Multiple Definitions from a Single Resolution Proof (Extended Abstract)

Friedrich Slivovsky

School of Computer Science & Informatics, University of Liverpool, Liverpool, UK

Abstract

Propositional formulas frequently contain implicitly defined variables, and extracting their explicit definitions is often useful. Following the standard proof of Beth's definability theorem, these definitions can be obtained as Craig interpolants of formulas expressing implicit definability. In practice, one calls a SAT solver for each defined variable and extracts an interpolant from the resulting resolution proof. We propose an alternative approach where a single resolution refutation serves as a witness of definability for a set of variables. We then present a simple modification of standard interpolation systems that constructs a multi-output circuit representing all definitions in one pass over this proof. The running time and circuit size are $O(nm)$, where n is the number of defined variables and m the proof size. Preliminary experiments on Boolean functional synthesis benchmarks show that the single refutation witnessing definability is produced at least as quickly as the sequence of proofs of a per-variable baseline, but that the multi-output circuits are typically larger, sometimes substantially so on instances with many definitions. Extraction that asymptotically improves upon $O(nm)$ is identified as the main open question.

Keywords

Definability, Synthesis, Interpolation Systems, Resolution

1. Introduction

Several problems in formal methods ask for Boolean functions that satisfy a propositional specification. Boolean Functional Synthesis [1], Quantified Boolean Formulas [2], and Dependency QBF [3] can all be viewed as such synthesis problems, differing in the dependencies that the functions are allowed to have. Identifying *implicitly defined* variables—those whose value is uniquely determined by the specification—and computing their explicit definitions can simplify the synthesis problem, and sometimes solve it outright.

By Beth's definability theorem, every variable that is implicitly defined in a formula has an explicit definition, and that definition can be obtained as a Craig interpolant of a formula expressing implicit definability [4]. Modern CDCL SAT solvers can produce refutations of these formulas in standard proof formats such as RUP [5, 6] and LRAT [7]. In principle LRAT is stronger than resolution, but in practice—after disabling a handful of preprocessing techniques at negligible cost—the proofs CDCL solvers emit are succinct resolution refutations, allowing the use of well-known constructions for extracting interpolants [8, 9, 10, 11]. This yields a practical pipeline for definition extraction: invoke a SAT solver on the implicit definability formula and compute an interpolant from the resolution proof if the formula is unsatisfiable.

This procedure has been used to good effect in several recent tools [12, 13, 14, 15]. One of its limitations is that it requires one SAT call per (defined) variable. On instances with many defined variables, which occur in Boolean functional synthesis, the cumulative cost of these calls dominates the running time of the preprocessing phase, even when techniques such as incremental SAT solving are used.

We propose an alternative scheme that obtains explicit definitions for an entire *set* of defined variables with a *single* SAT call. The solver returns a resolution refutation of a propositional formula expressing joint implicit definability of all n target variables. The formula for an individual variable is recovered

CI-BD-SOQE 2026: Workshop on Craig Interpolation, Beth Definability, and Second-Order Quantifier Elimination, at FLoC 2026: The 9th Federated Logic Conference, July 24–25, 2026, Lisbon, Portugal

f.slivovsky@liverpool.ac.uk (F. Slivovsky)

by applying a partial assignment. We show how to modify an interpolation system to compute an interpolant under such a partial assignment, and use this generalisation to extract all n definitions from the refutation in a single pass. The running time and the size of the resulting multi-output circuit are both $O(nm)$, where m is the proof size.

We implemented and evaluated the algorithm on a standard benchmark set for Boolean functional synthesis [1]. A naive implementation produced highly redundant circuits, but a handful of structural observations brought it down to a practically viable form. Compared with an optimised per-variable extractor that used incremental SAT solving, our implementation produced refutations at comparable or better speed, but the multi-output circuits it constructed were typically larger. This circuit-size gap dominated the runtime, especially on instances with many definitions.

The remainder of the paper is organised as follows. Section 2 establishes notation and recalls relevant background. Section 3 introduces the interpolation system that computes interpolants under partial assignments, proves it correct, and uses it to extract a vector of definitions from a single refutation. Section 4 describes our implementation and reports experimental results. Section 5 concludes with a discussion and identifies the main open challenges.

2. Preliminaries

Propositional Formulas. Let $\mathcal{V}$ be a countably infinite set of propositional variables. A *literal* is a variable v or its negation $\neg v$; the negation of a literal ℓ is written $\bar{\ell}$. A *clause* $C = \ell_1 \vee \cdots \vee \ell_k$ is a finite set of literals, and the empty clause is denoted $\bot$. A formula in *conjunctive normal form* (CNF) is a finite set of clauses, identified with their conjunction. We write $var(\varphi)$ for the set of variables occurring in φ and $\varphi \models \psi$ for the entailment relation. For a clause C and a set of variables S, the *restriction* $C|_S$ is the disjunction of those literals of C whose variables lie in S.

A *partial assignment* is a partial function $\sigma : \mathcal{V} \rightharpoonup \{\bot, \top\}$; its domain $dom(\sigma)$ is the set of variables on which σ is defined, and σ is a *(total) assignment* when $dom(\sigma) \supseteq var(\varphi)$ for the formula φ under consideration. We identify a partial assignment with the set of literals it satisfies: σ assigns $\top$ to a variable v iff $v \in \sigma$, and $\bot$ iff $\bar{v} \in \sigma$. Given a formula F and a partial assignment σ, we write $F[\sigma]$ for the formula obtained by replacing every occurrence of each variable $v \in dom(\sigma)$ in F by $\sigma(v)$ and simplifying the resulting Boolean constants in the standard way. For a CNF formula F, $F[\sigma]$ is again in CNF: each clause $C \in F$ is removed if it contains a literal satisfied by σ, and otherwise has its σ-falsified literals dropped.

Resolution. The *resolution rule* derives a clause $C \vee D$ from $C \vee p$ and $D \vee \neg p$; the variable p is the *pivot* of the step. A *resolution refutation* of a CNF φ is a sequence $C_1, \ldots, C_m = \bot$ in which each C_i is either a clause of φ or the result of resolving two earlier clauses; φ is unsatisfiable if and only if such a refutation exists. A refutation has a natural directed acyclic graph (DAG) structure with the input clauses as sources and each derived clause connected to its two parents; the refutation is *tree-like* (or *linear*) when this DAG is a tree, that is, when each derived clause is consumed at most once.

Boolean Circuits. A *Boolean circuit* over inputs X is a directed acyclic graph whose sources are labelled with the variables in X and whose internal vertices (*gates*) compute Boolean functions of their fan-in. We work with *De Morgan circuits*, whose gates are binary $\wedge$, binary $\vee$, unary $\neg$, and the nullary constants $\bot$ and $\top$. The *size* of a circuit is its number of internal gates. A *multi-output* circuit designates several gates as outputs and computes a tuple of Boolean functions of the inputs, sharing all gates that are reachable from more than one output.

Interpolation. Let A and B be CNFs with $A \wedge B \models \bot$. A *Craig interpolant* of (A, B) is a Boolean circuit I over the shared variables $var(A) \cap var(B)$ such that $A \models I$ and $I \wedge B \models \bot$. A Craig interpolant always exists, and several *interpolation systems* construct one as a Boolean circuit by traversing a resolution refutation of $A \wedge B$ [8, 9, 10, 11].

We use the interpolation system of McMillan [11]. Every clause C in a resolution refutation of $A \wedge B$ is annotated with a *partial interpolant* $itp(C)$, a circuit over $var(A) \cap var(B)$, defined inductively:

- For an input clause $C \in A$, $itp(C) = C|_{var(B)}$. For $C \in B$, $itp(C) = \top$.
- If C is derived from C_1 and C_2 by resolving on a pivot p, then

$$itp(C) = \begin{cases} itp(C_1) \vee itp(C_2) & \text{if } p \in var(A) \setminus var(B), \\ itp(C_1) \wedge itp(C_2) & \text{otherwise.} \end{cases}$$

Every partial interpolant satisfies the invariant

$$A \models itp(C) \vee C|_{var(A)\setminus var(B)} \qquad \text{and} \qquad B \wedge itp(C) \models C|_{var(B)}. \tag{1}$$

For the empty clause both restrictions vanish, so $itp(\bot)$ is a Craig interpolant of (A, B) [11].

Beth's Definability Theorem. A variable y is *implicitly defined* by a set of variables $X \subseteq var(\varphi) \setminus \{y\}$ in a CNF φ if every two satisfying assignments of φ that agree on X also agree on y. An *explicit definition* of y in terms of X is a Boolean circuit f over X with $\varphi \models y \leftrightarrow f(X)$. Beth's theorem [16] states that the two notions coincide: y is implicitly defined by X in φ if and only if it has an explicit definition in terms of X.

Let φ' denote φ with every variable $v \in var(\varphi) \setminus X$ renamed to a fresh copy v'. Then y is implicitly defined by X in φ if and only if the formula

$$\varphi \ \wedge \ y \ \wedge \ \varphi' \ \wedge \ \neg y' \tag{2}$$

is unsatisfiable, in which case any Craig interpolant of $(\varphi \wedge y, \varphi' \wedge \neg y')$ is an explicit definition of y in terms of X [4].

3. Obtaining Multiple Definitions from a Single Resolution Proof

Let $\varphi = \varphi(X, Y, Z)$ be a propositional formula, where $Y = \{y_1, \ldots, y_n\}$. Then every y_i is defined by X in φ if and only if the following formula is unsatisfiable:

$$\underbrace{\varphi(X, Y, Z)}_{A} \wedge \underbrace{\varphi(X, Y', Z')}_{B} \wedge \underbrace{\bigwedge_{i=1}^{n} \big((y_i \vee y_i' \vee e_i) \wedge (\neg y_i \vee \neg y_i' \vee e_i)\big) \wedge \bigvee_{i=1}^{n} \neg e_i}_{G} \tag{3}$$

The final clause is satisfied only if one of the e_i is false, which requires y_i and y_i' to take on different values. Accordingly, a satisfying assignment exists if and only if there is a y_i that is not defined by X in φ. A formula equivalent to (2) representing definability of $y_i \in Y$ can be obtained by applying the partial assignment $\sigma_i = \{y_i, \neg y_i', \neg e_i\} \cup \{e_j : j \neq i\}$ to (3): group A simplifies to $\varphi \wedge y_i$, group B becomes $\varphi' \wedge \neg y_i'$, and all clauses in group G are satisfied. In the next subsection, we show how to modify an interpolation system to compute interpolants from refutations of formulas of this shape under a suitable partial assignment.

3.1. Computing Interpolants under Partial Assignments

Let $A \wedge B \wedge G$ be an unsatisfiable propositional formula and σ a partial assignment of its variables such that $dom(\sigma) \cap var(A) \cap var(B) = \emptyset$ and $\sigma \models G$. Define $\sigma_A = \sigma|_{var(A)}$, $\sigma_B = \sigma|_{var(B)}$. Then $A \wedge \sigma_A \wedge B \wedge \sigma_B$ is unsatisfiable, and there exists a Craig interpolant of $(A \wedge \sigma_A,\ B \wedge \sigma_B)$ over $var(A) \cap var(B)$. We modify McMillan's interpolation system (see Section 2) to compute such an interpolant from a resolution refutation of $A \wedge B \wedge G$, given σ.

Definition of Partial Interpolants. Each clause C in the refutation is annotated with a partial interpolant $itp(C)$ that may be *undefined* (written $\bullet$). Input clauses are labelled as in McMillan's system, except for clauses from G:

- $C \in A$: $itp(C) = C|_{var(B)}$.
- $C \in B$: $itp(C) = \top$.
- $C \in G$: $itp(C) = \bullet$.

For a derived clause C obtained by resolving C_1 (containing p) with C_2 (containing $\bar{p}$):

1. If $itp(C_1) = \bullet$, set $itp(C) = itp(C_2)$ (and symmetrically if $itp(C_2) = \bullet$).
2. Otherwise, if $p \in dom(\sigma)$, set

$$itp(C) = \begin{cases} itp(C_2) & \text{if } \sigma(p) = \top \quad (\bar{p} \text{ is falsified}), \\ itp(C_1) & \text{if } \sigma(p) = \bot \quad (p \text{ is falsified}). \end{cases}$$

3. Otherwise (both defined and $p \notin dom(\sigma)$), apply McMillan's rule:

$$itp(C) = \begin{cases} itp(C_1) \vee itp(C_2) & \text{if } p \in var(A) \setminus var(B), \\ itp(C_1) \wedge itp(C_2) & \text{otherwise.} \end{cases}$$

Because G is satisfiable, the refutation must use at least one clause from $A \cup B$, and the inheritance rule preserves definedness of itp along any path through a clause with defined partial interpolant. Hence $itp(\bot) \neq \bullet$. The system satisfies the following invariant.

Proposition 1. *For every clause C in the refutation:*

- *if $itp(C) = \bullet$, then σ satisfies C;*
- *if $itp(C) \neq \bullet$, then*

$$A \wedge \sigma_A \models itp(C) \vee C[\sigma]|_{var(A)\setminus var(B)}, \qquad B \wedge \sigma_B \wedge itp(C) \models C[\sigma]|_{var(B)}. \tag{4}$$

In particular, $itp(\bot)$ is a Craig interpolant of $(A \wedge \sigma_A,\ B \wedge \sigma_B)$.

Proof. By induction on the position of C in the refutation.

1. C is an input clause. We consider three cases:
   a) $C \in A$. Then $var(C) \subseteq var(A)$ and $itp(C) = C|_{var(B)} = C|_{var(A)\cap var(B)}$. Since $dom(\sigma) \cap var(A) \cap var(B) = \emptyset$, σ leaves $itp(C)$ untouched and $itp(C) = C[\sigma]|_{var(B)}$, giving the right invariant. For the left invariant, $A \models C = C|_{var(A)\cap var(B)} \vee C|_{var(A)\setminus var(B)}$, and σ_A either satisfies C (in which case $C[\sigma] = \top$) or removes some σ-falsified literals from $C|_{var(A)\setminus var(B)}$; either way $A \wedge \sigma_A \models itp(C) \vee C[\sigma]|_{var(A)\setminus var(B)}$.
   b) $C \in B$. Then $var(C) \subseteq var(B)$ and $itp(C) = \top$. The left invariant is trivial. For the right invariant, $B \models C$, hence $B \wedge \sigma_B \models C[\sigma]$, and $C[\sigma]|_{var(B)} = C[\sigma]$.
   c) $C \in G$. Then $itp(C) = \bullet$, and $\sigma \models G$ by hypothesis, so σ satisfies C.
2. C is the resolvent of $C_1 \vee p$ and $\neg p \vee C_2$ with $itp(C_1 \vee p) = I_1$ and $itp(\neg p \vee C_2) = I_2$.
   a) $I_1 = \bullet$ and $I_2 \neq \bullet$ (the symmetric case is analogous). Then $itp(C) = I_2$ by the inheritance rule. By the induction hypothesis σ satisfies $C_1 \vee p$ via some literal ℓ.
      - If $\ell \neq p$, then $\ell \in C_1 \subseteq C$, so σ satisfies C and $C[\sigma] = \top$, making both sides of (4) trivial.
      - If $\ell = p$, then $\sigma(p) = \top$ and $\neg p$ is σ-falsified, so $(\neg p \vee C_2)[\sigma] = C_2[\sigma]$ is a sub-clause of $C[\sigma]$. The inductive invariant for $\neg p \vee C_2$ transfers to C via $itp(C) = I_2$, since $C_2[\sigma]|_S \models C[\sigma]|_S$ for every S.
   b) $I_1 = I_2 = \bullet$. Then $itp(C) = \bullet$. By the induction hypothesis σ satisfies clauses $C_1 \vee p$ and $\neg p \vee C_2$, so it must also satisfy their resolvent C.

c) $I_1 \neq \bullet$ and $I_2 \neq \bullet$.

i. $p \in dom(\sigma)$. Without loss of generality $\sigma(p) = \top$ (the other case is symmetric); then $itp(C) = I_2$. As in case (a) with $\ell = p$, $\neg p$ is σ-falsified, $(\neg p \vee C_2)[\sigma] = C_2[\sigma]$ is a sub-clause of $C[\sigma]$, and the inductive invariant for $\neg p \vee C_2$ transfers to C.

ii. $p \notin dom(\sigma)$. McMillan's rule applies, so $itp(C) = I_1 \vee I_2$ if $p \in var(A) \setminus var(B)$ and $itp(C) = I_1 \wedge I_2$ otherwise. The argument follows McMillan's standard correctness with $A \wedge \sigma_A$ and $B \wedge \sigma_B$ in place of A and B, and $C[\sigma]$ in place of C.

□

3.2. Extracting All n Definitions from One Refutation

Applying $\sigma_i = \{y_i, \neg y_i', \neg e_i\} \cup \{e_j : j \neq i\}$ to (3) produces a formula equivalent to the implicit definability encoding (2) for y_i. Proposition 1 therefore guarantees that the generalised interpolation system, applied to any resolution refutation of (3) under σ_i, returns a Craig interpolant of these two formulas over X, which is an explicit definition of y_i by Beth's theorem.

To obtain all n definitions from one refutation, we run the system n times in parallel. Every clause C carries a vector $(itp_1(C), \ldots, itp_n(C))$ of partial interpolants, one per target, initialised by the leaf rules and updated at each resolution step by the rule for the corresponding σ_i. The interpolant $itp_i(\bot)$ at the empty clause is the desired definition of y_i. For a refutation of length m, the running time and output size is $O(nm)$.

4. Implementation and Experiments

We implemented the definition extraction algorithm in C++, using CADICAL [17] for proof generation and ABC [18] to represent definition circuits as And-Inverter-Graphs (AIGs). For evaluation we use a standard benchmark set for Boolean functional synthesis comprising 594 specifications [19]. Each specification is a propositional formula $\varphi(X, Y)$ over *input* variables X and *output* variables Y. Given a specification together with a set $\{y_1, \ldots, y_n\} \subseteq Y$ of outputs that are implicitly defined by X, our implementation generates a resolution refutation of the corresponding joint definability formula (3) and runs the interpolation algorithm to extract their definitions.

A naive implementation that computes the n partial interpolants independently scales linearly with the number of defined variables and is not practically viable. A *shared-label* representation avoids most of this overhead by exploiting that, at almost every resolution step, the rule for itp_i does not depend on i. The pivots assigned by at least one σ_i are exactly the variables y_j, y_j', e_j for $j = 1, \ldots, n$. At any other pivot, no σ_i assigns the pivot, so rule 3 of the generalised interpolation system applies uniformly across all target variables and the entire vector evolves identically. At a pivot of the form y_j or y_j', only target j uses an assigned-pivot rule, while every other target still uses rule 3 with the same operation. At a pivot of the form e_j, every σ_i assigns the pivot, but the falsified-side premise is the same for every $i \neq j$, so target j takes one premise's partial interpolant and all remaining targets take the other. In every case at most one target deviates from the rule shared by all others.

We therefore store each clause as a single *shared* label together with a (typically small) collection of per-target *overrides*: for selected target indices, a per-target label that supersedes the shared one. The effective partial interpolant $itp_i(C)$ is the override recorded at target i if there is one, and the shared label otherwise. Input clauses carry no overrides. At a resolution step on a pivot not assigned by any σ_i, the new shared label is the uniform rule applied to the premises' shared labels, and each existing override is updated by applying the same rule to the premises' value for the corresponding target. At a resolution step on a pivot assigned by σ_j, the new shared label is updated by the rule for indices $i \neq j$, and the resolvent records a new override at target j computed by the rule for j. The number of overrides at a clause is bounded by the number of pivots assigned by some σ_i encountered along its derivation, and is often small in practice. Structural sharing across the multi-output circuit produced by the interpolation system then yields a circuit far smaller than the union of n independent definitions.

A second improvement avoids computing labels that do not contribute to any final interpolant. If every path from a clause to the empty clause runs through a resolution step on some pivot p assigned by σ_i at which the *other* premise's partial interpolant is selected, then itp_i at that clause is never read and need not be computed. We identify these unneeded labels by a leaves-to-root traversal of the refutation that records which labels would be computed, followed by a top-down pass that restricts attention to those actually needed at the empty clause. We refer to this implementation as SINGLE-PROOF, since it extracts all definitions from a single refutation of the implicit definability formula for Y as a whole.

We compare it against a PER-VARIABLE baseline that takes the same defined set $\{y_1, \ldots, y_n\}$ as input and processes the variables in order: for each y_i it generates a resolution refutation of the implicit definability formula for y_i in terms of $X \cup \{y_1, \ldots, y_{i-1}\}$ and interpolates the definition from this refutation, then adds y_i to the set of defining (shared) variables for the next iteration. This enables sharing of subcircuits, and in combination with incremental SAT solving, the individual SAT calls are fast.

To isolate the cost of extracting the definitions from the cost of identifying the defined subset of Y, we first ran an independent detection pass with a 1,800 second budget on each of the 594 specifications, recording the resulting defined set $\{y_1, \ldots, y_n\}$. Detection finished within this budget on 582 instances. The 12 that timed out all have between 13,000 and 718,000 output variables. We then fed the recorded defined sets to both extraction tools so that detection time does not factor into the comparison. The 12 instances on which detection itself timed out count as failures for both tools.

We ran both versions of definition extraction with a 300 second timeout and an 8 GB memory limit, recording wall-clock time and the size of the resulting multi-output circuit (number of AIG gates). PER-VARIABLE solved 568 instances within the timeout and SINGLE-PROOF solved 551, with the 551 solved by SINGLE-PROOF a strict subset of those solved by PER-VARIABLE. Across the 551 instances solved by both, the median wall-clock time of SINGLE-PROOF is 1.7× that of PER-VARIABLE, and its median circuit size is 2.2× larger. Per-instance comparisons are plotted in Figure 1.

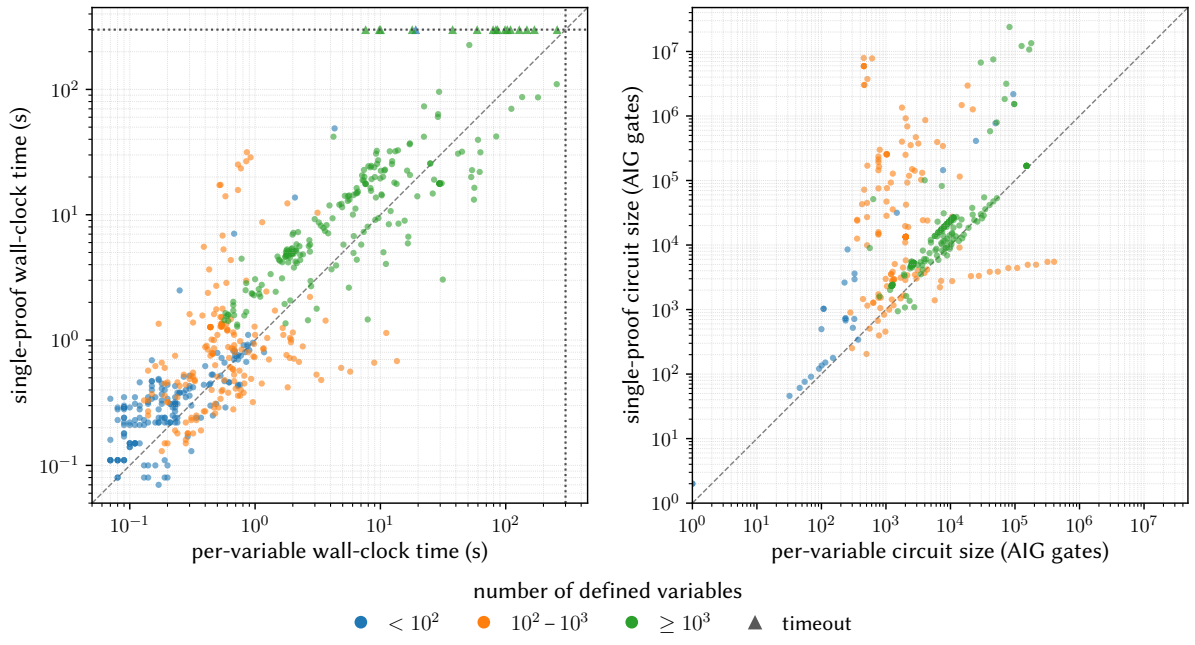

Figure 1: Per-instance comparison of SINGLE-PROOF against PER-VARIABLE on the 594 Boolean functional synthesis benchmarks, with a 300 second timeout. **Left**: wall-clock time. The dotted lines mark the timeout. **Right**: AIG sizes of the multi-output definition circuit on the 366 instances solved by both that have at least one defined variable. Both axes are logarithmic.

Figure 1 shows that SINGLE-PROOF generally produces larger circuits than PER-VARIABLE, in some cases by more than an order of magnitude. The gap widens with the number of implicitly defined

variables, and on the largest such instances SINGLE-PROOF times out. On instances with few defined variables the two configurations are comparable in both wall-clock time and AIG size.

To further isolate the cost of computing the refutation from the cost of building the circuit, we re-ran both tools with circuit construction disabled. In this proof-only mode, SINGLE-PROOF solved 579 instances and PER-VARIABLE 568, with 568 instances solved by both. SINGLE-PROOF additionally solved 11 instances on which PER-VARIABLE timed out. On the 568 instances solved by both, the median wall-clock time of SINGLE-PROOF is $0.74\times$ that of PER-VARIABLE—that is, SINGLE-PROOF is in fact faster than PER-VARIABLE at producing the refutation, and the runtime gap reported above is attributable to interpolation rather than to the SAT calls. A per-instance comparison is shown in Figure 2. On

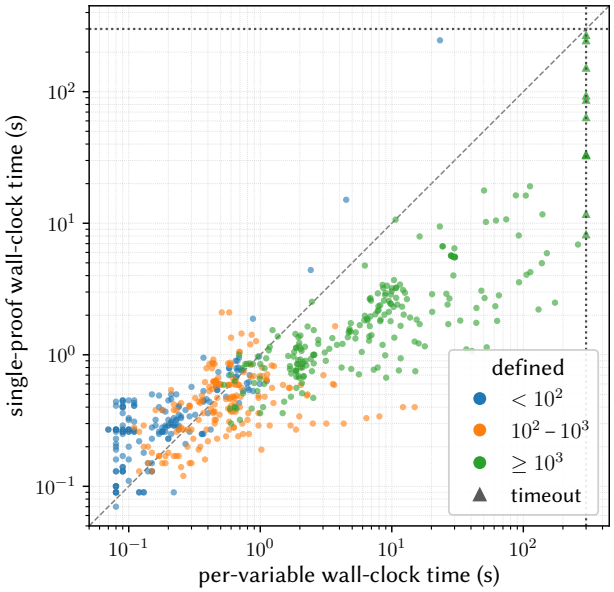

Figure 2: Per-instance wall-clock time when both tools are run in proof-only mode (no circuit extraction), with a 300 second timeout. The dotted lines mark the timeout.

instances with many implicitly defined variables—precisely those where SINGLE-PROOF produces the largest circuits in Figure 1—the joint refutation is obtained as quickly as or faster than the sequence of per-variable proofs. A more efficient extraction would let SINGLE-PROOF convert this advantage into an overall speedup.

5. Discussion

In our experiments, single-proof extraction produced larger circuits than the per-variable approach, while the underlying refutations were small and generated quickly. The bottleneck therefore lies in definition extraction, and improving its $O(nm)$ bound (where n is the number of defined variables and m the size of the refutation) is the main open challenge. Is $O(m \log m + n)$ or even $O(m + n)$ achievable?

One caveat to the runtime comparison is that PER-VARIABLE interleaves detection and extraction in a single SAT call, so its detection cost is inseparable from extraction, whereas SINGLE-PROOF has a distinct detection stage that computes the defined output variables before producing the refutation. The runtimes in Section 4 exclude detection for both tools, and accounting for it would shift the comparison slightly in favour of PER-VARIABLE. Identifying the set of outputs implicitly defined by the inputs is an interesting question in its own right [20].

Declaration on Generative AI

During the preparation of this work, the author used Claude (Anthropic) in order to: Drafting content, Paraphrase and reword, Improve writing style. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] S. Akshay, S. Chakraborty, S. Shah, Tractable representations for boolean functional synthesis, Ann. Math. Artif. Intell. 92 (2024) 1051–1096. doi:10.1007/S10472-023-09907-5.

[2] O. Beyersdorff, M. Janota, F. Lonsing, M. Seidl, Quantified boolean formulas, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, IOS Press, 2021, pp. 1177–1221. doi:10.3233/FAIA201015.

[3] C. Scholl, R. Wimmer, Dependency quantified boolean formulas: An overview of solution methods and applications - extended abstract, in: O. Beyersdorff, C. M. Wintersteiger (Eds.), Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings, Lecture Notes in Computer Science, Springer, 2018, pp. 3–16. doi:10.1007/978-3-319-94144-8_1.

[4] P. Koopmann, C. Wernhard, F. Wolter, Interpolation in classical propositional logic, CoRR abs/2508.11449 (2025). arXiv:2508.11449.

[5] E. I. Goldberg, Y. Novikov, Verification of proofs of unsatisfiability for CNF formulas, in: Design, Automation and Test in Europe Conference and Exposition, DATE 2003, 2003, pp. 10886–10891. doi:10.1109/DATE.2003.1253775.

[6] M. Heule, W. A. H. Jr., N. Wetzler, Trimming while checking clausal proofs, in: Formal Methods in Computer-Aided Design, FMCAD 2013, 2013, pp. 181–188.

[7] L. Cruz-Filipe, M. J. H. Heule, W. A. H. Jr., M. Kaufmann, P. Schneider-Kamp, Efficient certified RAT verification, in: Automated Deduction — CADE 26, volume 10395 of *Lecture Notes in Computer Science*, 2017, pp. 220–236. doi:10.1007/978-3-319-63046-5_14.

[8] G. Huang, Constructing Craig interpolation formulas, in: Computing and Combinatorics, First Annual International Conference, COCOON '95, volume 959 of *Lecture Notes in Computer Science*, 1995, pp. 181–190. doi:10.1007/BFb0030832.

[9] J. Krajíček, Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic, Journal of Symbolic Logic 62 (1997) 457–486. doi:10.2307/2275541.

[10] P. Pudlák, Lower bounds for resolution and cutting plane proofs and monotone computations, Journal of Symbolic Logic 62 (1997) 981–998. doi:10.2307/2275583.

[11] K. L. McMillan, Interpolation and SAT-based model checking, in: Computer Aided Verification, 15th International Conference, CAV 2003, volume 2725 of *Lecture Notes in Computer Science*, 2003, pp. 1–13. doi:10.1007/978-3-540-45069-6_1.

[12] F. Slivovsky, Interpolation-based semantic gate extraction and its applications to QBF preprocessing, in: S. K. Lahiri, C. Wang (Eds.), Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I, Lecture Notes in Computer Science, Springer, 2020, pp. 508–528. doi:10.1007/978-3-030-53288-8_24.

[13] P. Golia, F. Slivovsky, S. Roy, K. S. Meel, Engineering an efficient boolean functional synthesis engine, in: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021, IEEE, 2021, pp. 1–9. doi:10.1109/ICCAD51958.2021.9643583.

[14] F. Reichl, F. Slivovsky, S. Szeider, Certified DQBF solving by definition extraction, in: C. Li, F. Manyà (Eds.), Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings, Lecture Notes in Computer Science, Springer, 2021, pp. 499–517. doi:10.1007/978-3-030-80223-3_34.

[15] F. Reichl, F. Slivovsky, Pedant: A certifying DQBF solver, in: K. S. Meel, O. Strichman (Eds.), 25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022,

Haifa, Israel, August 2-5, 2022, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 20:1–20:10. doi:10.4230/LIPICS.SAT.2022.20.

[16] E. W. Beth, On Padoa's method in the theory of definition, Indagationes Mathematicae 15 (1953) 330–339.

[17] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, F. Pollitt, Cadical 2.0, in: A. Gurfinkel, V. Ganesh (Eds.), Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I, Lecture Notes in Computer Science, Springer, 2024, pp. 133–152. doi:10.1007/978-3-031-65627-9_7.

[18] R. K. Brayton, A. Mishchenko, ABC: an academic industrial-strength verification tool, in: T. Touili, B. Cook, P. B. Jackson (Eds.), Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, Lecture Notes in Computer Science, Springer, 2010, pp. 24–40. doi:10.1007/978-3-642-14295-6_5.

[19] S. Akshay, S. Chakraborty, S. Goel, S. Kulal, S. Shah, Boolean functional synthesis: hardness and practical algorithms, Formal Methods Syst. Des. 57 (2021) 53–86. doi:10.1007/S10703-020-00352-2.

[20] J. Lagniez, P. Marquis, Boosting definability bipartition computation using SAT witnesses, in: S. A. Gaggl, M. V. Martinez, M. Ortiz (Eds.), Logics in Artificial Intelligence - 18th European Conference, JELIA 2023, Dresden, Germany, September 20-22, 2023, Proceedings, Lecture Notes in Computer Science, Springer, 2023, pp. 697–711. doi:10.1007/978-3-031-43619-2_47.