{y_{i+1}^Y \leftrightarrow g_{i+1}^Y, \dots, y_{i+g}^Y \leftrightarrow g_{i+g}^Y\}, o^Y \in \nu^{\{y_{i+g}^Y\}}\rangle$$

Each input $y_p^Y \in I^Y$ is a Boolean variable. For each gate, g_k^Y is of the form:

$$g_k^Y = \left(\nu^{y_{k_1}^Y} \bigcirc_k \nu^{y_{k_2}^Y}\right) | 1 \leq k_1, k_2 < k$$

with $\nu^{y_{k_1}^Y}$ and $\nu^{y_{k_2}^Y}$ representing the (possibly negated) inputs to the gate and $\bigcirc_k$ being a Boolean operator (e.g., $\wedge$, $\vee$, $=$, $\oplus$).

For example, Fig. 1a illustrates the circuit $\overset{2}{\underset{3}{Y}}(I = \{y_1, y_2, y_3\}) = \langle G = \{y_4 \leftrightarrow y_1 \wedge y_2, y_5 \leftrightarrow y_4 \wedge \neg y_3\}, o \equiv \neg y_5\}\rangle$. Towards extending the semantics to circuits, we define ternary simulation [37, 24]. Intuitively, ternary simulation propagates an input assignment τ from the circuit inputs through its gates to the output.

▶ **Definition 2** (Ternary Simulation). *Given a circuit* $\overset{g}{\underset{i}{Y}}$ *with inputs* $I^Y = \{y_1^Y, \ldots, y_i^Y\}$, *gates* $G^Y = \{y_{i+1}^Y \leftrightarrow g_{i+1}^Y, \ldots, y_{i+g}^Y \leftrightarrow g_{i+g}^Y\}$, *output* $o^Y \in \nu^{\{y_{i+g}^Y\}}$, *and an assignment* $\tau(I^Y) : I^Y \mapsto \{0, 1, X\}$ *to the circuit's inputs,* ternary simulation *transforms* τ *into the assignment* $\tau^S(\{y_1^Y, \ldots, y_{i+g}^Y\})$, *where:*

1. $\tau^S(y_p^Y) := \tau(y_p^Y)$ *for each input* $y_p^Y \in I^Y$.
2. *For each gate* $y_k^Y \leftrightarrow g_k^Y$, *where* $g_k^Y = \left(\nu^{y_{k_1}^Y} \bigcirc_k \nu^{y_{k_2}^Y}\right)$*:*

$$\tau^S(y_k^Y) := \tau^S(\nu^{y_{k_1}^Y}) \bigcirc_k \tau^S(\nu^{y_{k_2}^Y})$$

For the example circuit Y in Fig. 1a, ternary propagation would propagate $\tau(I \equiv \{y_1, y_2, y_3\}) \equiv \{y_1 := 0\}$ to $\tau^S(\{y_1, \ldots, y_5\}) \equiv \{y_1 := 0, y_4 := 0, y_5 := 0\}$.

We are now prepared to define what it means for an input assignment to serve as a circuit solution. Our definition relies on *entailment* [38] following the most general of the three formulations of a circuit solution we presented in [21]. Intuitively, $\tau(I)$ is a solution if extending $\tau(I)$ to *any* Boolean assignment and propagating by ternary simulation always results in the circuit outputting 1.

▶ **Definition 3** (Entailment, Solution). *Given a circuit* $\overset{g}{\underset{i}{Y}}$ *with inputs* $I^Y = \{y_1^Y, \ldots, y_i^Y\}$, *gates* $G^Y = \{y_{i+1}^Y \leftrightarrow g_{i+1}^Y, \ldots, y_{i+g}^Y \leftrightarrow g_{i+g}^Y\}$, *output* $o^Y \in \nu^{\{y_{i+g}^Y\}}$, *and an assignment* $\tau(I^Y) : I^Y \mapsto \{0, 1, X\}$ *to the circuit's inputs,* τ entails Y *(denoted by* $\tau \models Y$*), if* $\rho^S(o) = 1$ *for* any ρ *which substitutes* every X *in* τ *by* any *Boolean value. Furthermore,* τ *is a* solution *to* Y *if and only if* $\tau \models Y$.

For example, in the circuit Y shown in Fig. 1a, $\tau_1(I) \equiv \{y_1 := 0\} \models Y$, as any extension followed by propagation of τ_1 yields $o \equiv \neg y_5 := 1$. Conversely, $\tau_2(I) \equiv \{y_1 := 1\} \not\models Y$, since extending τ_2 to $\{y_1 := 1, y_2 := 1, y_3 := 0\}$ and propagating results in $o \equiv \neg y_5 := 0$. We proceed with mappings-related definitions.

▶ **Definition 4** (Mapping, Permutation). *Let* $I^Y = \{y_1, \ldots, y_i\}$ *and* $I^Z = \{z_1, \ldots, z_i\}$ *be ordered sets of* i *Boolean variables each. A* mapping π *is a function* $\pi : I^Y \to \nu^{I^Z}$ *injective w.r.t. variables, i.e., if* $y_p \in Dom(\pi)$ *is mapped to either* z_q *or* $\neg z_q$*, then no other element* $y_r \in Dom(\pi)$*, where* $r \neq p$*, can be mapped to* z_q *or* $\neg z_q$.

A mapping π *is* total *if* $Dom(\pi) = I^Y$*, whereas any mapping, including total mappings, is considered* partial *(i.e., the term mapping refers to a partial mapping unless specified otherwise). A mapping* π *is a* permutation *if its range contains only non-negated variables, i.e.,* $\pi(y) \in I^Z$ *for all* $y \in Dom(\pi)$.

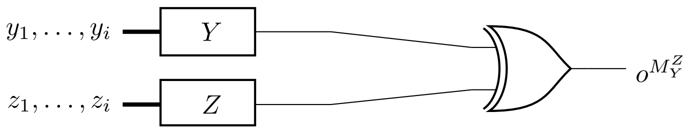
Given $I^Y = \{y_1, y_2, y_3\}$ and $I^Z = \{z_1, z_2, z_3\}$, two example (total) mappings are: $\rho_1 = [\neg z_3; z_2; z_1]$ and $\rho_2 = [z_1; z_3; z_2]$, where we represent the mapping $[y_1 \mapsto \nu^{z_{q_1}}; \ldots; y_n \mapsto \nu^{z_{q_n}}]$ by the shortened notation $[\nu^{z_{q_1}}, \ldots, \nu^{z_{q_n}}]$. Here, ρ_1 is *not* a permutation because its range includes a negated variable, whereas ρ_2 is. When representing partial mappings, we use the bullet sign $\bullet$ for any unmapped elements. For example, $\rho_3 = [y_1 \mapsto \neg z_3;\ y_3 \mapsto z_1]$ can be written as $[\neg z_3; \bullet; z_1]$.

We next extend the semantics to mappings. A (total ternary) assignment τ satisfies a (partial) mapping π if τ can be extended to an assignment ρ that assigns Boolean values consistently with π to all the variables in Dom(π).

▶ **Definition 5** (Satisfy a Mapping). *Given the ordered sets* $I^Y = \{y_1, \ldots, y_i\}$ *and* $I^Z = \{z_1, \ldots, z_i\}$*, a mapping* $\pi : I^Y \to \nu^{I^Z}$*, and an assignment* $\tau(I^Y \cup I^Z)$*, we say that* τ satisfies π *(denoted by* $\tau \models\!\approx \pi$*) if there exists* ρ *such that* $\rho \supseteq \tau$ *and:*

$$\forall y_p \in Dom(\pi) : \rho(y_p) \in \{0, 1\} \text{ and } \rho(y_p) = \rho(\pi(y_p)).$$

Consider, for example, the mapping $\pi \equiv [z_2; \bullet; \bullet]$ from $I^Y = \{y_1, y_2, y_3\}$ to $I^Z = \{z_1, z_2, z_3\}$. Any assignment τ that does not assign Boolean values to both y_1 and z_2, in a way inconsistent with π, satisfies π. For example, $\{y_1 := 0; z_2 := 1\}$ does *not* satisfy π, but $\{y_1 := 0\}$ does (since $\{y_1 := 0\}$ can be extended to $\{y_1 := 0; z_2 := 0\}$). We next define the *miter* [11], a circuit that combines two given circuits by XORing their outputs as follows:



▶ **Definition 6** (Miter). *Given two circuits* $\overset{g}{\underset{i}{Y}}(I^Y)$ *and* $\overset{g'}{\underset{i}{Z}}(I^Y)$*, their miter* M_Y^Z *is the circuit*
$\overset{g+g'+1}{\underset{2i}{M_Y^Z}}(\{y_1, \ldots, y_i, z_1, \ldots, z_i\}) = \langle \{y_{i+1}, \cdots y_{i+g}, z_{i+1}, \cdots, z_{i+g'}, y_{i+g} \oplus z_{i+g'}\}, y_{i+g} \oplus z_{i+g'} \} \rangle$.

The miter is used to determine if a mapping π is a match or a mismatch: π is a mismatch iff there is a witness – an assignment to the miter inputs – that entails the miter and satisfies π; otherwise, π is a match. Below, we formalize the related definitions for both P- and NP-equivalence, with the only difference being the restriction of the examined mappings to permutations for P-equivalence.

▶ **Definition 7** (Mismatch, (Mismatch) Witness, Match). *Given circuits* $\overset{g}{\underset{i}{Y}}(I^Y)$ *and* $\overset{g'}{\underset{i}{Z}}(I^Z)$ *and a mapping* $\pi : I^Y \rightarrow R$*, where* $R \equiv \nu^{I^Z}$ *for NP- and* $R \equiv I^Z$ *for P-equivalence,* π *is a* mismatch *between* Y *and* Z *iff there exists an assignment* $\sigma(I^Y \cup I^Z)$*, called a* (mismatch) witness *for* π*, such that:*
1. $\sigma \models M_Y^Z$*, and*
2. $\sigma \approx\!\!\!\!\models \pi$.

Furthermore, any mapping $\pi : I^Y \rightarrow R$ *that is not a mismatch between* Y *and* Z *is a* match *between them.*

Given the two circuits in Fig. 1, the assignment $\tau(I^Y \cup I^Z) \equiv \{y_1 := 1, y_2 := 1; y_3 := 0; z_3 := 0\}$ is a mismatch witness for the identity mapping $\pi \equiv [z_1, z_2, z_3]$. First, propagating τ causes the outputs of Y and Z to differ, resulting in M_Y^Z outputting 1; thus, τ entails the miter. Second, τ satisfies π, as it can be extended consistently with π by assigning 1 to both z_1 and z_2. Conversely, the mapping $\rho = [z_3, z_1, z_2]$ is a match, as no assignment satisfies ρ and entails the miter.

To enable testing whether a mapping is a match using a SAT solver, we introduce the concepts of a circuit formula and a mapping formula.

▶ **Definition 8** (Circuit Formula). *Given a circuit* $\overset{g}{\underset{i}{Y}}$ *with inputs* $I^Y = \{y_1^Y, \ldots, y_i^Y\}$*, gates* $G^Y = \{y_{i+1}^Y \leftrightarrow g_{i+1}^Y, \ldots, y_{i+g}^Y \leftrightarrow g_{i+g}^Y\}$*, output* $o^Y \in \nu^{\{y_{i+g}^Y\}}$*, its Boolean* circuit formula f^Y *is:*

$$f^Y \equiv o^Y \wedge \bigwedge_{k=i+1}^{i+g} \left(y_k^Y \leftrightarrow g_k^Y\right).$$

▶ **Definition 9** (Mapping Formula). *Given the ordered sets* $I^Y = \{y_1, \ldots, y_i\}$ *and* $I^Z = \{z_1, \ldots, z_i\}$ *and a mapping* $\pi : I^Y \rightarrow \nu^{I^Z}$*,* π*'s Boolean* mapping formula f^π *is:*

$$f^\pi \equiv \bigwedge_{p=1}^{i} (y_p \leftrightarrow \pi(y_p)).$$

The following lemma is central to the algorithms in this paper and the underlying works [26, 25]. It can be easily verified.

▶ **Lemma 10.** *Given two circuits Y_i^g and $Z_i^{g'}$, a mapping π between their inputs is a match if and only if $f^{M_Y^Z} \wedge f^{\pi}$ is unsatisfiable.*

We need to define subsumption for mappings. Intuitively, π_1 subsumes π_2 if π_2 agrees with π_1 on the entire domain of π_1.

▶ **Definition 11** (Subsumption for Mappings). *Given the ordered sets I^Y and I^Z of the same cardinality, let $\pi_1, \pi_2 : I^Y \rightarrow \nu^{I^Z}$ be two mappings. We say that π_1* subsumes *π_2, denoted $\pi_1 \subseteq \pi_2$, if for every $y_p \in Dom(\pi_1)$, we have $y_p \in Dom(\pi_2)$ and $\pi_1(y_p) = \pi_2(y_p)$.*

For example, $\pi_1 = [\bullet, \neg z_3; \neg z_2]$ subsumes $\pi_2 = [z_1, \neg z_3; \neg z_2]$. We are now ready to define all-Boolean-matching for both NP- and P-equivalence.

▶ **Definition 12** (All-Boolean-Matching). *Given circuits $\overset{g}{\underset{i}{Y}}(I^Y)$ and $\overset{g'}{\underset{i}{Z}}(I^Z)$, and assuming $R \equiv \nu^{I^Z}$ for NP- and $R \equiv I^Z$ for P-equivalence, an* all-Boolean-matching *algorithm reports a set of mappings $S \subseteq (I^Y \rightarrow R)$ such that:*
1. *Every $\pi \in S$ is a match between Y and Z, and*
2. *Any $\rho : I^Y \rightarrow R$ not subsumed by a $\pi \in S$ is a mismatch between Y and Z.*

Notably, we allow a total match to be subsumed by more than one of the reported matches. In other words, the reported matches need not be disjoint.

We conclude with a brief review of relevant SAT-related concepts and generalization.

A *cardinality constraint* is a Boolean constraint that ensures that at-most, at-least or exactly k literals hold in a literal set. A *clause* is a disjunction (set) of literals. A *cube* is a conjunction (set) of literals. A formula F is in *Conjunctive Normal Form (CNF)* if it is a conjunction (set) of clauses. Given a CNF formula F, a SAT solver decides whether F is satisfiable. Given a satisfiable formula, a SAT solver also returns its *total Boolean solution (solution)*. Many SAT solvers are *incremental* [20, 33]: they can be invoked multiple times, where, for every new query $\text{SAT}(F, A)$, the SAT solver also receives a cube of *assumption literals (assumptions)* A, which hold only for the current query. The solver then decides whether $F \wedge A$ is satisfiable (where F contains all the clauses provided so far). If $F \wedge A$ is unsatisfiable, $\text{SAT}(F, A)$ returns an *Unsatisfiable Core (UC)*, that is, a cube $A' \subseteq A$, such that $F \wedge A'$ is still unsatisfiable [20].

Given a circuit T and its Boolean solution $\sigma(I^T)$ (that is, σ entails T), *generalization* [35, 27, 39, 21] transforms (or *generalizes*) σ into a smaller ternary solution σ' such that $\sigma' \subseteq \sigma$ by replacing as many Boolean values as possible with X's, while ensuring that σ' still entails T. One commonly used generalization approach is the ternary-simulation-based *Forward Ternary Simulation (`FTS`)* [37, 19], which iteratively attempts to replace each input's Boolean value with the don't-care value X, using ternary simulation (Def. 2) to check whether the circuit remains satisfied. Another common ternary-simulation-based approach is *Backward Ternary Simulation (`BTS`)* [37, 39], also known as *justification*. This approach checks which internal gates and, ultimately, inputs can be assigned X while still satisfying the circuit, proceeding backward from outputs to inputs. In [21], we demonstrated that *Minimal Unsatisfiable Core (`MUC`)*-based generalization [14], which leverages properties of the so-called *dual circuit* (i.e., the original circuit with its output negated), is theoretically more powerful and empirically more efficient than both `BTS` and `FTS`. In fact, the best results in the context of enumerating circuit solutions were achieved by combining `FTS` and `MUC` (denoted `FTS&MUC` in this work and as `ROC` in [21]).

3 Previous Work: BooM

In this section, we present BooM's all-Boolean-matching flow [26, 25], including its SAT encoding in Sect. 3.1 and the sifter (BooMS) algorithm in Sect. 3.2.

$$\left(\begin{array}{c|cccc} & z_1 & z_2 & \dots & z_i \\ \hline y_1 & \{x^+_{11}, x^-_{11}\} & \{x^+_{12}, x^-_{12}\} & \dots & \{x^+_{1i}, x^-_{1i}\} \\ y_2 & \{x^+_{21}, x^-_{21}\} & \{x^+_{22}, x^-_{22}\} & \dots & \{x^+_{2i}, x^-_{2i}\} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_i & \{x^+_{i1}, x^-_{i1}\} & \{x^+_{i2}, x^-_{i2}\} & \dots & \{x^+_{ii}, x^-_{ii}\} \end{array}\right)$$

(a) Indicator variables $D = \{x^+_{pq}, x^-_{pq} \mid 1 \leq p, q \leq i\}$ to represent mappings between $I^Y = \{y_1, \dots, y_i\}$ and $I^Z = \{z_1, \dots, z_i\}$, where x^+_{pq} indicates if $y_p \mapsto z_q$, and x^-_{pq} indicates if $y_p \mapsto \neg z_q$.

$$\left(\forall p \in [1, \dots, i] : \left(\textstyle\sum_{q=1}^{i}(x^+_{pq} + x^-_{pq})\right) = 1\right) \bigwedge \left(\forall q \in [1, \dots, i] : \left(\textstyle\sum_{p=1}^{i}(x^+_{pq} + x^-_{pq})\right) = 1\right)$$

(b) The *map-validity* constraint (comprising a conjunction of two cardinality constraints) ensures that each $y_p \in I^Y$ maps to exactly one z_q or $\neg z_q$, where no other $y_r \in I^Y | r \neq p$ maps to z_q or $\neg z_q$.

$$\bigwedge_{1 \leq p,q \leq i} (x^+_{pq} \rightarrow (y_p \leftrightarrow z_q)) \wedge (x^-_{pq} \rightarrow (y_p \leftrightarrow \neg z_q))$$

(c) *Map-to-inputs* constraint: x^+_{pq} implies $y_p \leftrightarrow z_q$, and x^-_{pq} implies $y_p \leftrightarrow \neg z_q$.

Figure 2 BooM's SAT encoding: indicator variables and related constraints.

3.1 SAT Encoding

BooM maintains two SAT instances: `bucket` and `misms`. Both include a Boolean variable for each input of both circuits: $I^Y = \{y_1, \dots, y_i\}$ and $I^Z = \{z_1, \dots, z_i\}$. To reason about mappings between I^Y and I^Z, both instances contain the indicator variables $D = \{x^+_{pq}, x^-_{pq} \mid 1 \leq p, q \leq i\}$, which represent mappings between $I^Y = \{y_1, \dots, y_i\}$ and $I^Z = \{z_1, \dots, z_i\}$, where x^+_{pq} indicates if $y_p \mapsto z_q$, and x^-_{pq} indicates if $y_p \mapsto \neg z_q$. See Fig. 2a for an illustration. Additionally, we will use the notation x^s_{pq} to denote either x^+_{pq} or x^-_{pq}.

For P-equivalence, all x^-_{pq} variables are fixed to 0. This is the only adjustment needed to adapt the SAT encoding from the default NP-equivalence to P-equivalence.

The first SAT instance, `bucket`, is initialized with the map-validity constraint shown in Fig. 2b, ensuring that it initially represents *all possible mappings*.

The second SAT instance, `misms`, is initialized with *only mismatches*. To achieve this, in addition to the map-validity constraint, it is also initialized with the map-to-inputs constraint shown in Fig. 2c, and the miter formula $f^{M^Z_Y}$ – representing the miter circuit translated to CNF. Each solution to `misms` corresponds to a total mapping π due to the map-validity constraint, but it is restricted to mismatches because every solution must assign input values consistently with the mapping (enforced by the map-to-inputs constraint) while satisfying the miter formula $f^{M^Z_Y}$, ensuring a mismatch.

In the presentation below, we assume that circuits and cardinality constraints are implicitly translated into clauses. Our implementation uses Tseitin encoding [41] for circuit translation and nested encoding [7] for cardinality constraints.

3.2 `BooMS` Algorithm

We present the `BooMS` algorithm in Alg. 2 (please recall Sect. 1.3 for its high-level flow). Differently from the original presentation [26, 25], we isolated the SiftMis subroutine in Alg. 1 to enable SiftMis's integration into our novel `EBatC` algorithm (Sect. 4.2). As shown in Alg. 2, `BooMS` begins by invoking SiftMis.

SiftMis, shown in Alg. 1, takes circuits Y and Z and returns the SAT instance `bucket` containing all total *matches* between them as its solutions. As explained in Sect. 3.1, `bucket` is initialized with all possible mappings (line 1), while SiftMis also maintains another SAT instance `misms`, initialized to capture only mismatches (line 2).

SiftMis proceeds with a while loop at line 3, iterating over all mismatches by querying `misms` until no more are found (i.e., when `misms` returns UNSAT). In each iteration, the algorithm queries `misms` for a new mismatch, extends the witness to cover additional mismatches, and blocks them in *both* `misms` and `bucket`, as detailed in Sect. 3.2.1. Once no more mismatches remain, the algorithm returns `bucket`.

Going back to `BooMS` (Alg. 2), after initializing `matches` with the matches by invoking SiftMis, `BooMS` iteratively reports and blocks the matches.

3.2.1 Mismatch Handling in `BooMS`

We begin by showing the clause added by `BooMS` to block mismatches (when BlkMis is applied at lines 5 and 6 of SiftMis in Alg. 1), distinguishing between NP- and P-equivalence.

For NP-equivalence, the following clause C_σ^N is added to both `bucket` and `misms`:

$$C_\sigma^N := \bigvee_{1\leq p,q\leq i} \begin{cases} x_{pq}^+ & \text{if } \sigma(y_p) \neq \sigma(z_q) \text{ and } \sigma(y_p), \sigma(z_q) \in \{0,1\}, \\ x_{pq}^- & \text{if } \sigma(y_p) = \sigma(z_q) \text{ and } \sigma(y_p), \sigma(z_q) \in \{0,1\}. \end{cases}$$

Adding C_σ^N ensures that every mismatch π satisfied by σ is blocked: for each such π, C_σ^N forces at least one Boolean-assigned y_p to map to $\neg\pi(y_p)$. Furthermore, no mappings unsatisfied by σ are blocked. Indeed, for any such π, there must exist a y_p such that $\sigma(y_p) \neq \sigma(\pi(y_p))$ (otherwise, π would have been satisfied by σ), ensuring that C_σ^N is satisfied by either x_{pq}^+ or x_{pq}^-, assuming $\pi(y_p) = z_q$.

For P-equivalence, the blocking clause is as follows:

$$C_\sigma^P := \bigvee x_{pq}^+ \quad \text{for all } p, q \text{ such that } \sigma(y_p \in I^Y) = 0 \text{ and } \sigma(z_q \in I^Z) = 1$$

This enforces one of the Y inputs to be mapped to a Z input currently assigned a different value, thereby blocking all satisfied mismatches – and only them, since, by construction, C_σ^P includes a satisfied indicator for any mapping unsatisfied by σ, similar to NP-equivalence.

We introduce another notation: Given an assignment σ to $I^Y \cup I^Z$, we call an indicator $x_{pq}^s \in D$ (recall that $s \in \{-,+\}$) an *X-indicator* if either y_p, z_q, or both are assigned X in σ.

Only for NP-equivalence, before blocking, `BooMS` extends the witness through generalization (via `BTS`): it replaces Boolean values assigned to inputs in σ with X's, while still satisfying the miter. Notably, any X-indicators are then dropped from the blocking clause, thereby blocking more mismatches at once. This is valid because extending the generalized witness to *any* Boolean assignment (by replacing all X's with Boolean values) results in a Boolean mismatch witness, with our clause blocking all mismatches satisfied by these witnesses at once (as if a blocking clause were added for each such Boolean mismatch). In contrast, for *P-equivalence*, generalizing and dropping X-indicators from the blocking clause is incorrect (see Sect. 5) likely explaining why `BooM` does not extend witnesses for P-equivalence.

Algorithm 1 SiftMis.

```
Input: Circuits Y (g over, i under) and Z (g' over, i under)
Output: The SAT instance bucket with only the matches remaining
Initialize the bucket SAT instance with all mappings        ▷ by map-validity (Fig. 2b)
Initialize the misms SAT instance                            ▷ by map-validity, map-to-inputs, f^{M_Y^Z}
while σ := SAT(misms) is satisfiable do
    σ′ := ExtWit(M_Y^Z, σ)                  ▷ Extend the witness σ to satisfy more mismatches
    BlkMis(misms, σ′)                        ▷ Block the satisfied mismatches in misms
    BlkMis(bucket, σ′)                       ▷ Block the satisfied mismatches in bucket
return bucket
```

Algorithm 2 BooMS.

```
Input: Circuits Y (g over, i under) and Z (g' over, i under)
Output: All the matches between Y and Z
matches := SiftMis(Y, Z)
while π := SAT(matches) is satisfiable do
    Report π to the user
    Block π in matches, with a clause containing ¬x^s_pq for every satisfied x^s_pq ∈ D
```

4 All-Boolean-Matching Algorithms in `EBat`

We introduce our high-level algorithms `EBatP` (Sect. 4.1) and `EBatC` (Sect. 4.2).

4.1 The Picker `EBatP`

`EBatP` implements a straightforward *picker* classification algorithm: given a bucket of total mappings, the picker iteratively removes a mapping π, reporting it if it is a match. The key to `EBatP`'s efficiency lies in classifying and removing multiple total mappings simultaneously for *both* mismatches and matches – unlike the previous state-of-the-art algorithm `BooMS` (Sect. 3.2), which did so only for mismatches. Moreover, `EBatP` handles mismatches significantly more efficiently, especially for P-equivalence (more on this later).

`EBatP` is presented in Alg. 3. In addition to the input circuits, `EBatP` can optionally accept a pre-initialized SAT instance, `bucket`, from the user. For the remainder of Sect 4.1, assume that `bucket` is *not* provided.

`EBatP` maintains two SAT instances: `bucket` and `classifier`. `bucket`, initialized at line 2 with map-validity constraint in Fig. 2b, maintains unclassified total mappings as its solutions (similarly to `bucket` in `BooMS`). `classifier`, initialized at line 3 with the miter formula $f^{M_Y^Z}$ and the map-to-inputs constraint shown in Fig. 2c, is used to classify a mapping as either a match or a mismatch.

Algorithm 3 `EBatP`.

Input: Circuits $\overset{g}{\underset{i}{Y}}$ and $\overset{g'}{\underset{i}{Z}}$, and, optionally, the SAT instance `bucket`

Output: All the matches between Y and Z

```
1:  if bucket is not provided by the user then
2:      Initialize the bucket SAT instance                ▷ by map-validity (Fig. 2b)
3:  Initialize the classifier SAT instance    ▷ by f^{M_Y^Z} and map-to-inputs (Fig. 2c)
4:  while π := SAT(bucket) is satisfiable do
5:      σ := SAT(classifier, Q^π)                                   ▷ Is π a match?
6:      if UNSAT then                                                ▷ π is a match
7:          π' := StrenMatch(classifier, π)      ▷ Strengthen π by minimizing the UC
8:          Report the match π' to the user        ▷ Report the strengthened match π
9:          BlkMatch(bucket, π')                   ▷ Block π' by adding ¬Q^{π'} to bucket
10:     else                                                          ▷ π is a mismatch
11:         σ' := ExtWit(M_Y^Z, σ)     ▷ Extend the witness σ to satisfy more mismatches
12:         BlkMis(bucket, σ')                           ▷ Block the satisfied mismatches
```

$$Q^\pi \equiv \bigwedge_{p:y_p \in \mathrm{Dom}(\pi)} \begin{cases} x^+_{pq} & \text{if } \pi(y_p) = z_q \\ x^-_{pq} & \text{if } \pi(y_p) = \neg z_q \end{cases}$$

(a) π-*cube* Q^π: assuming Q^π triggers π (i.e, f^π), given the map-to-inputs constraint.

$$\pi^Q(y_p) = \begin{cases} z_q, \ \text{if } x^+_{pq} \in Q \\ \neg z_q, \ \text{if } x^-_{pq} \in Q \\ \text{unmapped (i.e., } y_p \notin \mathrm{Dom}(\pi^Q)\text{), if } x^+_{pq} \notin Q \text{ and } x^-_{pq} \notin Q \end{cases}$$

(b) Extracting the mapping π^Q from a π-cube Q.

Figure 3 Building the cube Q^π to represent a mapping π and extracting a mapping from a cube.

4.1.1 The Main Loop

After initializing both SAT instances, the algorithm enters the while loop at line 4, iterating over all mappings until no further mappings are found (i.e., when `bucket` returns UNSAT), at which point it terminates.

Each iteration of the algorithm queries `bucket` for an unclassified total mapping π, classifies as either a match or a mismatch via `classifier`, transforms π into a set Γ where every $\rho \in \Gamma$ is a match iff π is, and blocks Γ in `bucket`, reporting matches.

Going back to Alg. 3, assume `bucket` returns a non-classified total mapping π at line 4. By Lemma 10, π is a match iff the conjunction of the miter formula $f^{M_Y^Z}$ (held by `classifier`) and the mapping formula $f^\pi \equiv \bigwedge_{p=1}^{i}(y_p \leftrightarrow \pi(y_p))$ is unsatisfiable. The algorithm tests whether π is a match in `classifier` at line 5 by invoking `classifier` under the assumption cube Q^π in Fig. 3a that enforces f^π for the current `classifier` query.

4.1.2 The Match Case

If `classifier` returns UNSAT, π is classified as a match. The algorithm invokes the function StrenMatch to *strengthen* π to a smaller partial match π' such that $\pi' \subseteq \pi$, based on the unsatisfiable core $U \subseteq Q^{\pi}$, obtained from `classifier`. The UC U induces the partial mapping π^U (see Fig. 3b), which must be a match (otherwise, U would not have been a UC because `classifier` $\wedge U$ would have been satisfiable by a mismatch witness for π^U). Since $U \subseteq Q^{\pi}$, it follows that $\pi^U \subseteq \pi$. Instead of simply returning π^U, StrenMatch, shown below, attempts to derive an even smaller match by iteratively minimizing U [18]:

Function StrenMatch(`classifier`, π):
  U := latest unsatisfiable core from `classifier` ▷ $\pi^U \subseteq \pi$
  forall $a \in U$: **if** SAT(`classifier`, $U \setminus \{a\}$) is UNSAT **then** $U := U \setminus \{a\}$
  return π^U ▷ Recall Fig. 3b

Finally, Alg. 3 invokes the function BlkMatch, shown below, to block the match π', returned by StrenMatch, by adding the clause $\neg Q^{\pi'}$ to `bucket`:

Function BlkMatch(`bucket`, π'):
  AddClause(`bucket`, $\neg Q^{\pi'}$)

4.1.3 The Mismatch Case

If `classifier` returns SAT with a witness σ, then π is a mismatch, satisfied by σ. The algorithm invokes ExtWit, which extends σ to σ', aiming to satisfy additional mismatches while still satisfying π, then calls BlkMis to block all mismatches satisfied by σ'.

For NP-equivalence, our mismatch handling is similar to `BooMS` (Sect. 3.2.1), with the notable exception of the generalization algorithm for extending mismatches. (Recall our brief review of generalization algorithms from the last paragraph of Sect. 2.) While `BooMS` applied `BTS`, and our previous work on enumeration in circuit SAT found that the `FTS`&`MUC` combination works best in that context [21], we show that the `BTS`&`MUC` combination outperforms both `BTS` and `FTS`&`MUC` for all-Boolean-matching (Sect. 7).

Our mismatch handling algorithm for P-equivalence is presented in Sect. 5.

4.2 The Combined `EBatC`

Alg. 4 introduces our combined sift-and-pick `EBatC` algorithm. Our design of `EBatP` and `BooMS` laid the foundation for expressing `EBatC` concisely.

Algorithm 4 `EBatC`.

Input: Circuits $\overset{g}{\underset{i}{Y}}$ and $\overset{g'}{\underset{i}{Z}}$
Output: All the matches between Y and Z
1: `matches` := SiftMis(Y, Z) ▷ See Alg. 1
2: **return** `EBatP`($Y, Z,$ `matches`) ▷ See Alg. 3

`EBatC` first generates the SAT instance `matches` with all total matches using SiftMis (as in `BooMS`). It then enumerates matches via `EBatP` with `matches` as input. Since `matches` contains only matches, `classifier` queries in `EBatP` always return UNSAT. However, these queries remain essential, as strengthening relies on the UC returned by `classifier`.

The key difference between EBatC and our picker algorithm EBatP is that EBatC processes mismatches first, then matches, while EBatP iterates over all mappings. We expected EBatC to perform better since incremental SAT solvers typically handle similar consecutive queries more efficiently. Indeed, Sect. 7 empirically confirms that querying mismatches first (EBatC) is more effective than mixing match and mismatch queries without prior knowledge (EBatP).

4.2.1 The Trade-off between EBatP and EBatC

While EBatC outperforms EBatP in our experiments (Sect. 7), EBatP has an inherent advantage: it is an **anytime** algorithm. Unlike EBatC, which must process all mismatches before enumerating any matches, EBatP begins generating solutions immediately. This property is valuable for difficult instances, where EBatC may stall in its initial phase and yield no matches. In such cases, EBatP can be used to return as many matches as possible to the user.

5 Mismatch Handling for P-Equivalence in EBat

Contrary to NP-equivalence, for P-equivalence, applying generalization to extend the mismatch witness, and dropping X-indicators from the blocking clause (recall Sect. 3.2.1), is illegal – it might block valid matches, as demonstrated by the following example.

Recall Fig. 1. Please keep in mind that, in both valid matches, y_3 maps to z_2. Consider the mapping $\pi = [z_2, z_3, z_1]$. π is a mismatch with the witness $\sigma \equiv \{y_1 := 1, y_2 := 1, y_3 := 0, z_1 := 0, z_2 := 1, z_3 := 1\}$ (as σ satisfies π and propagating σ in Y and Z results in the outputs of the two circuits being assigned different values, thus satisfying the miter). The corresponding blocking clause in BooMS is $C_\sigma^P \equiv (x_{32}^+ \vee x_{33}^+)$ (recall Sect. 3.2.1). Generalization could extend σ by substituting z_2's value with X to $\sigma' \equiv \{y_1 := 1, y_2 := 1, y_3 := 0, z_1 := 0, z_2 := X, z_3 := 1\}$, where σ' is still a mismatch witness. If we allowed dropping X-indicators from C_σ^P, the resulting clause would be $C_{\sigma'}^P \equiv (x_{33}^+)$. However, that would force y_3 to always be mapped to z_3, erroneously blocking both valid matches where y_3 maps to z_2.

The root cause of this limitation is that, for P-equivalence, extending the generalized witness to a Boolean assignment does *not* necessarily result in a mismatch witness as it might violate the following *0-1 balance* property, unique to P-equivalence: to serve as a mismatch witness, a Boolean assignment σ must assign an equal number of b's to inputs in I^Y and I^Z for both $b \in \{0, 1\}$. This is because, otherwise, σ cannot satisfy any mapping, since, under P-equivalence, every input $y_p \in Y$ assigned $b \in \{0, 1\}$ must be mapped to some input $z_q \in Z$ that is also assigned the very same b.

Intuitively, dropping X-indicators from C_σ^P effectively introduces a blocking clause C_ρ^P for a non-witness assignment ρ that violates 0-1 balance. However, adding such clauses – i.e., BooM's blocking clauses based on non-witnesses – is incorrect.

5.1 Witness-Extension for P-Equivalence

Our novel witness-extension procedure for P-equivalence relies on Boolean logic rather than ternary logic and generalization. It aims to maximize the *merit* of the witness:

Let σ be a Boolean mismatch witness. Let $f_\sigma \in \{0, 1\}$ be the more frequent value assigned by σ to I^Y (or, equivalently, I^Z because of 0-1 balance), and $l_\sigma \neq f_\sigma$ be the less frequent value, assuming $f_\sigma = 0$ in case of a tie. The *merit* M_σ of σ is the count of variables in I^Y assigned f_σ. Observe that the higher the merit, the more total mismatches σ satisfies.

Indeed, for P-equivalence, a Boolean witness σ satisfies $s(\sigma) \equiv M_\sigma! \times (i - M_\sigma)!$ total mismatches. The maximum occurs when $M_\sigma = i$ (i.e., all inputs are assigned the same value, satisfying all $i!$ permutations), while the minimum is reached when M_σ is minimized at $\lceil \frac{i}{2} \rceil$. As M_σ increases, $s(\sigma)$ – and consequently the number of satisfied total mismatches – also increases, confirming that a higher merit leads to more satisfied total mismatches, as intended.

For clarification of the latest notions, consider a high-level example with two circuits, each having three inputs. The assignment $\rho \equiv \{y_1 := 1, y_2 := 0, y_3 := 0, z_1 := 0, z_2 := 1, z_3 := 0\}$ can serve as a mismatch witness if it satisfies the miter, where $f_\rho = 0$ and $M_\rho = 2$.

Our witness-extension approach for P-equivalence is *not* applied as part of the ExtWit function following the `bucket` SAT query in `EBatP` (Alg. 3, line 5) or the `misms` SAT query in SiftMis (Alg. 1, line 3). Instead, we heuristically bias the SAT solver before the above queries to favor high-merit witnesses. Specifically, we leverage the SAT solver's capability to *bias* variables in a given set V toward given target polarities for a particular SAT query. This involves *boosting* the variable decision heuristic scores for each $v \in V$ before the query, followed by *forcing* the variables in V to their target polarities (i.e., whenever $v \in V$ is selected by the decision heuristic, it is assigned its target polarity) [2, 30, 31].

We employ *alternation*: before each SAT query, we bias all inputs in $I^Y \cup I^Z$ to a polarity a, initially set to 1 and flipped to $\neg a$ after each query. We also tested *fixing* the target polarity to 0 or 1, but alternation performed better. Its superiority likely stems from its adaptability both to instances where 1 maximizes the merit and those where 0 is advantageous. Despite its simplicity, our approach achieves remarkable efficiency, enabling the solution of over three times as many instances (Sect. 7).

5.2 Blocking for P-Equivalence

We introduce a new blocking algorithm for P-equivalence, shown in Alg. 5. It can be configured to either `enf` (based on [26]) or `dyn` (novel).

Algorithm 5 BlockMisP(`bucket`, σ).

1: **Input**: `alg` $\in \{$`enf`, `dyn`$\}$
2: **if** `alg` $=$ `enf` or $M_\sigma < i - 3$ **then**
3: Add the following clause to `bucket`: $C := \bigvee x^+_{pq} | p : y_p \in I^Y_{l_\sigma}$ and $q : z_q \in I^Z_{f_\sigma}$
4: **else** ▷ `alg` $=$ `dyn` and $M_\sigma \geq i - 3$
5: **for** every total permutation π from $I^Y_{l_\sigma}$ to $I^Z_{l_\sigma}$ **do**
6: Clause $C := \{\}$ ▷ blocking clause per π
7: **for** every $p : y_p \in I^Y_{l_\sigma}$ **do**
8: Let q be the index of $z_q = \pi(y_p)$
9: $C := C \cup \{\neg x^+_{pq}\}$ ▷ $\sigma(y_p) = \sigma(z_q) = l_\sigma$
10: Add C to `bucket`

`enf` follows `BooM`'s mismatch-blocking approach to P-equivalence, reviewed in Sect. 3.2.1. It creates a single blocking clause C shown in line 3. Specifically, it adds x^+_{pq} to C for every combination of $I^Y_{l_\sigma} := \{y_p \in I^Y | \sigma(y_p) = l_\sigma\}$ and $I^Z_{f_\sigma} := \{z_q \in I^Z | \sigma(z_q) = f_\sigma\}$ (using l_σ and f_σ instead of 0 and 1 in `BooMS`, respectively, because it yielded slightly better results in preliminary experiments).

We observed that, for high-merit witnesses, `enf` is less efficient than our `dyn` algorithm (lines 5–10). `dyn` adds $(i - M_\sigma)!$ clauses for every one of the $(i - M_\sigma)!$ total permutations π from $I^Y_{l_\sigma}$ to $I^Z_{l_\sigma}$, where every such clause blocks $M_\sigma!$ total mismatches. Every clause blocks a (partial) mismatch satisfied by the current witness, with all the clauses together blocking *all* currently satisfied mismatches, including the original π. Since the number of clauses grows super-exponentially as M_σ decreases, Alg. 5 reverts to `enf` whenever $M_\sigma < i - 3$.

For example, let σ be a witness with merit $i-1$, where $f_\sigma = 1$ and $\sigma(y_1) = \sigma(z_1) := 0$ (with the other variables assigned 1, as expected when f_σ is 1 and M_σ is $i-1$). `enf` would block with $x^+_{12} \vee x^+_{13} \vee \ldots \vee x^+_{1i}$, whereas `dyn` would block with the unit clause $\neg x^+_{11}$. Because of the constraint $\sum_{q=1}^{i} x^+_{1q} = 1$ (recall Fig. 2, with x^-_{pq}'s fixed to 0 for P-equivalence), both clauses achieve the same effect as expected, but `dyn` adds a substantially smaller clause.

6 Correctness Proof Outline

This section outlines the correctness proofs of our new algorithms.

Consider a straightforward *picker* classification algorithm. Given a bucket with total mappings, the picker iteratively removes a total mapping π from the bucket and reports it only if it is a match. Soundness (as per Def. 12) and termination are guaranteed by construction.

Although our picker implementation (`EBatP` in Alg. 3) classifies and removes multiple total mappings at once, correctness is still guaranteed as long as the invariants in Fig. 4 hold for each iteration (note that Alg. 3 reports the matches, and only them, by construction).

1. **Termination**: Every examined total mapping π is removed.
2. **Soundness**: Any removed total mapping is a match iff π is a match.

Figure 4 Loop invariants for `EBatP` correctness.

Furthermore, one can easily verify that meeting the match-handling invariants in Fig. 5 and the mismatch-handling invariants in Fig. 6 ensures the `EBatP` loop invariants of termination and soundness in Fig. 4.

Moreover, our match-handling algorithm, presented in Sect. 4.1 as part of `EBatP`, satisfies the invariants in Fig. 5 by construction. The invariants in Fig. 6 are also satisfied by construction by the mismatch-handling procedures, including the witness-extension by generalization and blocking introduced already in `BooMS` (recall Sect. 3), as well as our novel blocking procedure for P-equivalence (Sect. 5.2). Our novel witness-extension procedure for P-equivalence (Sect. 5.1) clearly satisfies them, as it only adjusts the heuristics.

1. Given a match π, StrenMatch returns a match π' such that $\pi' \subseteq \pi$.
2. Given a match π', BlkMatch removes π' and only π' from `bucket`.

Figure 5 Match-handling invariants.

1. Given a witness σ for π, ExtWit returns a witness $\sigma' \subseteq \sigma$ for π.
2. Given a witness σ', BlkMis blocks all the mismatches witnessed by σ' in the given SAT instance, and no additional mappings.

Figure 6 Mismatch-handling invariants.

Similarly, one can easily formulate loop invariants to prove SiftMis correct, assuming the mismatch-handling invariants in Fig. 6. The correctness of `BooMS` and `EBatC` follows directly from that of SiftMis and `EBatP`.

7 Experimental Results

We implemented our new algorithms and the baseline BooMS algorithm within a new open-source all-Boolean-matching tool, EBat[1]. Based on preliminary experiments, we used the IntelSAT SAT solver [32] for all queries, except for UNSAT core extraction on the dual circuit, where CaDiCaL [8] proved more effective.

As the original BooMS in BooM [26, 25] does not function correctly, we use our re-implementation of BooMS as the baseline for comparison.

We adapted benchmarks from the following relevant works: [21], [26], and [15]. All circuits were converted to the AIGER format [9], with multi-output circuits transformed into single-output ones as described below. To manage the complexity of all-Boolean-matching, and because circuits with more than 50 inputs are rarely encountered in our industrial practice, we restricted our selection to circuits with up to 50 inputs. Below, we provide a detailed description of our benchmark selection process, resulting in a total of 388 instances, ranging from 1 to 32,153 gates:

From BooM [26]: We reused the ITC'99 suite [17], also used in BooM experiments [26]. Each benchmark consists of a combinatorial circuit and its optimized counterpart. To focus on a single output, we used *aigsplit* [9] to split the outputs and compared each instance to its optimized version. In total, we considered 9 circuits, resulting in 268 benchmarks.

From ICCAD'23 Boolean matching contest [15]: We selected 3 out of 5 circuits with up to 50 inputs (the other 2 are currently infeasible for all-Boolean matching) and transformed them into single-output circuits by creating a benchmark for every combination of output versus output, resulting in 48 benchmarks.

From [21]: We selected benchmarks originating from the EPFL [4] and ISCAS'85 [13], already transformed by [21] to be single-output by either applying `or`, `xor` or `last` (output) operator. We created six all-Boolean-matching instances from each circuit by comparing: `or` vs. `or`, `or` vs. `xor`, `or` vs. `last`, `xor` vs. `xor`, `xor` vs. `last`, and `last` vs. `last`. In total, we used 12 circuits from [21] (6 from EPFL and 6 from ISCAS'85), resulting in 72 benchmarks.

We conducted experiments on Intel®Xeon® machines (32GB memory, 3GHz CPU) with a 1-minute timeout, reflecting our industrial requirements. We measured the number of *solved* instances (all matches found or absence proven within timeout) and the *PAR-2 score* (solved benchmarks contribute runtime; unsolved ones contribute twice the timeout).

We studied the impact of:

1. *High-level algorithms*: the baseline BooMS vs. our novel EBatP and EBatC.
2. *Blocking schemes for P-equivalence*: the baseline enf vs. the new dyn.
3. *Witness-extension*:
   a. *NP-equivalence*: BTS as in BooMS vs. FTS&MUC [21] vs. our novel BTS&MUC.
   b. *P-equivalence*: none in the baseline BooMS vs. our novel Alternation algorithm and its two simpler versions, Fixed to 1 and Fixed to 0.
4. *Strengthening*: none as in the baseline BooMS vs. our novel strengthening approach.

7.1 The Results

Table 1 summarizes our experimental results. To complement the data in the table, for NP-equivalence, at least one configuration solved 328 benchmarks, with 65 not matching and 2 having only one match. For P-equivalence, the corresponding numbers are 333 solved, with 75 not matching and 6 with one match.

[1] EBat and all the benchmarks are available at https://github.com/yogevshalmon/ebat.

Table 1 Summary of results for NP-equivalence (top) and P-equivalence (bottom) for a 1-minute timeout. Each non-header row corresponds to a configuration of our `EBat` tool. The default configuration in `EBat` is highlighted in light gray, while the baseline configuration from `BooM` is highlighted in dark gray. Other configurations correspond to the `EBat` default with one (and only one) algorithmic feature changed, and with unchanged features appearing faded. The first four columns represent the algorithmic features of each configuration, as indicated by the column titles. The last two columns display the results: the number of solved instances and the PAR-2 score.

NP-Equivalence					
Algorithm	**Blocking**	**Witness-Extension**	**Strengthening**	**Solved**	**PAR-2**
EBatC	default	BTS&MUC	✓	327	686
EBatC	default	FTS&MUC	✓	325	939
EBatP	default	BTS&MUC	✓	322	1252
EBatC	default	BTS	✓	308	2932
EBatC	default	BTS&MUC	–	99	27826
BooMS	default	BTS	–	86	29466

P-Equivalence					
Algorithm	**Blocking**	**Witness-Extension**	**Strengthening**	**Solved**	**PAR-2**
EBatC	dyn	Alternation	✓	329	871
EBatC	enf	Alternation	✓	326	1324
EBatP	dyn	Alternation	✓	308	3467
EBatC	dyn	Fixed to 0	✓	303	4325
EBatC	dyn	Fixed to 1	✓	301	4465
EBatC	dyn	Alternation	–	115	26232
EBatC	dyn	–	✓	106	27460
BooMS	enf	–	–	90	29348

Overall, the default configuration of our new all-Boolean-matching tool, `EBat`, significantly outperforms the baseline `BooMS` algorithm for both NP- and P-equivalence, solving 3 to 4 times more benchmarks within our 1-minute timeout. Furthermore, on instances completed by both `BooMS` and our default configuration of `EBat`, `BooMS` issues, on average, 13,000 times more SAT queries for NP-equivalence and 1,500 times more SAT queries for P-equivalence.

In an additional experiment, we confirmed that this advantage persists with a longer 10-minute timeout: for P-equivalence, `BooMS` solves 95 instances, while `EBat` solves 339; for NP-equivalence, `BooMS` solves 87, compared to 335 by `EBat`.

Table 1 shows that, for NP-equivalence, strengthening is the key factor in improving performance. Also, our default generalization approach, `BTS&MUC`, outperforms both `BTS`, applied by `BooM`, and `FTS&MUC` from [21]. Finally, comparing the high-level algorithms, `EBatC` outperforms `EBatP`, while switching to `BooMS` in the default configuration is impossible, as `BooMS` cannot apply strengthening by construction.

For P-equivalence, our novel witness-extension procedure contributes the most, closely followed by strengthening; together, they enable solving 329 instances, compared to only 106–115 without either. Additionally, in witness-extension, alternation outperforms fixing to 0 or 1. Also, `EBatC` surpasses `EBatP`, and our `dyn` blocking scheme outperforms `BooM`'s `enf`. Additionally, even though `EBatC` outperforms `EBatP` on the same default configuration of `EBat`, for P-Equivalence, one instance is solved uniquely by `EBatP` (for NP-Equivalence, all instances solved by `EBatP` are also solved by `EBatC`).

8 Conclusion

Motivated by the industrial need for automated, timing-aware library mapping, we presented the first dedicated study on enumerating all matches between two Boolean circuits (all-Boolean-matching). We introduced novel algorithms and implemented them from scratch in EBat, the only open-source tool for this problem. EBat solves 3 to 4 times more benchmarks than the state-of-the-art algorithm BooMS, within both the application-driven 1-minute timeout and a 10-minute timeout for NP- and P-equivalence.

In future work, we plan to extend EBat to handle multiple outputs and sequential circuits, and implement circuit preprocessing techniques [12, 36, 5] to further improve its performance.

References

1 Afshin Abdollahi. Signature based boolean matching in the presence of don't cares. In Limor Fix, editor, *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 642–647. ACM, 2008. doi:10.1145/1391469.1391635.

2 Sabih Agbaria, Dan Carmi, Orly Cohen, Dmitry Korchemny, Michael Lifshits, and Alexander Nadel. Sat-based semiformal verification of hardware. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 25–32. IEEE, 2010. URL: https://ieeexplore.ieee.org/document/5770929/.

3 Manindra Agrawal and Thomas Thierauf. The boolean isomorphism problem. In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 422–430. IEEE Computer Society, 1996. doi:10.1109/SFCS.1996.548501.

4 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.

5 Daniil Averkov, Tatiana Belova, Gregory Emdin, Mikhail Goncharov, Viktoriia Krivogornitsyna, Alexander S. Kulikov, Fedor Kurmazov, Daniil Levtsov, Georgie Levtsov, Vsevolod Vaskin, and Aleksey Vorobiev. Cirbo: A new tool for boolean circuit analysis and synthesis. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(11):11105–11112, April 2025. doi:10.1609/aaai.v39i11.33207.

6 Luca Benini and Giovanni De Micheli. A survey of boolean matching techniques for library binding. *ACM Transactions on Design Automation of Electronic Systems*, 2(3), 1997. doi:10.1145/264995.264996.

7 Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 285–301. Springer, 2014. doi:10.1007/978-3-319-09284-3_22.

8 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

9 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.

10 Bernd Borchert, Desh Ranjan, and Frank Stephan. On the computational complexity of some classical equivalence relations on boolean functions. *Theory Comput. Syst.*, 31(6):679–693, 1998. doi:10.1007/S002240000109.

11 Daniel Brand. Verification of large synthesized designs. In Michael R. Lightner and Jochen A. G. Jess, editors, *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, pages 534–537. IEEE Computer Society / ACM, 1993. `doi:10.1109/ICCAD.1993.580110`.

12 Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010. `doi:10.1007/978-3-642-14295-6_5`.

13 Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator. In *Fortran. ISCAS'85*, 1985.

14 Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental formal verification of hardware. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 135–143. FMCAD Inc., 2011. URL: `http://dl.acm.org/citation.cfm?id=2157676`.

15 Chung-Han Chou, Chih-Jen Jacky Hsu, Chi-An Rocky Wu, Kuan-Hua Tu, and Kei-Yong Khoo. Invited paper: 2023 iccad cad contest problem a: Multi-bit large-scale boolean matching. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–4, 2023. `doi:10.1109/ICCAD57390.2023.10323797`.

16 Jason Cong and Yean-Yow Hwang. Boolean matching for lut-based logic blocks with applications toarchitecture evaluation and technology mapping. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 20(9):1077–1090, 2001. `doi:10.1109/43.945303`.

17 F. Corno, M.S. Reorda, and G. Squillero. Rt-level itc'99 benchmarks and first atpg results. *IEEE Design & Test of Computers*, 17(3):44–53, 2000. `doi:10.1109/54.867894`.

18 Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 36–41. Springer, 2006. `doi:10.1007/11814948_5`.

19 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. URL: `http://dl.acm.org/citation.cfm?id=2157675`.

20 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT, Proceedings*, 2003. `doi:10.1007/978-3-540-24605-3_37`.

21 Dror Fried, Alexander Nadel, Roberto Sebastiani, and Yogev Shalmon. Entailing generalization boosts enumeration. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPICS.SAT.2024.13`.

22 Shao-Lun Huang, Wei-Hsun Lin, Po-Kai Huang, and Chung-Yang Huang. Match and replace: A functional ECO engine for multierror circuit rectification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 32(3):467–478, 2013. `doi:10.1109/TCAD.2012.2226456`.

23 Zheng Huang, Lingli Wang, Yakov Nasikovskiy, and Alan Mishchenko. Fast boolean matching based on NPN classification. In *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013*, pages 310–313. IEEE, 2013. `doi:10.1109/FPT.2013.6718374`.

24 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. `doi:10.1147/SJ.83.0178`.

25 Chih Fan Lai, Jie Hong R. Jiang, and Kuo Hua Wang. Boolean matching of function vectors with strengthened learning. In *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2010. doi:10.1109/ICCAD.2010.5654215.

26 Chih Fan Lai, Jie Hong R. Jiang, and Kuo Hua Wang. BooM: A decision procedure for Boolean matching with abstraction and dynamic learning. In *Proceedings - Design Automation Conference*, 2010. doi:10.1145/1837274.1837398.

27 E. J. McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956. doi:10.1002/j.1538-7305.1956.tb03835.x.

28 Alan Mishchenko, Supratik Chatterjee, Robert Brayton, Xiangjian Wang, and Tony Kam. Technology mapping with boolean matching, supergates and choices. Technical report, EECS Department, University of California, Berkeley, March 2005.

29 Janett Mohnke, Paul Molitor, and Sharad Malik. Application of bdds in boolean matching techniques for formal logic combinational verification. *Int. J. Softw. Tools Technol. Transf.*, 3(2):207–216, 2001. doi:10.1007/S100090100039.

30 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. doi:10.23919/FMCAD.2019.8894273.

31 Alexander Nadel. Polarity and variable selection heuristics for sat-based anytime maxsat. *J. Satisf. Boolean Model. Comput.*, 12(1):17–22, 2020. doi:10.3233/SAT-200126.

32 Alexander Nadel. Introducing intel(r) SAT solver. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.8.

33 Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In *Theory and Applications of Satisfiability Testing - SAT, Proceedings*, 2012. doi:10.1007/978-3-642-31612-8_19.

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

35 W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952. doi:10.1080/00029890.1952.11988183.

36 Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. eslim: Circuit minimization with SAT based local improvement. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 23:1–23:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.SAT.2024.23.

37 J. Paul Roth, Willard G. Bouricius, and Peter R. Schneider. Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits. *IEEE Trans. Electron. Comput.*, 16(5):567–580, 1967. doi:10.1109/PGEC.1967.264743.

38 Roberto Sebastiani. Are you satisfied by this partial assignment? *CoRR*, abs/2003.04225, 2020. arXiv:2003.04225.

39 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. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 42(4):1351–1364, 2023. doi:10.1109/TCAD.2022.3198260.

40 Pawel Swierczynski, Marc Fyrbiak, Christof Paar, Christophe Huriaux, and Russell Tessier. Protecting against cryptographic trojans in fpgas. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*, pages 151–154. IEEE Computer Society, 2015. doi:10.1109/FCCM.2015.55.

41 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.

42 Kuo-Hua Wang, Chung-Ming Chan, and Jung-Chang Liu. Simulation and sat-based boolean matching for large boolean networks. In *Proceedings of the 46th Design Automation Conference,*

DAC 2009, San Francisco, CA, USA, July 26-31, 2009, pages 396–401. ACM, 2009. doi:10.1145/1629911.1630016.

43 Chaofan Yu, Lingli Wang, Chun Zhang, Yu Hu, and Lei He. Fast filter-based boolean matchers. *IEEE Embed. Syst. Lett.*, 5(4):65–68, 2013. doi:10.1109/LES.2013.2280582.

44 Juling Zhang, Guowu Yang, William N. N. Hung, and Jinzhao Wu. A canonical-based NPN boolean matching algorithm utilizing boolean difference and cofactor signature. *IEEE Access*, 5:27777–27785, 2017. doi:10.1109/ACCESS.2017.2778338.