Bit-precise Reasoning with Parametric Bit-vectors

Zvika Berger
Bar-Ilan University, Ramat Gan, Israel

Yoni Zohar
Bar-Ilan University, Ramat Gan, Israel

Aina Niemetz
Stanford University, Stanford, USA

Mathias Preiner
Stanford University, Stanford, USA

Andrew Reynolds
The University of Iowa, Iowa City, USA

Clark Barrett
Stanford University, Stanford, USA

Cesare Tinelli
The University of Iowa, Iowa City, USA

Abstract

The SMT-LIB theory of bit-vectors is restricted to bit-vectors of fixed width. However, several important applications can benefit from reasoning about bit-vectors of symbolic widths, i.e., parametric bit-vectors. Recent work has introduced an approach for solving formulas over parametric bit-vectors, via an eager translation to quantified integer arithmetic with uninterpreted functions. The approach was shown to be successful for several applications, including the bit-width independent verification of compiler optimizations, invertibility conditions, and rewrite rules. We extend and improve that approach in several aspects. Theoretically, we improve expressiveness by defining a new theory of parametric bit-vectors that supports more operators and allows reasoning about the bit-widths themselves. Algorithmically, we introduce a lazy algorithm that avoids the use of uninterpreted functions and quantified axioms for them. Empirically, we show a significant improvement by implementing and evaluating our approach, and comparing it experimentally to the previous one.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Software and its engineering → Formal methods

Keywords and phrases Satisfiability Modulo Theories, Bit-precise Reasoning, Parametric Bit-vectors

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.9

Supplementary Material *Artifact (Zenodo)*: `https://zenodo.org/records/15143242`

Funding This research was supported in part by the Stanford Center for Automated Reasoning, the Stanford Center for Blockchain Research, Defense Advanced Research Projects Agency (DARPA) contract FA875024-2-1001, Israel Science Foundation (ISF) grant number 619/21, National Science Foundation (NSF) grant number 2303489, and a gift from Amazon Web Services.

1 Introduction

Bit-precise reasoning as provided by Satisfiability Modulo Theories (SMT) for the theory of fixed-size bit-vectors [3] is a key requirement for a wide range of verification applications. In particular, it allows verification engines to reason about machine integers and hardware registers and supports a variety of operators (including arithmetic, bitwise, and word operators such as concatenation and extraction). It is also very useful for other theories, such as floating-point arithmetic and non-linear integer arithmetic, where reasoning over certain fragments of those theories can be reduced to reasoning in the theory of fixed-size bit-vectors [7, 17].

One of the biggest limitations of this theory is already evident in its name: it only allows reasoning about bit-vectors whose size (or, equivalently, width) is *fixed*. Thus, when declaring a bit-vector sort in the SMT-LIB language, its width must be specified as a numeral. This limitation poses a serious expressiveness issue: when a verification tool verifies a property using this theory of bit-vectors, the verification result is only valid for the explicitly specified bit-width(s). In the context of software

28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025).
Editors: Jeremias Berg and Jakob Nordström; Article No. 9; pp. 9:1–9:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

verification, this means that some programs are proven correct in the presence of, say, 32-bit integer variables, but if their type is changed to 64-bit integers, the verification process has to be repeated. Similar issues occur during the development of specialized solvers for this theory, where much effort is dedicated to the design of *rewrite rules* and the generation of auxiliary *lemmas* [23,28,31,34]. Before adding new rules or lemmas in the solver, it is valuable to be able to verify that they are correct, and their correctness is usually independent of bit-width. Being able to do so fast and automatically using SMT solvers (as opposed to following the lengthier process of developing interactive proofs [12,19]) is highly beneficial for the development process. However, state-of-the-art SMT solvers do not currently support reasoning over bit-vectors of parametric size. Thus, it is common to verify the correctness of such rewrite rules and lemmas only for fixed bit-widths, from 1 up to some reasonably large value. This does not yield a proof of correctness in general although it increases confidence that the rules and lemmas are also correct for arbitrary bit-widths [28,31].

Early work on bit-precise reasoning supported reasoning about parametric bit-vectors, but only for a small fragment of the theory (see, e.g., [5,15,16,35]). In all cases, however, the supported fragment is not expressive enough for modern verification efforts.

More recently, an approach for solving formulas over parametric bit-vectors via an *eager* translation to quantified non-linear integer arithmetic with uninterpreted functions was proposed by Niemetz et al. [29,30]. In that work, bitwise operators and the exponentiation function with base 2, which is required for translating arithmetic bit-vector operators to the integers, are axiomatized by means of uninterpreted functions and quantifiers. In particular, [30] introduced a theory of parametric bit-vectors where bit-widths are implicit, and so it is not possible to reason about then. The translation was implemented outside of an SMT solver and evaluated on three case studies: bit-width independent verification conditions for compiler optimizations [20], invertibility conditions [28], and rewrite rules [34]. In addition, since all three case studies only require reasoning about a *single* parametric bit-width, the implementation of the translation targets a fragment of the SMT-LIB theory of bit-vectors that does not involve multiple bit-widths. As a result, the possibility of verifying, e.g., compiler optimization reductions was limited, as it excluded optimizations involving bit-vector concatenations, extractions and extensions, all of which involve multiple bit-widths.

Later work [39] presented a translation of *fixed-size* bit-vectors to quantifier-free non-linear integer arithmetic with uninterpreted functions (coined *int-blasting*, generalizing earlier approaches, see, e.g., [6,8,38]) with the goal of improving the scalability of reasoning over large bit-vectors. Unlike [30], the approach of [39] is *lazy*, does not introduce any quantifiers and does not support reasoning over parametric bit-vectors. Further, it is integrated in cvc5 [1]. While not in general competitive with state-of-the-art bit-vector solvers, int-blasting showed promising performance improvements on problems involving large bit-widths arising from smart contract verification.

In this paper, we propose a procedure for reasoning about parametric bit-vectors based on the eager approach presented in [30]. Our technique extends and combines this eager approach with lazy techniques introduced for reasoning about fixed size bit-vectors via a reduction to the integers in [39]. In particular, we make the following contributions:

(i) We introduce a theory of parametric bit-vectors with symbolic bit-widths as part of its signature. Our theory definition simplifies yet strengthens the theory of parametric bit-vectors introduced in [30] while enabling the reasoning over bit-widths.

(ii) We present a *lazy* algorithm for determining satisfiability of parametric bit-vectors constraints that does not rely on quantifiers (as opposed to [30]) and generalizes the lazy handling of bitwise *and* from [39] to parametric bit-vectors. Additionally, instead of eagerly axiomatizing exponentiation with base 2 (as in [30]), we handle this operator lazily.

(iii) We provide an implementation of a solver for parametric bit-vectors. The implementation is generic: it supports reasoning over arbitrary parametric bit-vector formulas (which goes beyond the prototype implementation presented in [30]). In particular, it fully supports reasoning over bit-widths, constraints on bit-vectors with multiple widths, and operators combining or returning bit-vectors of different widths, such as extraction, concatenation and extension.

(iv) We evaluate our approach on a large and diverse set of benchmarks and show that our technique significantly improves over the eager approach introduced in [30].

For brevity and simplicity, and similarly to previous work [30,39], our formal presentation focuses on *quantifier-free* formulas. We stress however, that our implementation fully supports quantifiers.[1]

The remainder of this paper is organized as follows. In Section 2, we provide the necessary background on first-order logic. In Section 3, we present our definition for a theory of parametric bit-vectors. In Section 4, we describe our new solver for parametric bit-vectors. We provide a detailed evaluation of the solver in Section 5 and conclude with directions for future research in Section 6.

2 Preliminaries

We briefly review notions of many-sorted first-order theories with equality (see [13,37] for more details). Let S be a set of *sort symbols*, and for every sort $\sigma \in S$, let X_σ be an infinite set of *variables of sort* σ. We assume that sets X_σ are pairwise disjoint and define X as the union of sets X_σ. A *signature* Σ consists of a set $\Sigma^s \subseteq S$ of sort symbols and a set Σ^f of function symbols. Arities of function symbols are defined in the usual many-sorted way. Constants are treated as nullary functions. We assume that Σ includes a Boolean sort Bool and the Boolean constants $\top$ (true) and $\bot$ (false). Function symbols whose return sort is Bool are also called *predicate* symbols. We assume that for each sort $\sigma \in \Sigma^s$, a binary equality predicate symbol $\approx_\sigma$ is included in the signature. We will write just $\approx$ when σ is clear or not important.

We assume the usual definitions of well-sorted terms, literals, and formulas, and refer to them as Σ-terms, Σ-literals, and Σ-formulas, respectively. We include the ternary if-then-else operator ite_σ of arity $\mathsf{Bool} \times \sigma \times \sigma \to \sigma$ for each $\sigma \in \Sigma^s$, and omit σ when it is clear from the context.

A Σ-*interpretation* $\mathcal{I}$ maps: each $\sigma \in \Sigma^s$ to a distinct non-empty set of values $\sigma^\mathcal{I}$ (the *domain* of σ in $\mathcal{I}$); each $x \in X_\sigma$ to an element $x^\mathcal{I} \in \sigma^\mathcal{I}$; and each $f^{\sigma_1 \cdots \sigma_n \sigma} \in \Sigma^f$ to a total function $f^\mathcal{I}$: $\sigma_1^\mathcal{I} \times ... \times \sigma_n^\mathcal{I} \to \sigma^\mathcal{I}$ if $n > 0$, and to an element in $\sigma^\mathcal{I}$ if $n = 0$. We use the usual inductive definition of the satisfiability relation $\models$ between Σ-interpretations and Σ-formulas.

A *theory* T is a pair (Σ, I), where Σ is a signature and I is a non-empty class of Σ-interpretations that is closed under variable reassignment, i.e., if interpretation $\mathcal{I}'$ only differs from an $\mathcal{I} \in I$ in how it interprets variables, then also $\mathcal{I}' \in I$. Each interpretation $\mathcal{I} \in I$ interprets Bool as the two-element set $\{\top^\mathcal{I}, \bot^\mathcal{I}\}$ and $\approx_\sigma$ as the identity relation over each sort $\sigma \in \Sigma^s$. A Σ-formula φ is T-*satisfiable* (resp. T-*unsatisfiable*) if it is satisfied by some (resp. no) interpretation in I; it is T-*valid* if it is satisfied by all interpretations in I.

The theory $T_{IA} = (\Sigma_{IA}, I_{IA})$ of integer arithmetic is defined as in the SMT-LIB 2 standard [2,4]. Its signature Σ_{IA} is shown in Table 1 and includes a single sort Int, function and predicate symbols $\{+, -, \cdot, \mathrm{div}, \mathrm{mod}, <, \leq, >, \geq\}$, and a constant symbol for every integer value. Given a set $\mathcal{F}$ of function symbols disjoint from those of Σ_{IA}, the signature $\Sigma_{IA}(\mathcal{F})$ is obtained from Σ_{IA} by the addition of the symbols in $\mathcal{F}$. The theory $T_{IA}(\mathcal{F})$ consists of all $\Sigma_{IA}(\mathcal{F})$-interpretations whose reduct to Σ_{IA} is a T_{IA}-interpretation (i.e., the symbols in $\mathcal{F}$ are freely interpreted). We may remove set braces when listing the elements of $\mathcal{F}$, e.g., by writing $\Sigma_{IA}(f, g)$ instead of $\Sigma_{IA}(\{f, g\})$.

[1] In fact, one of the benchmark sets in our experimental evaluation contains quantified formulas.

Symbol	SMT-LIB Syntax	Arity
$\approx_{\mathsf{Int}}, \not\approx_{\mathsf{Int}}$	`=`, `distinct`	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$
$0, 1, 2, \ldots$	`0, 1, 2, ...`	Int
$+, -, \cdot, \mathrm{div}, \mathrm{mod}$	`+`, `-`, `*`, `div`, `mod`	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
$\leq, \geq, <, >$	`<=`, `>=`, `<`, `>`	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$

Table 1 Signature Σ_{IA} of the theory of integer arithmetic T_{IA}.

3 A Theory of Parametric Bit-vectors

The SMT-LIB theory of bit-vectors T_{BV} does not support parametric bit-vectors. Neimetz et al. [30] define a theory of parametric bit-vectors based on auxiliary functions (outside of the signature of the theory). These functions provide an implicit mapping from bit-vector variables to their symbolic bit-width and from symbolic parametric bit-vector constants to Σ_{IA}-terms. A notion of *admissibility* is introduced to exclude mappings that assign non-standard interpretations (e.g., interpretations with zero or negative bit-widths). Furthermore, a parametric bit-vector formula is viewed as a *class* of fixed-size bit-vector formulas, i.e., a class of instances with concrete values for the bit-widths. This allows for a notion of *well-sortedness* based on the well-sortedness of these instances. As a consequence of relying on auxiliary mappings for symbolic bit-widths and constants, however, it is not possible to reason about bit-widths within that theory.

In the following, we introduce a new formal definition of the theory of parametric bit-vectors T_{PBV}. As in [30], we have a single sort PBV for bit-vectors of parametric size. Compared to that work, however, our definition of T_{PBV} does not rely on meta-level functions to maintain well-sortedness constraints. Instead, we make symbolic bit-widths of parametric bit-vector terms explicit in the signature via the new operator $|\textvisiblespace|$ and introduce an explicit representation of parametric bit-vector constants via the conversion operator $\mathrm{to\text{-}pbv}$. This enables reasoning about both parametric bit-vectors and their bit-widths within the theory.

3.1 Theory Definition

We define the theory of parametric bit-vectors T_{PBV} as the pair (Σ_{PBV}, I_{PBV}), where the signature Σ_{PBV} is the extension of Σ_{IA} described in Table 2. In addition to the integer sort Int, its set of sort symbols includes a single, new sort PBV for bit-vectors of parametric size. The set of function and predicate symbols of Σ_{PBV} consists of parametric variants of a strict subset of the fixed-size bit-vector operators defined in SMT-LIB 2. This set of *parametric bit-vector operators*, however, is complete in the sense that it suffices to express parametric variants of the remaining bit-vector operators from SMT-LIB 2. Additionally, we introduce two new function symbols, $|\textvisiblespace|$ of arity $\mathsf{PBV} \to \mathsf{Int}$ and $\mathrm{to\text{-}pbv}$ of arity $\mathsf{Int} \times \mathsf{Int} \to \mathsf{PBV}$, in the signature.

In all the interpretations of I_{PBV}, the domain of Int is the set of integer numbers, and the domain of PBV is the set of all bit-vectors of all possible positive widths. The operators in Σ_{IA} are interpreted in the same way as in T_{IA}. The new symbol $|\textvisiblespace|$ is interpreted as the function that maps each bit-vector x to its bit-width expressed as an integer. The new symbol $\mathrm{to\text{-}pbv}$ is interpreted as the function that maps any two integers n and m to the bit-vector of size n that represents $m \bmod 2^n$ in binary notation if $n > 0$, and to an arbitrary bit-vector otherwise. It can model constants of parametric bit-widths, e.g., $\mathrm{to\text{-}pbv}(k, 0)$ represents the bit-vector that consists of k zeros when $k \geq 0$. Operator $\circ$ is interpreted as the function that maps any two bit-vectors (of any two possible widths) to the bit-vector consisting of their concatenation. Operator $\textvisiblespace[\textvisiblespace : \textvisiblespace]$ denotes the function that maps a bit-vector x, an integer i,

Symbol	SMT-LIB Syntax	Arity
$\Sigma_{PBV} = \Sigma_{IA}$ + the following operators		
$\approx_{\mathsf{PBV}}, \not\approx_{\mathsf{PBV}}$	`=, distinct`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{Bool}$
$<_{\mathrm{u}}, >_{\mathrm{u}}, <_{\mathrm{s}}, >_{\mathrm{s}}$	`bvult, bvugt, bvslt, bvsgt`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{Bool}$
$\leq_{\mathrm{u}}, \geq_{\mathrm{u}}, \leq_{\mathrm{s}}, \geq_{\mathrm{s}}$	`bvule, bvuge, bvsle, bvsge`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{Bool}$
$\sim, -^{\mathcal{B}}$	`bvnot, bvneg`	$\mathsf{PBV} \to \mathsf{PBV}$
$\&, \|, \oplus$	`bvand, bvor, bvxor`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$\ll, \gg, \gg_a$	`bvshl, bvlshr, bvashr`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$+^{\mathcal{B}}, -^{\mathcal{B}}$	`bvadd, bvsub`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$\cdot^{\mathcal{B}}, \mathrm{mod}^{\mathcal{B}}, \mathrm{div}^{\mathcal{B}}$	`bvmul, bvurem, bvudiv`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$\textvisiblespace[\textvisiblespace : \textvisiblespace]$	`pextract`	$\mathsf{PBV} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{PBV}$
$\circ$	`concat`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$\mathrm{ext}_{\mathrm{z}}$	`pzero_extend`	$\mathsf{Int} \times \mathsf{PBV} \to \mathsf{PBV}$
ext_{s}	`psign_extend`	$\mathsf{Int} \times \mathsf{PBV} \to \mathsf{PBV}$
$\|\textvisiblespace\|$	`bvsize`	$\mathsf{PBV} \to \mathsf{Int}$
to-pbv	`int_to_pbv`	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{PBV}$
$\Sigma_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}}) = \Sigma_{IA}$ + the following operators		
$\&^{\mathbb{N}}$	`piand`	$\mathsf{Int} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
pow_2	`pow2`	$\mathsf{Int} \to \mathsf{Int}$

Table 2 Signatures Σ_{PBV} and $\Sigma_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$, defined as extensions of signature Σ_{IA}.

and an integer j, in that order, to the bit-vector of width $i - j + 1$ that ranges from the j-th bit of x to the i-th bit of x (inclusive) if $0 \leq j \leq i < |x|$, and is interpreted arbitrarily otherwise. For a bit-vector x, we refer to the i-th bit of x as $x[i]$, which abbreviates $x[i : i]$. We interpret $x[0]$ as the least significant bit (LSB) and $x[|x| - 1]$ as the most significant bit (MSB). Operators $\mathrm{ext}_{\mathrm{z}}$ and ext_{s} take an integer and a bit-vector argument and are interpreted as in the SMT-LIB theory of fixed-size bit-vectors whenever the first argument (the number of added bits) is non-negative: the former extends the second argument with leading 0s, while the latter extends it with its most significant bit. If the number of added bits is negative, the interpretation is arbitrary. Operators $\sim$ and $-^{\mathcal{B}}$ correspond to one's and two's complement. All other operators of Σ_{PBV} are interpreted according to the following principle: if both operands have the *same* (positive) bit-width, then their interpretation is the same as in the theory of fixed size bit-vectors. Otherwise, the interpretation is arbitrary.

▶ Remark 1. Recall that the signature of the theory of fixed-size bit-vectors T_{BV} has a unique sort $\mathsf{BV}(i)$ for each bit-width $i \in \{1, 2, 3, \ldots\}$. A more natural choice for a theory of parametric bit-vectors might then be the generalization of T_{BV} to sorts of the form $\mathsf{BV}(t)$ where t is an arbitrary integer term. However, this introduces the complication that syntactically distinct sort terms with equivalent integer arguments, such as $\mathsf{BV}(1+3)$ and $\mathsf{BV}(4)$ or $\mathsf{BV}(x+x)$ and $\mathsf{BV}(2 \cdot x)$, would have to denote the same domain (bit-vectors of 4 bits) or domain family (bit-vectors of even bit-width). This exceeds the expressiveness of many-sorted logic where, in effect, distinct sorts denote disjoint domains. For this reason, we define a single sort PBV to represent bit-vectors of all bit-widths. As a consequence, in our signature Σ_{PBV}, terms can be well-sorted that do not match well-sorted terms in T_{BV}. For example, the term $x +^{\mathcal{B}} y$ is well-sorted in our case even when x and y denote two bit-vectors of different bit-widths, something that is not permitted in the language of T_{BV}. Thus, we establish restrictions on the interpretations of Σ_{PBV} (described in Section 3.2) to have semantics that coincide with the semantics of the theory of fixed-size bit-vectors. In the example above, the restriction will be

Algorithm 1 Function $\textsc{Adm}$ to recursively construct admissibility constraints. Symbol z denotes constants of sort Int and x constants of sort PBV. Symbol $\bullet$ ranges over symbols of Σ_{IA}, Boolean connectives, and $\text{ite}_{\mathsf{Int}}$. Symbol $\diamond$ ranges over symbols in Σ_{PBV} not explicitly handled otherwise.

```
function ADM(e)
    match e:
        $x$                                 $\rightarrow$   $\textsc{Bw}(e) > 0$
        $z$                                 $\rightarrow$   $\top$
        $\text{to-pbv}(k, t)$               $\rightarrow$   $\textsc{Bw}(e) > 0 \wedge \textsc{Adm}(t)$
        $|t|$                               $\rightarrow$   $\textsc{Adm}(t)$
        $t[i : j]$                          $\rightarrow$   $0 \leq j \leq i < \textsc{Bw}(t) \wedge \textsc{Adm}(t)$
        $\text{ext}_z(n, t)$                $\rightarrow$   $n \geq 0 \wedge \textsc{Adm}(t)$
        $\text{ext}_s(n, t)$                $\rightarrow$   $n \geq 0 \wedge \textsc{Adm}(t)$
        $t_1 \circ t_2$                     $\rightarrow$   $\textsc{Adm}(t_1) \wedge \textsc{Adm}(t_2)$
        $\text{ite}_{\mathsf{PBV}}(t_1, t_2, t_3)$   $\rightarrow$   $\textsc{Bw}(t_2) \approx \textsc{Bw}(t_3) \wedge \bigwedge_{i=1}^{3} \textsc{Adm}(t_i)$
        $\bullet(t_1, \ldots, t_n)$         $\rightarrow$   $\bigwedge_{i=1}^{n} \textsc{Adm}(t_i)$
        $\diamond(t_1, \ldots, t_n)$        $\rightarrow$   $(\bigwedge_{i=2}^{n} \textsc{Bw}(t_1) \approx \textsc{Bw}(t_i)) \wedge (\bigwedge_{i=1}^{n} \textsc{Adm}(t_i))$
```

that x and y can only be interpreted as bit-vectors of the same width.

Note that parametric bit-vectors are expected to be supported more faithfully in the upcoming Version 3 of SMT-LIB, which is based on a higher-order logic with dependent types.

3.2 Satisfiability and Admissible Satisfiability

The way it is defined, theory T_{PBV} contains *non-standard* interpretations as it has to account for terms like $(x \circ y) +^{\mathcal{B}} x$ and formulas like $x \circ x \approx x$ that are not well sorted in the theory of fixed-size bit-vectors. To address this laxness of T_{PBV}'s sort system, we therefore define a notion of *admissible* T_{PBV}-satisfiability that excludes such spurious interpretations. This definition relies on function $\textsc{Adm}$, defined in Algorithm 1. Function $\textsc{Adm}$ collects width constraints while traversing over a given Σ_{PBV}-term. It is defined via a function $\textsc{Bw}$ as given in Algorithm 2, which constructs an integer term representing the symbolic bit-width of a parametric bit-vector term.

▶ **Definition 2.** *Let φ be a Σ_{PBV}-formula and $\mathcal{I}$ a T_{PBV}-interpretation. Let $\textsc{Adm}$ be a function as defined in Algorithm 1. Interpretation $\mathcal{I}$ is* admissible w.r.t. φ *if $\mathcal{I} \models \textsc{Adm}(\varphi)$. Formula φ is* admissibly T_{PBV}-satisfiable *if it is satisfied by a T_{PBV}-interpretation that is admissible w.r.t. φ.*

Thus, a formula φ is *admissibly* T_{PBV}-satisfiable only if it is satisfied by an *admissible* T_{PBV}-interpretation $\mathcal{I}$. We require that an admissible interpretation $\mathcal{I}$ assign bit-widths to Σ_{PBV}-terms in φ while satisfying the *admissibility condition* described by the function $\textsc{Adm}$ defined in Algorithm 1.

▶ **Example 3.** Consider a formula φ given as $y \approx z \circ w$. The admissibility condition $\textsc{Adm}(\varphi)$ is determined as $|y| \approx |z| + |w| \wedge |y| > 0 \wedge |z| > 0 \wedge |w| > 0$. An admissible T_{PBV}-interpretation $\mathcal{I}$ satisfying φ is given by $y^{\mathcal{I}} = 00$, $z^{\mathcal{I}} = 0$ and $w^{\mathcal{I}} = 0$. Thus, φ is admissibly T_{PBV}-satisfiable. Now, consider formula φ' given as $x \approx x +^{\mathcal{B}} (x \circ x)$. Its admissibility condition $\textsc{Adm}(\varphi')$ is defined as $|x| \approx |x| \wedge |x| > 0 \wedge |x| \approx |x| + |x|$ (after simplifications), which is T_{PBV}-unsatisfiable. That is, there exists no admissible T_{PBV}-interpretation for φ' and, thus, φ' is not admissibly T_{PBV}-satisfiable. However, φ' is T_{PBV}-satisfiable: consider a T_{PBV}-interpretation $\mathcal{I}$, given by $x^{\mathcal{I}} = 0$. Then, $(x \circ x)^{\mathcal{I}}$ is the bit-vector 00. Since x and $x \circ x$ do not have the same bit-width, $\mathcal{I}$ may interpret $x +^{\mathcal{B}} (x \circ x)$ arbitrarily, and so we can set $(x +^{\mathcal{B}} (x \circ x))^{\mathcal{I}}$ to be again the bit-vector 0. We then get that $\mathcal{I}$ satisfies φ'.

Algorithm 2 Function BW to compute the bit-width of PBV-terms. Symbol x denotes variables of sort PBV. Symbol $\diamond$ ranges over the symbols of Σ_{PBV} of sort PBV not explicitly handled otherwise.

```
function BW($e$)
    match e:
        $x$                                  $\rightarrow$  $|x|$
        $\text{to-pbv}(k, t)$                $\rightarrow$  $k$
        $t[i : j]$                           $\rightarrow$  $i - j + 1$
        $\text{ext}_z(n, t)$                 $\rightarrow$  $\text{BW}(t) + n$
        $\text{ext}_s(n, t)$                 $\rightarrow$  $\text{BW}(t) + n$
        $t_1 \circ t_2$                      $\rightarrow$  $\text{BW}(t_1) + \text{BW}(t_2)$
        $\text{ite}_{\mathsf{PBV}}(t_1, t_2, t_3)$  $\rightarrow$  $\text{BW}(t_2)$
        $\diamond(t_1, \ldots, t_n)$         $\rightarrow$  $\text{BW}(t_1)$
```

Note that our notion of admissibility corresponds to the notion of admissibility introduced by Niemetz et al. [30]. However, its definition differs in one key aspect. In [30], admissibility was enforced by using auxiliary functions and by instantiating parametric bit-vector formulas for all possible fixed bit-widths via universal quantification over the parametric bit-widths. Bit-widths and mappings from PBV-constants to Σ_{IA}-terms were implicit in the signature, whereas in our definition of Σ_{PBV}, we explicitly include operators $|\_|$ and to-pbv. This allows us to explicitly define admissibility conditions and to reason about symbolic bit-widths in Σ_{PBV}-formulas.

4 A Solver for Parametric Bit-vectors

In this section, we describe a solver for admissible T_{PBV}-satisfiability. Its main component is the translation of Σ_{PBV}-formulas to formulas over a signature $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$. As defined in Table 2, signature $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$ extends Σ_{IA} with two new function symbols, pow_2 and $\&^{\mathbb{N}}$.

We define a theory $T_{IA}(\text{pow}_2, \&^{\mathbb{N}})$ over $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$, where symbols pow_2 and $\&^{\mathbb{N}}$ are interpreted *arbitrarily*. Additionally, for the purpose of the translation, we consider a theory $T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$ over $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$ in which the interpretation of pow_2 and $\&^{\mathbb{N}}$ is *fixed* and defined as follows. The term $\text{pow}_2(n)$ is interpreted as the n-th power of 2 whenever n is non-negative, and as 0 otherwise. The interpretation of $\&^{\mathbb{N}}(k, a, b)$ is defined as follows. If $k > 0$ and $a, b \in \{0, \ldots, 2^k - 1\}$, then $\&^{\mathbb{N}}(k, a, b)$ is the integer that corresponds to performing a bitwise conjunction on the unsigned bit-vector representations of a and b. That is, let a', b' be the unsigned bit-vector representations of a and b of width k, then the interpretation of $\&^{\mathbb{N}}(k, a, b)$ is defined as $\Sigma_{i=0}^{k-1} a'[i] \cdot b'[i] \cdot 2^i$. If $k > 0$ and $a, b \notin \{0, \ldots, 2^k - 1\}$, then the interpretation of $\&^{\mathbb{N}}(k, a, b)$ is the same as $\&^{\mathbb{N}}(k, a \bmod 2^k, b \bmod 2^k)$. If $k \leq 0$, then its interpretation is 0.

▶ Remark 4. Notice that pow_2 is defined over the integers. Thus, we always interpret a negative exponent $-n$ as 0, as this is consistent with truncating the integer division in $2^{-n} \equiv \frac{1}{2^n}$ towards zero. Further, notice that $\&^{\mathbb{N}}$ defines a special case for bit-width $k \leq 0$. Our translation will never produce $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formulas that are satisfiable due to these corner cases because it ensures that: (1) the exponents of pow_2 are integer terms representing bit-widths and (2) integer terms that represent bit-widths are always positive. In fact, for our purposes, the special cases mentioned above could also be interpreted arbitrarily. We choose a fixed interpretation since it is easier to implement.

Starting with a Σ_{PBV}-formula, we first simplify the formula by applying rewrite rules $\mathcal{RW}_{\mathcal{B}}$, as described in Section 4.1. The rewritten Σ_{PBV}-formula is then translated to a $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formula using function TRANS, as described in Section 4.2. Finally, the $T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-satisfiability of the resulting $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formula is determined via procedure SOLVE$_\star$, described in Section 4.3.

4.1 The Parametric Bit-Vector Rewriter $\mathcal{RW}_{\mathcal{B}}$

State-of-the-art SMT solvers rely heavily on simplification techniques that are applies as a preprocessing step, before the main solver process begins. One such technique is based on term rewriting. For the theory of fixed-size bit-vectors (for which the dominant solving technique is bit-blasting, an eager reduction to propositional logic), SMT solvers implement hundreds of rewrite rules. However, rewrite rules for T_{BV} that target bit-blasting are not necessarily beneficial for alternative solving procedures. In particular, they may not be useful for our T_{PBV} procedure, which relies on a translation to integer arithmetic. Thus, some care is required in selecting rewrite rules for our procedure.

We implemented a set of rewrite rules for T_{PBV} based on rewrite rules for T_{BV} as implemented in the SMT solver cvc5 [1]. Rewrite rules implemented in cvc5 are documented in the domain-specific language RARE [33], which allows for easy adoption and lifting to T_{PBV}. We selected T_{BV}-rewrite rules for lifting to T_{PBV} based on the following principles: (i) a rewrite rule should not introduce a bitwise operator; (ii) the translation of the rewritten formula to the integers should not have more occurrences of operators mod and pow_2 than the original one. These principles are based on the fact that state-of-the-art solvers for integer arithmetic do not support bitwise operations over integers, and also offer very limited support for exponentiation. Further, mod is non-linear and one of the most expensive operations for integer arithmetic procedures. Interestingly, following these guiding principles, some T_{BV}-rewrite rules are useful for T_{PBV} when applied in the *opposite* direction.

The rewrite rules that we have implemented are presented in Section A.

▶ **Example 5.** Consider a T_{BV}-rule that rewrites $(x \mathbin{\&} y)[i:j]$ to $(x[i:j]) \mathbin{\&} (y[i:j])$. This is useful for bit-blasting since it reduces the size of the bit-level representation. However, for our translation to integer arithmetic, extractions are expensive as they introduce div, mod and pow_2 terms. Thus, we did not lift this rule to T_{PBV} but included its right-to-left variant in $\mathcal{RW}_{\mathcal{B}}$ instead.

4.2 The Translation Function TRANS

After obtaining the rewritten Σ_{PBV}-formula φ, we translate it into a $\Sigma_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$-formula φ' such that φ is admissibly T_{PBV}-satisfiable iff φ' is $T_{IA}(\mathrm{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-satisfiable. The translation function TRANS is given in Algorithms 3 and 4. It is based on the one in [30], with changes underlined.

Prior to the actual conversion step, Σ_{PBV}-formula φ is augmented, conjunctively, with the admissibility constraints $\mathrm{ADM}(\varphi)$, constructed as described in Algorithm 1, as well as the *range and size constraints* $\mathrm{RANGE}(\varphi)$. The latter are defined over integer constants that represent either parametric bit-vector constants or symbolic bit-widths, as described in Algorithm 4. The latter traverses the formula and adds constraints for the range of the integer terms representing bit-vector terms, according to their original bit-widths.

The augmented Σ_{PBV}-formula is then converted via function CONV into an equisatisfiable $\Sigma_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$-formula as defined in Algorithm 3.

We assume a one-to-one mapping χ from variables of sort PBV to variables of sort Int, as well as a one-to-one mapping κ from PBV-terms to variables of sort Int, such that their images are disjoint. Intuitively, χ translates variables of sort PBV to corresponding variables of sort Int while κ translates terms of the form $|t|$ to fresh integer variables representing the size of the parametric bit-vector term t. Compared to our previous work [30], our new translation handles the new operators $|\_|$ and to-pbv, improves the encodings of the operators $\gg$, $|$, and $\oplus$ to eliminate applications of mod, and adds support for the operators $\_[\_:\_]$, ext_s, and ext_z. Notice that the encoding of $\gg_a$ is not explicitly described, as that operator can be expressed in terms of $\gg$ and ite.

Following Zohar et al. [39], we do not handle $|$ and $\oplus$ natively. Instead, we eliminate them by means of $\&$, $+^{\mathcal{B}}$ and $-^{\mathcal{B}}$, based on the following identities derived from [18]: $x \mid y \approx$

Algorithm 3 Translation function $\textsc{Trans}$ and conversion function $\textsc{Conv}$. We use x for PBV variables, z for Int variables, $\mathsf{uts}(k, z)$ for $2 \cdot (z \bmod \mathrm{pow}_2(k-1)) - z$, $\bowtie$ for any operator in $\{<, \leq, >, \geq\}$, and $\bullet$ for ite, logical connectives and symbols in Σ_{IA}.

```
function $\textsc{Trans}(\varphi)$
    return $\textsc{Conv}(\varphi \wedge \textsc{Range}(\varphi) \wedge \underline{\textsc{Adm}(\varphi)})$

function $\textsc{Conv}(e)$
    match e:
        $\underline{z}$                    $\rightarrow$ $\underline{z}$
        $\underline{|t|}$                  $\rightarrow$ $\underline{\kappa(t)}$
        $\underline{\text{to-pbv}(k, t)}$  $\rightarrow$ $\underline{\textsc{Conv}(t) \bmod \mathrm{pow}_2(k)}$
        $x$                    $\rightarrow$ $\chi(x)$
        $t_1 \approx t_2$      $\rightarrow$ $\textsc{Conv}(t_1) \approx \textsc{Conv}(t_2)$
        $t_1 \bowtie_u t_2$    $\rightarrow$ $\textsc{Conv}(t_1) \bowtie \textsc{Conv}(t_2)$
        $t_1 \bowtie_s t_2$    $\rightarrow$ $\mathsf{uts}(\kappa(t_1), \textsc{Conv}(t_1)) \bowtie \mathsf{uts}(\kappa(t_2), \textsc{Conv}(t_2))$
        $t_1 +^{\mathcal{B}} t_2$    $\rightarrow$ $(\textsc{Conv}(t_1) + \textsc{Conv}(t_2)) \bmod \mathrm{pow}_2(\kappa(t_1))$
        $t_1 -^{\mathcal{B}} t_2$    $\rightarrow$ $(\textsc{Conv}(t_1) - \textsc{Conv}(t_2)) \bmod \mathrm{pow}_2(\kappa(t_1))$
        $t_1 \cdot^{\mathcal{B}} t_2$  $\rightarrow$ $(\textsc{Conv}(t_1) \cdot \textsc{Conv}(t_2)) \bmod \mathrm{pow}_2(\kappa(t_1))$
        $t_1 \operatorname{div}^{\mathcal{B}} t_2$  $\rightarrow$ $\mathrm{ite}(\textsc{Conv}(t_2) \approx 0, \mathrm{pow}_2(\kappa(t_1)) - 1, \textsc{Conv}(t_1) \operatorname{div} \textsc{Conv}(t_2))$
        $t_1 \operatorname{mod}^{\mathcal{B}} t_2$  $\rightarrow$ $\mathrm{ite}(\textsc{Conv}(t_2) \approx 0, \textsc{Conv}(t_1), \textsc{Conv}(t_1) \bmod \textsc{Conv}(t_2))$
        $\sim t$               $\rightarrow$ $\mathrm{pow}_2(\kappa(t)) - (\textsc{Conv}(t) + 1)$
        $-^{\mathcal{B}} t$    $\rightarrow$ $(\mathrm{pow}_2(\kappa(t)) - \textsc{Conv}(t)) \bmod \mathrm{pow}_2(\kappa(t))$
        $t_1 \ll t_2$          $\rightarrow$ $(\textsc{Conv}(t_1) \cdot \mathrm{pow}_2(\textsc{Conv}(t_2))) \bmod \mathrm{pow}_2(\kappa(t_1))$
        $t_1 \gg t_2$          $\rightarrow$ $\underline{\textsc{Conv}(t_1) \operatorname{div} \mathrm{pow}_2(\textsc{Conv}(t_2))}$
        $t_1 \ \&\ t_2$        $\rightarrow$ $\underline{\&^{\mathbb{N}}(\kappa(t_1), \textsc{Conv}(t_1), \textsc{Conv}(t_2))}$
        $t_1 \mid t_2$         $\rightarrow$ $\underline{\textsc{Conv}(t_1) + \textsc{Conv}(t_2) - \&^{\mathbb{N}}(\kappa(t_1), \textsc{Conv}(t_1), \textsc{Conv}(t_2))}$
        $t_1 \oplus t_2$       $\rightarrow$ $\underline{\textsc{Conv}(t_1) + \textsc{Conv}(t_2) - 2 \cdot \&^{\mathbb{N}}(\kappa(t_1), \textsc{Conv}(t_1), \textsc{Conv}(t_2))}$
        $t_1 \circ t_2$        $\rightarrow$ $\underline{\textsc{Conv}(t_1) \cdot \mathrm{pow}_2(\kappa(t_2)) + \textsc{Conv}(t_2)}$
        $\underline{t[i:j]}$               $\rightarrow$ $\underline{(\textsc{Conv}(t) \operatorname{div} \mathrm{pow}_2(j)) \bmod \mathrm{pow}_2(i - j + 1)}$
        $\underline{\mathrm{ext}_z(n, t)}$ $\rightarrow$ $\underline{\textsc{Conv}(t)}$
        $\underline{\mathrm{ext}_s(n, t)}$ $\rightarrow$ $\underline{\mathrm{ite}(\textsc{Conv}(t[\kappa(t) - 1]) \approx 1,}$
                                   $\underline{(\mathrm{pow}_2(n) - 1) \cdot \mathrm{pow}_2(\kappa(t)) + \textsc{Conv}(t), \textsc{Conv}(t))}$
        $\underline{\bullet(t_1, \ldots, t_n)}$  $\rightarrow$ $\underline{\bullet(\textsc{Conv}(t_1), \ldots, \textsc{Conv}(t_n))}$
```

$(x +^{\mathcal{B}} y) -^{\mathcal{B}} (x \ \&\ y)$ and $x \oplus y \approx (x \mid y) -^{\mathcal{B}} (x \ \&\ y)$. This elimination is embedded in the translation function and improves upon the original elimination in [39] by avoiding the introduction of additional mod operations.

▶ **Example 6.** Consider again the formula $\varphi := y \approx z \circ w$ from Example 3. Then, $\textsc{Trans}(\varphi) = \textsc{Conv}(\varphi \wedge \textsc{Range}(\varphi) \wedge \textsc{Adm}(\varphi))$. Now, $\textsc{Conv}(\varphi)$ is $y' \approx z' \cdot \mathrm{pow}_2(k_w) + w'$, where $y' = \chi(y)$, $z' = \chi(z)$, $w' = \chi(w)$, and $k_w = \kappa(w)$. Further, $\textsc{Conv}(\textsc{Range}(\varphi))$ is $0 \leq y' < \mathrm{pow}_2(k_y) \wedge 0 \leq z' < \mathrm{pow}_2(k_z) \wedge 0 \leq w' < \mathrm{pow}_2(k_w)$, where $k_y = \kappa(y)$ and $k_z = \kappa(z)$. The result of $\textsc{Adm}(\varphi)$ is given in Example 3, and then $\textsc{Conv}$ replaces $|y|$, $|z|$, and $|w|$ by k_y, k_z, and k_w, respectively.

Our translation function $\textsc{Trans}$ makes heavy use of $\bmod$ and pow_2 operations. In practice, some of these applications can be safely eliminated to optimize the encoding. For example, consider the term $(x +^{\mathcal{B}} x) +^{\mathcal{B}} x$, where $\kappa(x) = k$. The result of $\textsc{Conv}$ on this term is $(((x' + x') \bmod 2^k) + x') \bmod 2^k$, where $x' = \chi(x)$. Instead, we directly translate it to $((x' + x') + x') \bmod 2^k$.

Algorithm 4 Function RANGE to recursively construct size and range constraints. We use x for PBV variables, z for Int variables, and $\bullet$ for ite, logical connectives and symbols in Σ_{PBV}.

```
function RANGE($e$)
    match e:
        $x$                          $\rightarrow$  $0 \leq \chi(x) < \mathrm{pow}_2(\kappa(x))$
        $\underline{|t|}$            $\rightarrow$  $\underline{e \approx \textsc{Bw}(t) \wedge \textsc{Range}(t)}$
        $z$                          $\rightarrow$  $\top$
        $\bullet(t_1, \ldots, t_n)$  $\rightarrow$  $\bigwedge_{i=1}^{n} \textsc{Range}(t_i)$
```

Algorithm 5 Procedure for $T_{IA}(\mathrm{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$-satisfiability. We assume SOLVE is a procedure for $T_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$-satisfiability that returns "sat" or "unsat" and an interpretation $\mathcal{I}$ for satisfiable formulas. For a set X, *Terms*(X) denotes the set of terms occurring in X.

```
function SOLVE$_\star$($\varphi$)
    $\Gamma \leftarrow \{\varphi\}$
    loop
        $result, \mathcal{I} \leftarrow \textsc{Solve}(\Gamma)$
        if $result$ is "unsat" return "unsat"
        $\mathcal{L} \leftarrow \mathcal{L}_{\&^{\mathbb{N}}}(\{\&^{\mathbb{N}}(k, t_1, t_2) \mid \&^{\mathbb{N}}(k, t_1, t_2) \in \mathit{Terms}(\Gamma)\}, \mathcal{I})$
        $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}_{\mathrm{pow}_2}(\{\mathrm{pow}_2(t) \mid \mathrm{pow}_2(t) \in \mathit{Terms}(\mathcal{L} \cup \Gamma)\}, \mathcal{I})$
        if $\mathcal{I} \models \mathcal{L}$ return "sat"
        $\Gamma \leftarrow \Gamma \cup \{\psi \mid \psi \in \mathcal{L}$ and $\mathcal{I} \not\models \psi\}$
```

4.3 The $T_{IA}(\mathrm{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$-Solver

After translating a Σ_{PBV}-formula to a $\Sigma_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$-formula, we check if it is $T_{IA}(\mathrm{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$-satisfiable with a dedicated CEGAR-style procedure SOLVE$_\star$, which iteratively refines over-approximations of pow_2 and $\&^{\mathbb{N}}$ as given in Algorithm 5. This procedure takes an approach similar to the incremental linearization of non-linear arithmetic constraints [9, 10]. It relies on a set of quantifier-free lemmas $\mathcal{L}$ which fully specify the semantics of operators pow_2 and $\&^{\mathbb{N}}$. The set is constructed by instantiating the (implicitly) universally quantified lemma schemas $\mathcal{L}_{\mathrm{pow}_2}$, shown in Table 3, and $\mathcal{L}_{\&^{\mathbb{N}}}$, shown in Table 4, for all pow_2-terms and $\&^{\mathbb{N}}$-terms in φ. The instantiations of the lemmas also depend on the values of variables in the current interpretation $\mathcal{I}$. Note that when computing the instantiation for pow_2-lemmas, we also include pow_2-terms occurring in lemmas for $\&^{\mathbb{N}}$.

The procedure further assumes the availability of a subprocedure SOLVE to determine the $T_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$-satisfiability of a set of $\Sigma_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$-formulas Γ. Recall that in $T_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$, symbols pow_2 and $\&^{\mathbb{N}}$ are uninterpreted and thus, Γ is an over-approximation of φ in $T_{IA}(\mathrm{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$. If SOLVE determines the unsatisfiability of Γ, we conclude with "unsat". If it determines its satisfiability, we conclude with "sat" only if the resulting interpretation $\mathcal{I}$ also satisfies all lemmas in $\mathcal{L}$. Otherwise, we refine Γ by adding to it all lemmas that are not satisfied by $\mathcal{I}$.

The set $\mathcal{L}_{\mathrm{pow}_2}$ in Table 3 includes 4 lemma schemas from [30], which describe basic properties of the pow_2 operator. In addition, we consider the following three new lemmas. Lemma neg considers negative inputs, for which pow_2 is defined to be interpreted as 0 in $T_{IA}(\mathrm{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$. Lemma bound strengthens a lemma from recent work proposing a CEGAR-style approach for a new SMT theory for integer arithmetic with exponentiation [14]. From there, we get that when $x \geq 3$, then $2 \cdot x + 1$ is a lower bound for $\mathrm{pow}_2(x)$. We notice that when $x \geq 7$, we have $2 \cdot x^2$ as a tighter lower bound. In order to avoid non-linear multiplications, we linearize this bound via concrete values $v = x^{\mathcal{I}}$. We further include instances of value with $v = x^{\mathcal{I}}$, to block concrete model values.

Name	Lemma Schema	Source
	$\mathrm{pow}_2(x) \in S$	
positive	$x \geq 0 \Rightarrow \mathrm{pow}_2(x) > 0$	[30]
even	$x \geq 1 \Rightarrow \mathrm{pow}_2(x) \bmod 2 \approx 0$	[30]
div	$x \geq 0 \Rightarrow x \mathrm{~div~} \mathrm{pow}_2(x) \approx 0$	[30]
neg	$x < 0 \Rightarrow \mathrm{pow}_2(x) \approx 0$	new
bound	$(7 \leq v \wedge v \leq x) \Rightarrow v \cdot x + v^2 < \mathrm{pow}_2(x)$	new
value	$(0 \leq x \wedge x \approx v) \Rightarrow \mathrm{pow}_2(x) \approx 2^v$	new
	$\mathrm{pow}_2(x) \in S, \mathrm{pow}_2(y) \in S$	
monotonicity	$(0 \leq x \wedge x < y) \Rightarrow \mathrm{pow}_2(x) < \mathrm{pow}_2(y)$	[30]

Table 3 The set of lemma schemas defined by $\mathcal{L}_{\mathrm{pow}_2}(S, \mathcal{I})$, over a set of pow_2-terms S and an interpretation $\mathcal{I}$. The variables x and y in the formulas above are instantiated for each pow_2-term in S. Symbol v is a schema variable standing for an arbitrary numeral. It is instantiated with $v := x^{\mathcal{I}}$.

The set of lemma schemas $\mathcal{L}_{\&^{\mathbb{N}}}$ in Table 4 includes 8 lemmas that were introduced in [30] and capture basic properties of the $\&^{\mathbb{N}}$ operator. In addition, we consider 4 lemmas that did not appear in [30]. Lemma empty captures the corner case when the bit-width is non-positive. Lemmas lsb and one capture the cases where one of the arguments is even or 1. Note that due to lemma sym, it is sufficient to consider only one variant of each lemma. Lemma $\mathrm{sum}_{\geq}$ is based on the fact that for a fixed bit-width n, $\&^{\mathbb{N}}(n, x, y)$ is defined as $\Sigma_{i=0}^{n-1}\mathsf{ex}_i(x)\cdot\mathsf{ex}_i(y)\cdot 2^i$, with $\mathsf{ex}_i(x)$ defined as $(x \mathrm{~div~} 2^i) \bmod 2$ as in Table 4. We observe that for a bit-width $m \geq n$ with $0 \leq x < 2^n$ and $0 \leq y < 2^n$, the equation $\&^{\mathbb{N}}(m, x, y) \approx \&^{\mathbb{N}}(n, x, y)$ holds — consider, for example, $\&^{\mathbb{N}}(5, 1, 1) \approx \&^{\mathbb{N}}(4, 1, 1)$. Lemma $\mathrm{sum}_{\geq}$ is a generalization of this observation. It is instantiated with $v = k^{\mathcal{I}}$.

▶ **Example 7.** Consider the formula $x \,\&\, x \approx x$. Its translation will be $\&^{\mathbb{N}}(k, x, x) \approx x \bmod \mathrm{pow}_2(k)$, with the addition of the appropriate admissibility and range constraints. Then lemma div will be instantiated in set $\mathcal{L}$ as $k \geq 0 \Rightarrow k \mathrm{~div~} \mathrm{pow}_2(k) \approx 0$. Lemma min will be instantiated as $x \approx 0 \Rightarrow \&^{\mathbb{N}}(k, x, x) \approx 0$. In contrast, the approach of [30] introduces the universal closure of such lemmas, instead of specific instantiations.

Note that procedure $\textsc{Solve}_{\star}$ resembles the lazy approach of [39], with several key differences. First, since we are dealing with parametric bit-widths, our algorithm does not necessarily terminate, as the set of possible bit-widths is unbounded. Second, our algorithm not only lazily handles operator $\&^{\mathbb{N}}$, but also pow_2. Operator pow_2 did not occur when int-blasting in [39] since there, all exponents were concrete constants. Thus, exponentiation was simply evaluated. Finally, our sets of lemmas for $\&^{\mathbb{N}}$ and pow_2 expand both the axioms from [30] and the lemmas from [39].

Putting all components described in this section together, we obtain an incomplete but sound procedure $\textsc{Solve}_{\star}(\textsc{Trans}(\mathcal{RW}_{\mathcal{B}}(\varphi)))$ for admissible T_{PBV}-satisfiability.

▶ **Theorem 8.** *Let φ be a Σ_{PBV}-formula and let $\varphi' = \textsc{Trans}(\mathcal{RW}_{\mathcal{B}}(\varphi))$. If $\textsc{Solve}_{\star}(\varphi')$ terminates, then φ is admissibly T_{PBV}-satisfiable if and only if $\textsc{Solve}_{\star}(\varphi')$ returns "sat".*

4.4 Proof of Theorem 8

The correctness argument for Theorem 8 is similar to those from [30, 39]: assuming a T_{PBV}-interpretation that satisfies a Σ_{PBV}-formula φ, while also being admissible w.r.t. φ, we can construct a $T_{IA}(\mathrm{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-interpretation that satisfies the translation of φ, as well as the relevant lemmas from Tables 3 and 4, and vice versa. Compared to [30], the auxiliary admissibility condition that was assumed is now replaced by the satisfaction of $\textsc{Adm}(\varphi)$. What is left to show is:

Name	Lemma Schema	Source
	$\&^{\mathbb{N}}(k,x,y) \in S$	
base	$(k > 0 \wedge x \approx 1 \wedge y \approx 1) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx 1$	[30]
max	$(k > 0 \wedge \langle x \rangle_k \wedge y \approx \mathrm{pow}_2(k) - 1) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx x$	[30]
min	$y \approx 0 \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx 0$	[30]
idem	$(k > 0 \wedge \langle x \rangle_k \wedge x \approx y) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx x$	[30]
contra	$(x + y) \bmod \mathrm{pow}_2(k) \approx \mathrm{pow}_2(k) - 1 \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx 0$	[30]
range	$0 \leq x \wedge 0 \leq y \Rightarrow 0 \leq \&^{\mathbb{N}}(k,x,y) \leq \min(x,y)$	[30]
empty	$k \leq 0 \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx 0$	new
lsb	$x \bmod 2 \approx 0 \Rightarrow \&^{\mathbb{N}}(k,x,y) \bmod 2 \approx 0$	new
one	$(k > 0 \wedge y \approx 1) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx x \bmod 2$	new
$\mathrm{sum}_{\geq}$	$(k \geq v \wedge v > 0 \wedge \langle x \rangle_v \wedge \langle y \rangle_v) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx \Sigma_{i=0}^{v-1} \mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i$	new
	$\&^{\mathbb{N}}(k,x,z) \in S$, $\&^{\mathbb{N}}(k,y,w) \in S$	
sym	$(x \approx w \wedge y \approx z) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx \&^{\mathbb{N}}(k,z,w)$	[30]
diff	$(k > 0 \wedge z \approx w \wedge x \not\approx y \wedge \langle x \rangle_k \wedge \langle y \rangle_k \wedge \langle z \rangle_k) \Rightarrow (\&^{\mathbb{N}}(k,x,z) \not\approx y \vee \&^{\mathbb{N}}(k,y,w) \not\approx x)$	[30]

Table 4 The set of lemma schemas defined by $\mathcal{L}_{\&^{\mathbb{N}}}(S,\mathcal{I})$ over a set of $\&^{\mathbb{N}}$-terms S. The variables k, x, z, and w in the formulas above are instantiated for each $\&^{\mathbb{N}}$-term in S. We use the expression $\langle t \rangle_k$ to denote $0 \leq t < \mathrm{pow}_2(k)$, the expression $\min(x,y)$ to abbreviate $\mathrm{ite}(x < y, x, y)$, and $\mathsf{ex}_i(x)$ for $(x \;\mathrm{div}\; 2^i) \bmod 2$. Symbol v is a schema variable standing for an arbitrary numeral. It is instantiated with $v := k^{\mathcal{I}}$.

1. The new translation of $\gg$ is correct.
2. The eliminations of $|$ and $\oplus$ are correct.
3. The treatment of extraction and extensions is correct.
4. The rules of the $\mathcal{RW}_{\mathcal{B}}$ rewriter are T_{PBV}-valid.
5. All lemmas of Tables 3 and 4 are $T_{IA}(\mathrm{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-valid.

We prove the first item in Section 4.4.1 and the second in Section 4.4.2. The third follows directly from the SMT-LIB standard. For the fourth, notice that we only use rewrite rules that are already used in cvc5 and are bit-width independent. For their correctness proofs, see, e.g., [19]. For the fifth, most lemmas are taken from [30]. Of the new lemmas that we add, correctness is trivial for lemmas neg, value, empty, lsb and one. We prove the correctness of the remaining lemmas in Section 4.4.3.

4.4.1 Correctness of Translation of $\gg$

It suffices to show that in every $T_{IA}(\mathrm{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-interpretation that satisfies the formulas $\textsc{Conv}(x) \geq 0$, $\mathrm{pow}_2(\textsc{Conv}(y)) > 0$, and $\mathrm{pow}_2(k) > \textsc{Conv}(x)$, the expressions $\textsc{Conv}(x) \;\mathrm{div}\; \mathrm{pow}_2(\textsc{Conv}(y))$ and $(\textsc{Conv}(x) \;\mathrm{div}\; \mathrm{pow}_2(\textsc{Conv}(y))) \bmod \mathrm{pow}_2(k)$ are interpreted the same. This holds, as every such interpretation also satisfies $\textsc{Conv}(x) \;\mathrm{div}\; \mathrm{pow}_2(\textsc{Conv}(y)) \leq \textsc{Conv}(x) < \mathrm{pow}_2(k)$.

4.4.2 Correctness of Elimination of $|$ and $\oplus$

We start with the following lemma:

▶ **Lemma 9.** *Suppose $k > 0$, $x \geq 0$ and $y < 2^k$. Then, $0 \leq x + y - \&^{\mathbb{N}}(k,x,y) < 2^k$ and $0 \leq x + y - 2 \cdot \&^{\mathbb{N}}(k,x,y) < 2^k$.*

Proof. In this proof we treat $\&^{\mathbb{N}}$ and pow_2 as arithmetic operators with a fixed interpretation as given to them by all $T_{IA}(\mathrm{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-interpretations. This saves us from superscripting interpretations to these symbols, which is not needed since their interpretations are fixed. We consider the required inequalities separately:

- $0 \leq x + y - \&^{\mathbb{N}}(k, x, y) < 2^k$:
  1. By the semantics of $\&^{\mathbb{N}}$, we have $\&^{\mathbb{N}}(k, x, y) \leq x$. Since $y \geq 0$, we get $\&^{\mathbb{N}}(k, x, y) \leq x + y$. Therefore, $x + y - \&^{\mathbb{N}}(k, x, y) \geq 0$.
  2. Next, we show $x + y - \&^{\mathbb{N}}(k, x, y) < 2^k$. Since $0 \leq x, y < 2^k$, we have: $x = \Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(x)$ and $y = \Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(y)$, where for every $z \in \mathbb{N}$, $\mathsf{ex}_i(z) = (z \text{ div } 2^i) \bmod 2$. By the semantics of $\&^{\mathbb{N}}$, we have $\&^{\mathbb{N}}(k, x, y) = \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 1, 0)$. Notice that we can also have $y = \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 1, 0) + \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0)$. Now, we clearly have that $x + y - \&^{\mathbb{N}}(k, x, y)$ equals $\Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(x) + \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 10) + \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0) - \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 1, 0)$, which is: $\Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(x) + \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0)$. This is equal to $\Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = 1, 1, 0) + \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0)$. Clearly, this must be at most $\Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = 1, 1, 0) + \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ite}(\mathsf{ex}_i(x) = 0, 1, 0)$, which equals $\Sigma_{i=0}^{k-1} 2^i$, which must be strictly smaller than 2^k.
- $0 \leq x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y) < 2^k$:
  1. By the semantics of $\&^{\mathbb{N}}$, we have $\&^{\mathbb{N}}(k, x, y) \leq x$ and $\&^{\mathbb{N}}(k, x, y) \leq y$. Therefore, $x + y \geq 2 \cdot \&^{\mathbb{N}}(k, x, y)$, and so $x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y) \geq 0$.
  2. For the other direction, since $\&^{\mathbb{N}}(k, x, y) \geq 0$ for all k, x, y, we have: $x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y) \leq x + y - \&^{\mathbb{N}}(k, x, y)$. By the inequality item, $x + y - \&^{\mathbb{N}}(k, x, y) < 2^k$. ◀

We now justify the elimination of operator $|$ and $\oplus$, relying on the following equations from [18]:

$$x +^{\mathcal{B}} y \approx (x \ \& \ y) +^{\mathcal{B}} (x \mid y) \qquad (1)$$

$$x \oplus y \approx (x \mid y) -^{\mathcal{B}} (x \ \& \ y) \qquad (2)$$

For $|$, we get get: $x \mid y \approx (x +^{\mathcal{B}} y) -^{\mathcal{B}} (x \ \& \ y)$. Translating the right-hand side of this equation to integer arithmetic with $\&^{\mathbb{N}}$, we get $((x + y) \bmod 2^k - (\&^{\mathbb{N}}(k, x, y))) \bmod 2^k$, where k is the bit-width of x and y. This can be simplified to $((x + y) - (\&^{\mathbb{N}}(k, x, y))) \bmod 2^k$. In order to show an equality to $((x + y) - (\&^{\mathbb{N}}(k, x, y)))$, which is our actual translation, it is left to show that $0 \leq ((x + y) - (\&^{\mathbb{N}}(k, x, y))) < 2^k$, as stated in the first item of Lemma 9.

The justification for the elimination of operator $\oplus$ is similar. Combining (1) and (2), we get that $x \oplus y \approx ((x +^{\mathcal{B}} y) -^{\mathcal{B}} (x \ \& \ y)) -^{\mathcal{B}} (x \ \& \ y)$ and thus $x \oplus y \approx x +^{\mathcal{B}} y -^{\mathcal{B}} (2 \cdot^{\mathcal{B}} x \ \& \ y)$. Translating the right-hand side of this equation to integer arithmetic with $\&^{\mathbb{N}}$, we get $((x + y \bmod 2^k) - (2 \cdot \&^{\mathbb{N}}(k, x, y) \bmod 2^k)) \bmod 2^k$, which can be simplified to $(x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y)) \bmod 2^k$. To prove that this is the same as $(x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y))$, it suffices to prove that $0 \leq (x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y)) < 2^k$, as stated in the second item of Lemma 9.

4.4.3 Correctness of Lemmas

The following lemmas prove the validity of our lemmas bound and $\text{sum}_{\geq}$.

▶ **Lemma 10.** *Suppose $v \geq 7$. Then $x \geq v \Rightarrow v \cdot x + v^2 < \mathrm{pow}_2(x)$.*

Proof. We first prove that $x \geq 7 \Rightarrow 2 \cdot x^2 < 2^x$ by induction on x. If $x = 7$ then this trivially holds. Assume that this holds for some x. Then, $2 \cdot (x+1)^2 = 2 \cdot x^2 + 4 \cdot x + 2$. By the induction hypothesis, $2 \cdot x^2 < 2^x$. Also, $4 \cdot x + 2 < 2 \cdot x^2 < 2^x$ when $x \geq 7$. Hence we get $2 \cdot (x+1)^2 < 2^{x+1}$. Let $v \geq 7$ and suppose $x \geq v$. Then, $v \cdot x \leq x^2$ and $v^2 \leq x^2$, and so $v \cdot x + v^2 \leq 2 \cdot x^2 < 2^x$. ◀

▶ **Lemma 11.** *If $k \geq v > 0$ then $(\langle x \rangle_v \wedge \langle y \rangle_v) \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx \Sigma_{i=0}^{v-1} \mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i$.*

Proof. We always have $\&^{\mathbb{N}}(k, x, y) \approx \Sigma_{i=0}^{k-1} \mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i$. Since $k \geq v$ and $0 \leq x, y < 2^v$, we have that $\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 0$ for every $v \leq i \leq k - 1$, and thus $\mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i = 0$. ◀

5 Evaluation

We have implemented our approach by extending two tools: the generic SMT solver API smt-switch [21] and the SMT solver cvc5 [1]. In smt-switch, we extended the SMT-LIB parser with support for parsing Σ_{PBV}-formulas, and implemented our rewriter $\mathcal{RW}_{\mathcal{B}}$ (Section 4.1) and the translation $\textsc{Trans}$ (Algorithm 3). In cvc5, we implemented our procedure $\textsc{Solve}_{\star}$ (Algorithm 5) by extending cvc5's non-linear arithmetic module with two subsolvers to lazily handle pow_2 and $\&^{\mathbb{N}}$. The subsolver for $\&^{\mathbb{N}}$ generalizes the $\&^{\mathbb{N}}$-subsolver in [39], which was only able to reason over $\&^{\mathbb{N}}$ with fixed bit-widths, by allowing reasoning over symbolic bit-widths.

Note that the handling of Σ_{PBV}-formulas and their translation is entirely implemented in smt-switch. Externalizing the Σ_{PBV}-translation pipeline enables the use of our approach in combination with other back-end solvers that support reasoning over pow_2 and $\&^{\mathbb{N}}$ operators.

5.1 Benchmarks

The SMT-LIB standard [2, 4] does not yet define a theory for parametric bit-vectors. Consequently, benchmarks encoding problems over parametric bit-vectors are not yet part of the SMT-LIB benchmark library. Thus, we evaluate our techniques utilizing a wide range of benchmark sets, originating from various sources, as detailed below.

alive (200 benchmarks). Set *alive* consists of verification conditions for compiler optimizations that are generated by Alive [20]. The evaluation of [30] included 180 such conditions. Here, we include 20 additional conditions that were not supported by the implementation of [30]. Among the new conditions, 17 involve multiple bit-widths and 3 include terms of the form $\text{to-pbv}(k, k)$, representing a bit-vector of length k whose unsigned integer value is k.

ic (180 benchmarks). Set *ic* consists of benchmarks to verify the correctness of the *invertibility conditions* formalized in Niemetz et al. [27], which utilizes them for quantifier instantiation. They are also instrumental to the local search procedure of [27]. The evaluation of [30] included correctness checks for 160 out of 180 conditions. Here, we add the remaining 20 checks that encode invertibility conditions over concatenation, which were not supported by the implementation of [30].

rewrite (1500 benchmarks). Set *rewrite* consists of the bit-width independent correctness checks of rewrite rule candidates for the theory of fixed-size bit-vectors considered in [30]. They were automatically generated using CVC4SY [36], a Syntax-Guided Synthesis (SyGuS) engine.

syrew (1500 benchmarks). Set *syrew* consists of bit-width independent versions of the equivalence checks of T_{BV}-terms from [31], which were enumerated by the SyGuS engine of cvc5. There, these checks were instantiated for large bit-widths to evaluate the performance of T_{BV}-solvers.

lemmas (70 benchmarks). Set *lemmas*, also originating from [31], consists of 70 refinement lemmas describing properties of arithmetic bit-vector operators for a CEGAR-style procedure for T_{BV} implemented in the SMT solver Bitwuzla [25]. These lemmas were verified to be correct up to bit-width 256 [31] but are required to be correct for arbitrarily large bit-widths. This benchmark set encodes bit-width independent correctness checks of these lemmas. Note that two benchmarks involve multiple bit-widths, and thus cannot be handled by the implementation of [30].

icfb (46 benchmarks). Set *icfb* originates from a local search procedure [24] that generalizes the technique from [27] and defines invertibility conditions (and conditions for a weaker notion of invertibility called consistency) over ternary bit-vectors. These conditions were verified to be correct up to bit-width 65 but are required to hold for any bit-width. This benchmark set consists of bit-width independent correctness checks of the conditions, of which 16 benchmarks involve multiple bit-widths. Note that some conditions cannot be encoded to T_{PBV} due to the occurrence of bit-width dependent functions that require counting, e.g., the counting of leading zeroes, and were

Feature	Baseline	Eager	Pbv
Target Theory	$T_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}}, \|^{\mathbb{N}}, \oplus^{\mathbb{N}})$	$T_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$	$T_{IA}(\mathrm{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$
Multiple Bit-widths	✗	✓	✓
Lazy pow_2	✗	✗	✓
Lazy $\&^{\mathbb{N}}$	✗	✗	✓
$\|$*-elimination*	✗	✓	✓
$\oplus$*-elimination*	✗	✓	✓
$\gg$ *without* mod	✗	✓	✓
New lemmas for $\&^{\mathbb{N}}$	✗	✓	✓
New lemmas for pow_2	✗	✓	✓
No redundant axioms	✗	✓	✓
$\mathcal{RW}_{\mathcal{B}}$	✗	✓	✓
mod*-reduction*	✗	✓	✓

Table 5 Configurations considered in our experimental setup.

thus excluded from this set. Further note that in the original publication [24], one of the invertibility conditions for operator $<_{\mathrm{u}}$ was given incorrectly (the implementation and verification of the rule used the correct definition) [26]. We include both versions for this condition.

mut (9442 benchmarks). The majority of the benchmarks in the sets above are unsatisfiable. For a more thorough evaluation on satisfiable benchmarks, we created a benchmark set *mut* by mutating the benchmarks from the other sets. This realistically mimics the introduction of bugs in verification conditions, e.g., by using a bitwise disjunction instead of conjunction. Let (f, g) be a pair of Σ_{PBV}-terms or function symbols with the same arity, and let φ be a Σ_{PBV}-formula in which f occurs. A mutation of φ is obtained from (f, g) by replacing the first occurrence of f by g. We generated mutations for pairs $(f, g) \in \{(\&, |), (+^{\mathcal{B}}, -^{\mathcal{B}}), (-^{\mathcal{B}}, \sim), (\ll, \gg), (\text{to-pbv}(k, 1), \text{to-pbv}(k, 0)), (\text{to-pbv}(k, k), \text{to-pbv}(k, 0))\}$ and their permutations (g, f), and filtered out duplicates.

5.2 Experimental Setup

For our experimental evaluation, we consider three configurations: configuration Baseline, which corresponds to the best configuration of [30]; configuration Pbv, which implements our techniques as described in Section 4; and configuration Eager, which corresponds to extending configuration Baseline with our rewriter $\mathcal{RW}_{\mathcal{B}}$ and optimizations as detailed in Table 5, while still handling operators pow_2 and $\&^{\mathbb{N}}$ eagerly. Recall that the implementation of [30] utilized translations that were tailored to each considered benchmark set, which does not allow a faithful comparison to our Pbv-configurations. Thus, we reimplemented the best configuration *comb* from [30] as configuration Baseline in our tool. This not only allows us to evaluate Baseline on a larger benchmark set, but also ensures a fair comparison since all configurations use the same underlying cvc5 version as well as the same infrastructure for parsing, manipulating and solving formulas.

Table 5 outlines the differences between the configurations. The features considered for each configuration are listed in the first column. *Target Theory* specifies the underlying theory (e.g., whether pow_2 and $\&^{\mathbb{N}}$ are uninterpreted or not). Accordingly, *Lazy* pow_2 and *Lazy* $\&^{\mathbb{N}}$ determine whether these operators are handled lazily or eagerly. For uninterpreted functions, we use quantified axiomatizations, and *No redundant axioms* determines whether these axiomatizations are always added to the input formula, regardless of whether the axiomatized operator occurs in the formula. *Multiple Bit-widths* corresponds to supporting multiple bit-widths in the same formula. $|$*-elimination* and $\oplus$*-elimination* denote the elimination of $|$ and $\oplus$ in the translation (as described in Section 4.2). The usage of an optimization to the translation of $\gg$, which eliminates a mod operation (as shown in Algorithm 3), is determined by $\gg$ *without* mod. *New lemmas for* $\&^{\mathbb{N}}$ and *New lemmas for* pow_2 correspond to the usage of the new lemmas from Tables 3 and 4. $\mathcal{RW}_{\mathcal{B}}$ determines whether rewriter $\mathcal{RW}_{\mathcal{B}}$ from Section 4.1 is enabled, and mod*-reduction* determines whether some optimizations that eliminate nested mod in arithmetic operators are applied, as mentioned in Section 4.2.

Configurations BASELINE and EAGER rely on an *eager* translation. They introduce uninterpreted functions that are axiomatized by means of quantified formulas that are added as preamble to each translation. In contrast, configuration PBV employs a translation to theory $T_{IA}(\mathrm{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$, which does not rely on quantifiers and handles operators pow_2 and $\&^{\mathbb{N}}$ *lazily*. Further, configuration BASELINE does not support multiple bit-widths in the same formula. Configuration EAGER, on the other hand, incorporates several optimizations over BASELINE, including the new lemmas, that are also incorporated in PBV. Notice that operators $|$ and $\oplus$ are handled natively in configuration BASELINE, while configurations EAGER and PBV use elimination rules for them. In Section B, we include an extended set of results with 14 more configurations.

▶ Remark 12. Even though our evaluation includes benchmark sets from the evaluation in [30], the numbers are not directly comparable due to the following reasons. Each benchmark in set *ic* describes an equivalence, which was split into two implications in [30]. Additionally, *conditional inverses* were introduced there, and were incorporated into the benchmarks with the main goal of proving invertibility conditions as correct. Benchmark set *rewrite* was solved in a semi-automated and iterative manner: whenever a rewrite candidate was proven correct, it was included as an axiom in the benchmarks to prove. In this evaluation, we refrain from such interventions, as our goal is to measure how efficient our approach is in a fully automated setting. Further, the translation of [30] utilized quantifier-specific optimizations and included instantiation patterns as well as concrete instantiations for some of the axioms. However, such patterns are often tailored to a specific solving procedure, and concrete instantiations are only sound for unsatisfiable benchmarks (which was the focus of [30]). In configuration BASELINE, we do not use any of these optimizations.

We ran our experiments on a cluster of 22 machines equipped with Intel Xeon Gold 6348 CPUs. For each solver and benchmark pair, we used a CPU time limit of 60 seconds and a memory limit of 8GB. Preliminary experiments showed that the difference in results with higher time limits were not significant. Also, no configuration exceeded the memory limit of 8GB during solving.

5.3 Results

Table 6 summarizes the results for each benchmark set described in Section 5.1, over the configurations introduced in Section 5.2, as well as the virtual best solver over these configurations in column VBS.

Overall, configuration PBV significantly outperforms BASELINE and EAGER in number of solved instances and runtime. Comparing PBV against EAGER indicates that using a lazily handling pow_2 and $\&^{\mathbb{N}}$ is superior to the quantifier-based eager approach. Comparing EAGER against BASELINE shows that the optimizations discussed in Section 4 are beneficial for the eager translation as well. On the 1863 instances commonly solved by all three configurations, EAGER is almost 4 times faster than BASELINE while PBV is more than 12 times faster than EAGER. On the 3060 instances commonly solved by EAGER and PBV, configuration PBV is more than 18 times faster. Notice that configurations BASELINE and EAGER do not solve any satisfiable benchmarks, due to the usage of quantified axiomatizations. In fact, the presence of these axiomatizations already prevents both BASELINE and EAGER from reporting that the formula $\top$ is satisfiable.

Our benchmark sets include in total 371 benchmarks that involve multiple bit-widths, which BASELINE does not support. Configuration PBV solves 178 instances, 20 of which are satisfiable. Configuration EAGER solves 110 instances, all of which are unsatisfiable.

Simplifying the T_{PBV}-formula via rewriting as described in Section 4.1 is overall beneficial for both the EAGER and the PBV configurations but has a bigger impact on EAGER. Without rewriting, EAGER solves 474 fewer instances, while PBV only loses 72 instances.

Proving the correctness of the invertibility conditions from [28] for arbitrary bit-widths (benchmark

Benchmarks	#	BASELINE	EAGER	PBV	VBS
alive	200	71	93	**107**	124
ic	180	43	58	**77**	81
rewrite	2006	658	1221	**1331**	1382
syrew	1500	558	720	**912**	951
lemmas	70	12	14	**23**	23
icfb	46	1	9	**12**	12
mut	9441	669	1084	**4863**	4915
total	13443	2012	3199	**7325**	7488
sat		0	0	**3641**	3641
unsat		2012	3199	**3684**	3847
unique		24	57	**4222**	4303
time [seconds]		709k	634k	**375k**	355k

Table 6 Overall number of solved benchmarks for each configuration. VBS corresponds to the virtual best solver over all three configurations. Time is the penalized runtime, counting unsolved instances as timeout.

set *ic*) has previously been attempted with two different approaches: *(i)* the eager T_{PBV} approach described in [30] and *(ii)* a bit-width independent formalization in Coq [12]. Out of the 81 invertibility conditions proven across our three configurations, 11 were not proven by *(i)* and *(ii)*. Of these 11, 8 involve multiple bit-widths which neither *(i)* nor *(ii)* can handle. The remaining 3 conditions involve operators that are not supported by *(ii)* (signed inequalities, multiplication and division).

For benchmarks in sets *lemmas* and *icfb*, we provide the first bit-width independent verification. The lemmas and conditions in these sets were only proven correct for a range of fixed bit-widths [24, 31]. Configuration PBV was able to confirm that the incorrect condition is indeed incorrect. However, it was unable to prove the corrected condition within the given time limit.

6 Conclusion and Further Research

We have presented a new theory of parametric bit-vectors T_{PBV} and proposed a dedicated, lazy procedure for solving T_{PBV}-formulas. We have shown experimentally that our techniques are a significant improvement over previous techniques in a wide range of benchmarks.

In future work, we plan to explore proof production for parametric bit-vectors, based on the algorithms presented in this paper. This will enable the integration of the proposed solver into proof assistants like Coq [11], Lean [22] or Isabelle/HOL [32], in order to increase automation in these tools for proofs involving parametric bit-vectors.

For our evaluation, we compiled sets of benchmarks for various challenging applications. These benchmarks are, however, relatively small in terms of formula size. It would be interesting to evaluate the scalability of our approach with respect to formula size. However, obtaining appropriate benchmarks for this purpose is currently challenging.[2] Generating and experimenting with such benchmarks is left for future work.

[2] For example, *parameterizing* bit-vector benchmarks from SMT-LIB (which are exclusively over fixed-size bit-vectors) while preserving their semantics is not straightforward.

A Rewrite Rules of the Parametric Bit-Vector Rewriter $\mathcal{RW}_{\mathcal{B}}$

The following table lists the rewrite rules implemented in our rewriter $\mathcal{RW}_{\mathcal{B}}$. We denote with k_x the bitwidth of x, use a lexicographic ordering $<_{lex}$ between terms, and normalize commutative operations ($+^{\mathcal{B}}$, $\&$, etc.) according to this ordering. For example, if $t_1 <_{lex} t_2$, then $t_1 +^{\mathcal{B}} t_2$ is rewritten to itself and $t_2 +^{\mathcal{B}} t_1$ is rewritten to $t_1 +^{\mathcal{B}} t_2$.

Rule Name	Term	Rewritten Term
bv-concat-extract-merge	(concat (extract k (+ j 1) s) (extract j i s))	(extract k i s)
bv-extract-extract	(extract l k (extract j i x))	(extract (+ i l) (+ i k) x)
bv-extract-whole	(extract k_x 0 x)	x
bv-add-zero	(bvadd x 0)	x
bv-reverse-extract-and	(bvand (extract j i x) (extract j i y))	(extract j i (bvand x y))
bv-xor-simplify-2	(bvxor x (bvnot x))	(bvnot 0)
bv-or-zero	(bvor x (int_to_pbv k_x 0))	x
bv-mul-one	(bvmul x (int_to_pbv k_x 1))	x
bv-mul-zero	(bvmul x (int_to_pbv k_x 0))	(int_to_pbv k_x 0)
bv-zero-extend-eliminate	(zero-extend 0 x)	x
bv-sign-extend-eliminate	(sign-extend 0 x)	x
bv-not-neq	(= x (bvnot x))	false
bv-neg-sub	(bvneg (bvsub x y))	(bvsub y x)
bv-neg-idemp	(bvneg (bvneg x))	x
bv-ugt-eliminate	(bvugt x y)	(bvult y x)
bv-uge-eliminate	(bvuge x y)	(bvule y x)
bv-sgt-eliminate	(bvsgt x y)	(bvslt y x)
bv-sge-eliminate	(bvsge x y)	(bvsle y x)
bv-shl-by-const-0	(bvshl x (int_to_pbv k_x 0))	x
bv-shl-by-const-2	(bvshl x (int_to_pbv k_x k_x))	(int_to_pbv k_x 0)
bv-lshr-by-const-0	(bvlshr x (int_to_pbv k_x 0))	x
bv-lshr-by-const-2	(bvlshr x (int_to_pbv k_x k_x))	(int_to_pbv k_x 0)
bv-ashr-by-const-0	(bvashr x (int_to_pbv k_x 0))	x
bv-bitwise-idemp-2	(bvor x x)	x
bv-or-one	(bvor x (bvnot x))	(bvnot (int_to_pbv k_x 0))
bv-xor-duplicate	(bvxor x x)	(int_to_pbv k_x 0)
bv-xor-zero	(bvxor x (int_to_pbv k_x 0))	x
bv-bitwise-not-or	(bvor x (bvnot x))	(bvnot (int_to_pbv k_x 0))
bv-xor-not	(bvxor (bvnot x) (bvnot y))	(bvxor x y)
bv-not-idemp	(bvnot (bvnot x))	x
bv-ult-zero-1	(bvult (int_to_pbv k_x 0) x)	(not (= x (int_to_pbv k_x 0)))
bv-ult-zero-2	(bvult x (int_to_pbv k_x 0))	false
bv-ult-self	(bvult x x)	false
bv-lt-self	(bvslt x x)	false
bv-ule-self	(bvule x x)	true
bv-ule-zero	(bvule x (int_to_pbv k_x 0))	(= x (int_to_pbv k_x 0))
bv-zero-ule	(bvule (int_to_pbv k_x 0) x)	true
bv-sle-self	(bvsle x x)	true
bv-ule-max	(bvule x (bvnot x))	true
bv-udiv-zero	(bvudiv x (int_to_pbv k_x 0))	(bvnot (int_to_pbv k_x 0))
bv-udiv-one	(bvudiv x (int_to_pbv k_x 1))	x
bv-urem-one	(bvurem x (int_to_pbv k_x 1))	(int_to_pbv k_x 0)
bv-urem-self	(bvurem x x)	(int_to_pbv k_x 0)
bv-shl-zero	(bvshl (int_to_pbv k_x 0) x)	(int_to_pbv k_x 0)
bv-lshr-zero	(bvlshr (int_to_pbv k_x 0) x)	(int_to_pbv k_x 0)
bv-ashr-zero	(bvashr (int_to_pbv k_x 0) x)	(int_to_pbv k_x 0)
bv-ult-one	(bvult x (int_to_pbv k_x 1))	(= x (int_to_pbv k_x 0))

Feature	OR-E	XOR-E	SH-M-E	ALL-E	POW2++	PIAND++
Target Theory	T_1					
Multiple Bit-widths	✗	✗	✗	✗	✗	✗
Lazy pow_2	✗	✗	✗	✗	✗	✗
Lazy $\&^{\mathbb{N}}$	✗	✗	✗	✗	✗	✗
$\mid$*-elimination*	✓	✗	✗	✓	✗	✗
$\oplus$*-elimination*	✗	✓	✗	✓	✗	✗
$\gg$ *without* mod	✗	✗	✓	✓	✗	✗
New lemmas for $\&^{\mathbb{N}}$	✗	✗	✗	✗	✗	✓
New lemmas for pow_2	✗	✗	✗	✗	✓	✗
No redundant axioms	✗	✗	✗	✗	✗	✗
$\mathcal{RW}_{\mathcal{B}}$	✗	✗	✗	✗	✗	✗
mod*-reduction*	✗	✗	✗	✗	✗	✗

Table 7 Intermediate configurations with $T_1 = T_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}}, |^{\mathbb{N}}, \oplus^{\mathbb{N}})$.

Feature	POW2-L	PIAND-L	EAGER$^{\not{R}\text{-}}$	EAGER$^{\not{R}}$	EAGER$^{\text{-}}$	PBV$^{\not{R}}$	PBV$^{\not{R}\text{-}}$	PBV$^{\text{-}}$
Target Theory	T_3	T_4	T_5	T_5	T_5	T_2	T_2	T_2
Multiple Bit-widths	✗	✗	✓	✓	✓	✓	✓	✓
Lazy pow_2	✓	✗	✗	✗	✗	✓	✓	✓
Lazy $\&^{\mathbb{N}}$	✗	✓	✗	✗	✗	✓	✓	✓
$\mid$*-elimination*	✗	✗	✓	✓	✓	✓	✓	✓
$\oplus$*-elimination*	✗	✗	✓	✓	✓	✓	✓	✓
$\gg$ *without* mod	✗	✗	✓	✓	✓	✓	✓	✓
New lemmas for $\&^{\mathbb{N}}$	✗	✓	✓	✓	✓	✓	✓	✓
New lemmas for pow_2	✓	✗	✓	✓	✓	✓	✓	✓
No redundant axioms	✗	✗	✓	✓	✓	✓	✓	✓
$\mathcal{RW}_{\mathcal{B}}$	✗	✗	✗	✗	✓	✗	✗	✓
mod*-reduction*	✗	✗	✗	✓	✗	✓	✗	✗

Table 8 Intermediate configurations. $T_3 = T_{IA}(\mathrm{pow}_{2\star}, \&^{\mathbb{N}})$, that is, pow_2 has a fixed interpretation, while $\&^{\mathbb{N}}$ is freely interpreted. $T_4 = T_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}}_{\star})$, that is, $\&^{\mathbb{N}}$ has a fixed interpretation, while pow_2 is freely interpreted. $T_5 = T_{IA}(\mathrm{pow}_2, \&^{\mathbb{N}})$. $T_2 = T_{IA}(\mathrm{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$.

B More Detailed Evaluation

In addition to the experimental evaluation presented above, we have performed an in-depth ablation study of the various features listed in Table 5. In this section, we describe the results for 14 configurations we evaluated in addition to the three described in Table 5. These additional configurations are defined as described in Tables 7 and 8.

Configuration OR-E is obtained from BASELINE by turning on the elimination of $|$, and similarly for XOR-E. The right-most 6 configurations of Table 8 differ by turning on or off the bit-vector rewriter $\mathcal{RW}_{\mathcal{B}}$, as well as the arithmetic optimizations that eliminate mod operators. In particular, EAGER$^{\not{R}\text{-}}$ is obtained from EAGER by turning both of these off.

The results of the additional configurations are shown in Tables 9 and 10. Table 9 focuses on configurations that are based on BASELINE. Note that the elimination of mod in $\gg$ (configuration SH-M-E) has a greater effect than the elimination of $|$ and $\oplus$, since our benchmarks include more occurrences of $\gg$. Table 10 includes configurations that evaluate the effect of the bit-vector rewriter $\mathcal{RW}_{\mathcal{B}}$. Overall, enabling the rewriter improves performance, especially for the eager configurations.

Benchmarks	#	OR-E	XOR-E	SH-M-E	ALL-E	POW2++	PIAND++	VBS
alive	200	71	71	71	71	71	71	72
ic	180	44	44	48	49	43	43	50
rewrite	2006	692	653	724	758	669	633	809
syrew	1500	561	556	604	603	570	551	625
lemmas	70	12	12	13	13	12	12	13
icfb	46	1	1	1	1	1	1	1
mut	9441	692	663	774	794	669	634	843
total	13443	2071	2000	2235	2289	2033	1945	2413
sat		0	0	0	0	0	0	0
unsat		2071	2000	2235	2289	2033	1955	2413

Table 9 Overall number of solved benchmarks for each configuration. VBS corresponds to the virtual best solver over all three configurations.

Benchmarks	#	POW2-L	PIAND-L	$\text{EAGER}^{\cancel{\mathcal{R}}-}$	$\text{EAGER}^{\cancel{\mathcal{R}}}$	EAGER^{-}	$\text{PBV}^{\cancel{\mathcal{R}}}$	$\text{PBV}^{\cancel{\mathcal{R}}-}$	PBV^{-}	VBS
alive	200	63	74	87	95	85	105	97	100	125
ic	180	55	45	58	59	59	75	75	77	83
rewrite	2006	681	797	732	925	1068	1296	1148	1189	1381
syrew	1500	580	593	608	721	606	911	796	798	956
lemmas	70	17	13	14	14	14	23	23	23	25
icfb	46	1	1	9	9	9	11	11	12	12
mut	9441	732	733	885	902	1088	4832	4840	4881	5138
total	13443	2129	2256	2393	2725	2929	7253	6990	7080	7720
sat		0	0	0	0	0	3652	3657	3651	3841
unsat		2129	2256	2393	2725	2929	3601	3333	3429	3879

Table 10 Overall number of solved benchmarks for each configuration. VBS corresponds to the virtual best solver over all three configurations.

References

1 Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

2 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2023.

3 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.7. Technical report, Department of Computer Science, The University of Iowa, 2025. Available at `www.SMT-LIB.org`.

4 Clark Barrett, A. Stump, and Cesare Tinelli. The SMT-LIB standard - version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories, Edinburgh, Scotland,(SMT '10)*, 2010.

5 Nikolaj S. Bjørner and Mark C. Pichora. Deciding fixed and non-fixed size bit-vectors. In *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 1998.

6 Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Ziyad Hanna, Zurab Khasidashvili, Amit Palti, and Roberto Sebastiani. Encoding RTL constructs for MathSAT: a preliminary report. *Electronic Notes in Theoretical Computer Science*, 144(2):3–14, 2006.

7 Martin Brain, Florian Schanda, and Youcheng Sun. Building better bit-blasting for floating-point problems. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings,*

Part I, volume 11427 of *Lecture Notes in Computer Science*, pages 79–98. Springer, 2019. doi:10.1007/978-3-030-17462-0_5.

8 Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*, pages 741–746. IEEE, 2002.

9 Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23*, pages 58–75. Springer, 2017.

10 Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Satisfiability modulo transcendental functions via incremental linearization. In *Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pages 95–113. Springer, 2017.

11 The Coq development team. The coq proof assistant reference manual version 8.9, 2019. URL: https://coq.inria.fr/distrib/current/refman/.

12 Burak Ekici, Arjun Viswanathan, Yoni Zohar, Cesare Tinelli, and Clark W. Barrett. Formal verification of bit-vector invertibility conditions in Coq. In Uli Sattler and Martin Suda, editors, *Frontiers of Combining Systems - 14th International Symposium, FroCoS 2023, Prague, Czech Republic, September 20-22, 2023, Proceedings*, volume 14279 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2023. doi:10.1007/978-3-031-43369-6_3.

13 Herbert Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

14 Florian Frohn and Jürgen Giesl. Satisfiability modulo exponential integer arithmetic. In *IJCAR (1)*, volume 14739 of *Lecture Notes in Computer Science*, pages 344–365. Springer, 2024.

15 Aarti Gupta and Allan L. Fisher. Parametric circuit representation using inductive boolean functions. In *CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 1993.

16 Aarti Gupta and Allan L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *ICCAD*, pages 192–199. IEEE Computer Society / ACM, 1993.

17 Fuqi Jia, Rui Han, Pei Huang, Minghao Liu, Feifei Ma, and Jian Zhang. Improving bit-blasting for nonlinear integer constraints. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 14–25, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3597926.3598034.

18 Henry S. Warren Jr. *Hacker's Delight, Second Edition*. Pearson Education, 2013.

19 Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. IsaRare: Automatic verification of SMT rewrites in Isabelle/HOL. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 311–330. Springer, 2024. doi:10.1007/978-3-031-57246-3_17.

20 Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *PLDI*, pages 22–32. ACM, 2015.

21 Makai Mann, Amalee Wilson, Yoni Zohar, Lindsey Stuntz, Ahmed Irfan, Kristopher Brown, Caleb Donovick, Allison Guman, Cesare Tinelli, and Clark W. Barrett. Smt-Switch: A solver-agnostic C++ API for SMT solving. In *SAT*, volume 12831 of *Lecture Notes in Computer Science*, pages 377–386. Springer, 2021.

22 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

23 Alexander Nadel. Bit-vector rewriting with automatic rule generation. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna*

Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, volume 8559 of *Lecture Notes in Computer Science*, pages 663–679. Springer, 2014. doi:10.1007/978-3-319-08867-9_44.

24 Aina Niemetz and Mathias Preiner. Ternary propagation-based local search for more bit-precise reasoning. In *FMCAD*, pages 214–224. IEEE, 2020.

25 Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2023. doi:10.1007/978-3-031-37703-7_1.

26 Aina Niemetz and Mathias Preiner. Correction: Ternary Propagation-Based Local Search for more Bit-Precise Reasoning. https://bitwuzla.github.io/data/fmcad2020/fmcad2020-correction.pdf, 2025.

27 Aina Niemetz, Mathias Preiner, and Armin Biere. Propagation based local search for bit-precise reasoning. *Formal Methods Syst. Des.*, 51(3):608–636, 2017.

28 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. On solving quantified bit-vector constraints using invertibility conditions. *Formal Methods Syst. Des.*, 57(1):87–115, 2021. URL: https://doi.org/10.1007/s10703-020-00359-9, doi:10.1007/S10703-020-00359-9.

29 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark W. Barrett, and Cesare Tinelli. Towards bit-width-independent proofs in SMT solvers. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 366–384. Springer, 2019. doi:10.1007/978-3-030-29436-6_22.

30 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark W. Barrett, and Cesare Tinelli. Towards satisfiability modulo parametric bit-vectors. *J. Autom. Reason.*, 65(7):1001–1025, 2021.

31 Aina Niemetz, Mathias Preiner, and Yoni Zohar. Scalable bit-blasting with abstractions. 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 178–200. Springer, 2024. doi:10.1007/978-3-031-65627-9_9.

32 Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Science & Business Media, 2002.

33 Andres Nötzli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W Barrett, and Cesare Tinelli. Reconstructing fine-grained proofs of rewrites using a domain-specific language. In *FMCAD*, pages 65–74, 2022.

34 Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark W. Barrett, and Cesare Tinelli. Syntax-guided rewrite rule enumeration for SMT solvers. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, pages 279–297. Springer, 2019.

35 Mark Christopher Pichora. *Automated Reasoning about Hardware Data Types Using Bit-Vectors of Symbolic Lengths*. PhD thesis, University of Toronto, CAN, 2003. AAINQ84686.

36 Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2019.

37 Cesare Tinelli and Calogero G. Zarba. Combining decision procedures for sorted theories. In Jóse Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence*, pages 641–653, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

38 Zhihong Zeng, Priyank Kalla, and Maciej Ciesielski. LPSAT: a unified approach to RTL satisfiability. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 398–402. IEEE, 2001.

39 Yoni Zohar, Ahmed Irfan, Makai Mann, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. Bit-precise reasoning via int-blasting. In *VMCAI*, volume 13182 of *Lecture Notes in Computer Science*, pages 496–518. Springer, 2022.