Bar-Ilan University

Bit-precise Reasoning with Parametric Bit-Vectors

Zvi Berger

Submitted in partial fulfilment of the requirements for the Master's Degree in the
Department of Computer Science,
Bar Ilan University

Ramat Gan, Israel 2025

This work was carried out under the supervision of Dr. Yoni Zohar, Department of Computer Science, Bar-Ilan University.

Contents

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. In this thesis, we extend and improve the existing 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 parametric bit-widths themselves. Algorithmically, we introduce a lazy algorithm that avoids quantifiers and uninterpreted functions. Empirically, we show a significant improvement by implementing and evaluating our approach, as well as comparing it to the previous one.

1 Introduction

Bit-precise reasoning as provided by Satisfiability Modulo Theories (SMT) for the theory of fixed-size bit-vectors [2] 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). Further, some versions of decision procedures for other theories, e.g., floating-point arithmetic and non-linear integer arithmetic, are based on reducing the original problem to this theory [12, 19].

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 bit-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 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. Further, in the development of decision procedures for this theory, a large amount of effort is dedicated to the design of *rewrite rules* and *lemmas* [26,30,33,36]. Before adding new rules or lemmas, 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 more lengthy process of developing interactive proofs [14, 22]) is highly beneficial for the development process. However, current state-of-the-art SMT solvers do not support reasoning over bit-vectors of parametric size. Thus, it is common to verify the correctness of such rewrite rules and lemmas with SMT solvers for the theory of fixed-size bit-vectors up to a reasonably large bit-width. This does not yield a proof of correctness in general but increases confidence that these rules and lemmas are also correct for larger bit-widths [30, 33].

Early work on bit-precise reasoning supported reasoning about parametric bit-vectors but only for a small fragment of the theory (see, e.g., [10, 17, 18, 37]). 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. [31, 32]. 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. Further, [32] introduced a theory of parametric bit-vectors in which it was

impossible to reason about bit-widths, as they were implicit to the signature of the theory. The translation was implemented outside of an SMT solver and evaluated on three case studies: bit-width independent verification conditions for compiler optimizations [23], invertibility conditions [30] and rewrite rules [36]. And 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. In particular, it was tailored to the specific case studies and did not support arbitrary input problems over parametric bit-vectors.

In later work [40], Zohar et al. presented a translation of *fixed-size* bit-vectors to quantifier-free non-linear integer arithmetic with uninterpreted functions (coined *int-blasting*) with the goal of improving the scalability of reasoning over large bit-vectors. Unlike [32], the approach of [40] 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 thesis, we propose a procedure for reasoning about parametric bit-vectors based on the eager approach presented in [32]. 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 [40]. 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 [32] 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 [32]) and generalizes the lazy handling of bitwise *and* from [40] to parametric bit-vectors. Additionally, instead of eagerly axiomatizing exponentiation with base 2 (as in [32]), we handle this operator lazily.

(*iii*) We provide an implementation of a solver for parametric bit-vectors. The implementation is generic in the sense that it supports reasoning over arbitrary parametric bit-vector formulas (as opposed to the implementation presented in [32]). In particular, reasoning over bit-widths and constraints over multiple bit-widths and operators that manipulate bit-widths (such as extraction, concatenation and extension) is fully supported.

(*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 [32].

For brevity and simplicity, and similarly to previous work [32, 40], our formal presentation focuses on *quantifier-free* formulas. We stress however, that our implementation fully supports quantifiers, and in fact one of the benchmark sets in our experimental evaluation contains quantified formulas.

The remainder of this thesis is organized as follows. In Section 2, we provide the necessary background. 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

This section outlines the formal foundations and logical frameworks relevant to our analysis. We review core concepts in propositional logic, first-order logic, satisfiability problems (SAT and SMT), and the theory of fixed-width bit-vectors and the theory of integers, providing the groundwork for the techniques discussed in later sections.

2.1 Propositional Logic and First-Order Logic

A propositional variable is a Boolean variable that can be assigned True or False. Let S be a set of Boolean variables. For every variable $x \in S$ we define two literals: a positive literal x and a negative literal $\overline{x}$. Also, we define a clause as a finite disjunction of literals. A formula φ is in a Conjunctive Normal Form (CNF) iff φ is a finite conjunction of clauses.

First-Order Logic (FOL) extends Propositional Logic (PL) with predicates, functions, and quantifiers [11]. We briefly review the notions of many-sorted first-order theories with equality (see [15, 39] 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.

2.2 SAT and SMT

The **propositional satisfiability problem** is to decide for a given propositional formula φ in CNF, whether it is satisfiable. This problem is called SAT (see [9] for more details) and it is NP-complete. The **Satisfiability Modulo Theories (SMT)** (see [6] for more z details) problem is similar to SAT, but the given formula φ that we need to determine if it is satisfiable is in a first-order theory. The formula can be in one theory, a union of two theories, or more. Whereas SAT is NP-complete but decidable, SMT problems can be undecidable (depending on the theory).

SMT-solvers can be classified into two main approaches: *eager* and *lazy* [7]. In the eager approach, the SMT-solver translates the formula to an equivalent propositional formula if such exists and sends it to a SAT-solver. In the lazy approach, the SMT-solver uses a SAT-solver to reason about the Boolean structure of the formula and uses a solver for the theory T to reason about conjunctions of literals in that theory. So, in the lazy approach, we use two solvers: one for the specific theory T and a SAT-solver. Notice that every theory T may need a different approach to be solved.

An example of a common theory is the uninterrupted functions theory (UF). In this theory, we can solve formulas with function symbols that do not have a fixed semantics.

2.3 SMT-LIB

The Satisfiability Modulo Theories Library (SMT-LIB) is an international initiative aimed at facilitating research and development in SMT. The SMT-LIB standard is a common standard for benchmarks for SMT [2]. SMT-LIB does not support parametric bit-vectors. We extended the SMT-LIB

language to support PBV formulas; see Figure 1 for an example.

```
(set-logic ALL)
(declare-const k Int)
(declare-const m Int)
(dcelare-const x (_ BitVec k))
(dcelare-const y (_ BitVec m))
(assert (bvult x (bvadd x y)))
(assert (= y (bvand x y)))
(check-sat)
```

Figure 1: A formula over parametric bit-vectors in SMT-LIB style syntax.

2.4 Fixed Width Bit-Vectors Theory

The theory $T_{\mathrm{BV}} = (\Sigma_{\mathrm{BV}}, I_{\mathrm{BV}})$ of fixed-size bit-vectors as defined in the SMT-LIB 2 standard [4] consists of the class of interpretations I_{BV} and signature Σ_{BV}, which includes a unique sort for each positive integer n (representing the bit-vector width), denoted here as $\sigma_{[n]}$. We consider a restricted set of bit-vector function and predicate symbols (or *bit-vector operators*) as listed in Table 1. For a given positive integer n, the domain ${\sigma_{[n]}}^{\mathcal{I}}$ of sort $\sigma_{[n]}$ in $\mathcal{I}$ is the set of all bit-vectors of size n. We assume that Σ_{BV} includes all *bit-vector constants* of sort $\sigma_{[n]}$ for each n, represented as bit-strings. However, to simplify the notation we will sometimes denote them by the corresponding natural number in $\{0, \ldots, 2^{n-1}\}$. All interpretations $\mathcal{I} \in I_{\mathrm{BV}}$ are identical except for the value they assign to variables. They interpret sort and function symbols as specified in SMT-LIB 2. All function symbols of non-zero arity in Σ^f_{BV} are overloaded for each $\sigma_{[n]} \in \Sigma^s_{\mathrm{BV}}$ and their semantics is defined according to the SMT-LIB standard.

Symbol	SMT-LIB Syntax	Sort
$\approx$, $\not\approx$	=, distinct	$\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$
$<_\mathrm{u}$, $>_\mathrm{u}$, $<_\mathrm{s}$, $>_\mathrm{s}$	bvult, bvugt, bvslt, bvsgt	$\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$
$\leq_\mathrm{u}$, $\geq_\mathrm{u}$, $\leq_\mathrm{s}$, $\geq_\mathrm{s}$	bvule, bvuge, bvsle, bvsge	$\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$
$\sim$, $-^{\mathcal{B}}$	bvnot, bvneg	$\sigma_{[n]} \to \sigma_{[n]}$
&, |, $\oplus$	bvand, bvor, bvxor	$\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$
$\ll$, $\gg$, $\gg_a$	bvshl, bvlshr, bvashr	$\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$
$+^{\mathcal{B}}$, $\cdot^{\mathcal{B}}$, $\mathrm{mod}^{\mathcal{B}}$, $\mathrm{div}^{\mathcal{B}}$	bvadd, bvmul, bvurem, bvudiv	$\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$
$[u:l]$	extract $(0 \leq l \leq u < n)$	$\sigma_{[n]} \to \sigma_{[u-l+1]}$
$\circ$	concatenation	$\sigma_{[n]} \times \sigma_{[m]} \to \sigma_{[n+m]}$

Table 1: Considered bit-vector operators with SMT-LIB 2 syntax.

Bit-Blasting The state of the art approach of SMT-solvers for the BV-theory is called bit-blasting [21]. Bit-blasting is an eager translation from a formula with bit-vectors to a propositional formula. The translated formula is sent to a SAT-solver. Even though bit-blasting is a highly efficient approach, since modern SAT-solvers succeed to solve very complex propositional formulas, bit-blasting works only with fixed-width bit-vectors. The problem with bit-blasting with bit-vectors that do not have a fixed-width is that the translation would need to be a formula with an infinite length.

2.5 Integer Arithmetic Theory

The theory $T_{IA} = (\Sigma_{IA}, I_{IA})$ of integer arithmetic is defined as in the SMT-LIB 2 standard [3,5]. Its signature Σ_{IA} is shown in Table 2 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.

I_{IA} is the class of all Σ_{IA}-interpretations I that interpret the arithmetic operators as the usual integer operators, as defined in SMT-LIB 2.

Symbol	SMT-LIB Syntax	Sort
$\approx_{\mathsf{Int}}, \not\approx_{\mathsf{Int}}$	`=`, `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 2: Signature Σ_{IA} of the theory of integer arithmetic T_{IA}.

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 $\mathcal{F}$-free fragment is a T_{IA}-interpretation (i.e., the symbols in $\mathcal{F}$ are freely interpreted).

For conciseness, we may remove set braces from this notation when listing the elements of $\mathcal{F}$, e.g., by writing $\Sigma_{IA}(f, g)$ instead of $\Sigma_{IA}(\{f, g\})$. Their semantics in this case is left unspecified in the SMT-LIB 2 standard, and they are thus treated as uninterpreted functions.

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. [32] 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). Further, 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 [32], 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 $|\sqcup|$ and introduce an explicit representation of parametric bit-vector constants via the conversion operator *topbv*. 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 3. 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, $|\sqcup|$ of arity $\mathsf{PBV} \to \mathsf{Int}$ and *topbv* 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

Symbol	**SMT-LIB Syntax**	**Sort**
$\Sigma_{PBV} = \Sigma_{IA}$ + 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`	$\mathsf{PBV} \to \mathsf{PBV}$
$\&$, $\mid$, $\oplus$	`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`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$\sqcup[\sqcup:\sqcup]$	`pextract`	$\mathsf{PBV} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{PBV}$
$\circ$	`concat`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
ext_z	`pzero_extend`	$\mathsf{Int} \times \mathsf{PBV} \to \mathsf{PBV}$
ext_s	`psign_extend`	$\mathsf{Int} \times \mathsf{PBV} \to \mathsf{PBV}$
$\lvert\sqcup\rvert$	`bvsize`	$\mathsf{PBV} \to \mathsf{Int}$
$topbv$	`int_to_pbv`	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{PBV}$
$\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}}) = \Sigma_{IA}$ + 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 3: Signatures Σ_{PBV} and $\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$, defined as extensions of signature Σ_{IA}.

positive widths. The operators in Σ_{IA} are interpreted in the same way as in T_{IA}. The new symbol $\lvert\sqcup\rvert$ is interpreted as the function that maps each bit-vector x to its bit-width expressed as an integer. The new symbol $topbv$ is interpreted as the function that maps any two integers n and m to the bit-vector of size n that represents $m \bmod 2^n$ if $n > 0$, and to an arbitrary bit-vector otherwise. 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 $\sqcup[\sqcup:\sqcup]$ denotes the function that maps a bit-vector x, an integer i, 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 ext_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, non-negative 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 argument, 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. That is, in the example above, x and y are indeed interpreted as bit-vectors of the same bit-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 type system, we therefore define a notion of *admissible* T_{PBV}-satisfiability that excludes such spurious interpretations.

Definition 1. *Let φ be a Σ_{PBV}-formula and $\mathcal{I}$ a T_{PBV}-interpretation. Let* Type *be a function as defined in Algorithm 1. Interpretation $\mathcal{I}$ is* admissible w.r.t. *φ if $\mathcal{I} \models \textsc{Type}(\varphi)$. 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 Type defined in Algorithm 1. Function Type collects admissibility constraints while traversing over a given Σ_{PBV}-term. It is defined via a function K as given in Algorithm 2, which con-

Algorithm 1 Function TYPE 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 ite_{Int}. Symbol $\diamond$ ranges over symbols in Σ_{PBV} not explicitly handled otherwise.

function $\textsc{Type}(e)$
match e:

x	$\rightarrow$	$\mathrm{K}(e) > 0$
z	$\rightarrow$	$\top$
$topbv(k,t)$	$\rightarrow$	$\mathrm{K}(e) > 0 \wedge \textsc{Type}(t)$
$\lvert t \rvert$	$\rightarrow$	$\textsc{Type}(t)$
$t[i:j]$	$\rightarrow$	$0 \leq j \leq i < \mathrm{K}(t) \wedge \textsc{Type}(t)$
$ext_z(n,t)$	$\rightarrow$	$n \geq 0 \wedge \textsc{Type}(t)$
$ext_s(n,t)$	$\rightarrow$	$n \geq 0 \wedge \textsc{Type}(t)$
$t_1 \circ t_2$	$\rightarrow$	$\textsc{Type}(t_1) \wedge \textsc{Type}(t_2)$
$ite_{\mathsf{PBV}}(t_1,t_2,t_3)$	$\rightarrow$	$\mathrm{K}(t_2) \approx \mathrm{K}(t_3) \wedge \bigwedge_{i=1}^{3} \textsc{Type}(t_i)$
$\bullet(t_1,\ldots,t_n)$	$\rightarrow$	$\bigwedge_{i=1}^{n} \textsc{Type}(t_i)$
$\diamond(t_1,\ldots,t_n)$	$\rightarrow$	$(\bigwedge_{i=2}^{n} \mathrm{K}(t_1) \approx \mathrm{K}(t_i)) \wedge (\bigwedge_{i=1}^{n} \textsc{Type}(t_i))$

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

function $\mathrm{K}(e)$
match e:

x	$\rightarrow$	$\lvert x \rvert$
$topbv(k,t)$	$\rightarrow$	k
$t[i:j]$	$\rightarrow$	$i - j + 1$
$ext_z(n,t)$	$\rightarrow$	$\mathrm{K}(t) + n$
$ext_s(n,t)$	$\rightarrow$	$\mathrm{K}(t) + n$
$t_1 \circ t_2$	$\rightarrow$	$\mathrm{K}(t_1) + \mathrm{K}(t_2)$
$ite_{\mathsf{PBV}}(t_1,t_2,t_3)$	$\rightarrow$	$\mathrm{K}(t_2)$
$\diamond(t_1,\ldots,t_n)$	$\rightarrow$	$\mathrm{K}(t_1)$

structs an integer term representing the symbolic bit-width of a parametric bit-vector term.

Example 1. *Consider a formula φ given as $y \approx z \circ w$. The admissibility condition $\textsc{Type}(\varphi)$ is determined as $\lvert y \rvert \approx \lvert z \rvert + \lvert w \rvert \wedge \lvert y \rvert > 0 \wedge \lvert z \rvert > 0 \wedge \lvert w \rvert > 0$. 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 $\textsc{Type}(\varphi')$ is defined as $\lvert x \rvert \approx \lvert x \rvert \wedge \lvert x \rvert > 0 \wedge \lvert x \rvert \approx \lvert x \rvert + \lvert x \rvert$ (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'.

Note that our notion of admissibility corresponds to the notion of admissibility introduced by Niemetz et al. [32]. However, its definition differs in one key aspect. In [32], admissibility was enforced via auxiliary functions and 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 to the signature, whereas in our definition of Σ_{PBV}, we explicitly include operators $|\sqcup|$ and *topbv*. 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 as illustrated in Figure 2. Its main component is the translation of Σ_{PBV}-formulas to formulas over a signature $\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$. As defined in Table 3, signature $\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$ extends Σ_{IA} with two new function symbols, pow_2 and $\&^{\mathbb{N}}$.

We define a theory $T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$ over $\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$, where symbols pow_2 and $\&^{\mathbb{N}}$ are interpreted *arbitrarily*. Additionally, for the purpose of the translation, we consider a theory $T_{IA}(\mathsf{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$ over $\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$ in which the interpretation of pow_2 and $\&^{\mathbb{N}}$ is *fixed* and defined as follows. The term $\mathsf{pow}_2(n)$ 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 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 2. *Notice that pow_2 is a function symbol defined over the integers. Thus, a negative exponent $-n$ will always produce 0 due to the integer division in $2^{-n} \equiv \frac{1}{2^n}$ truncating towards zero. Further notice that $\&^{\mathbb{N}}$ defines a special case for bit-width $k \leq 0$. Our translation will never produce $\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$-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 reality, for our use case, these special cases could also be interpreted arbitrarily. However, we chose a fixed interpretation since it is easier to implement.*

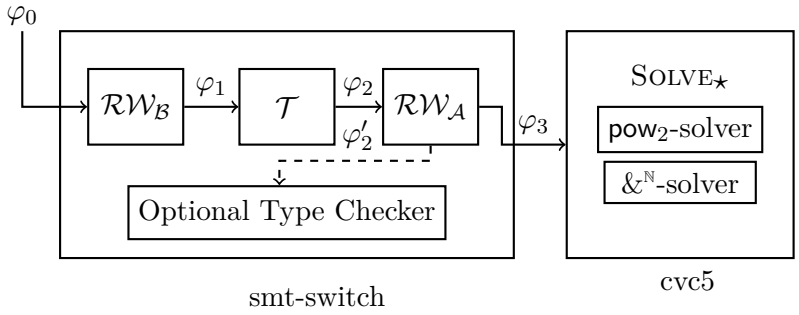

Figure 2: A solver for admissible T_{PBV}-satisfiability.

Starting with a Σ_{PBV}-formula φ_0, we first simplify the formula by applying rewrite rules $\mathcal{RW}_\mathcal{B}$, as described in Section 4.1. The rewritten Σ_{PBV}-formula φ_1 is then translated to a $\Sigma_{IA}(\mathsf{pow}_2, \&^\mathbb{N})$-formula φ_2 using function $\mathcal{T}$, as described in Section 4.2. After translation, we use an arithmetic rewriter $\mathcal{RW}_\mathcal{A}$ to produce φ_3, as described in Section 4.3. Optionally, another formula φ_2' is passed to a type checker. Finally, the $T_{IA}(\mathsf{pow}_{2_\star}, \&_\star^\mathbb{N})$-satisfiability of φ_3 is determined via procedure SOLVE$_\star$, described in Section 4.4.

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

State-of-the-art SMT solvers heavily apply simplification techniques on the input formula prior to the solving process, with respect to the actual solving procedure. One such technique is based on term rewriting. And, especially 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 are not generally useful for our T_{PBV} procedure, which relies on a translation to integer arithmetic.

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]. Table 4 presents all the rewrite rules applied in our approach. Rewrite rules implemented in cvc5 are documented in the domain-specific language RARE [35], 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.

Example 2. *Consider a T_{BV}-rule that rewrites the term $(x \,\&\, y)[i:j]$ to $(x[i:j]) \,\&\, (y[i:j])$. 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 $\mathcal{T}$

The translation function $\mathcal{T}$ is given in Algorithms 3 and 4. It is based on [32], with changes highlighted in blue.

Prior to the actual conversion step, Σ_{PBV}-formula φ_1 is augmented, conjunctively, with the admissibility constraints $\textsc{Type}(\varphi_1)$, constructed as described in Algorithm 1, as well as the *range and size constraints* $\textsc{Range}(\varphi_1)$. The latter are defined over integer constants that either represent parametric bit-vector constants or symbolic bit-widths, as described in Algorithm 4. The augmented Σ_{PBV}-formula is then converted via function C into a $\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$-formula as defined in Algorithm 3. We assume a one-to-one mapping χ from PBV-constants to Int-constants, as well as a one-to-one mapping κ from PBV-terms to Int-constants, such that their images are disjoint. We use χ to translate Σ_{PBV}-constants of sort PBV to fresh Int-constants. Mapping κ is used for translating terms of the form $|t|$ to fresh integer constants that represent the symbolic size of the parametric bit-vector term t. Compared to our previous work [32], our new translation handles the new operators $|_{\sqcup}|$ and *topbv*, improves the encodings of operators $\{\gg, |, \oplus\}$ to eliminate applications of mod, and adds support for operators $\{[:], ext_s, ext_z\}$. Notice that the encoding of $\gg_a$ is not explicitly described, as it can be expressed using $\gg$ and *ite*.

Following Zohar et al. [40], we do not handle $|$ and $\oplus$ natively. Instead, we eliminate them by means of $\&$, $+^{\mathcal{B}}$ and $-^{\mathcal{B}}$. This elimination is embedded in the translations of these operators and improves upon the original elimination in [40] by avoiding the introduction of additional mod operations. It can be shown that the translated terms are in the correct range thanks to the result below.

Lemma 1. *Suppose $k > 0$ and $0 \leq x, y < 2^k$. Then, $0 \leq x+y-\&^{\mathbb{N}}(k,x,y) < 2^k$ and $0 \leq x + y - 2 \cdot \&^{\mathbb{N}}(k,x,y) < 2^k$.*

Example 3. *Consider the formula φ from Example 1. Then, $\mathcal{T}(\varphi) = C(\varphi \wedge \textsc{Range}(\varphi) \wedge \textsc{Type}(\varphi))$. The result of $C(\varphi)$ is $y' \approx z' \cdot \mathsf{pow}_2(k_w) + w'$, where $y' = \chi(y)$, $z' = \chi(z)$, $w' = \chi(w)$, and $k_w = \kappa(w)$. The result of*

rule name	term	simplify
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 k1 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 k1 0))	x
bv-mul-one	(bvmul x (int_to_pbv k1 1))	x
bv-mul-zero	(bvmul x (int_to_pbv k1 0))	(int_to_pbv k1 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 k1 0))	x
bv-shl-by-const-2	(bvshl x (int_to_pbv k1 k1))	(int_to_pbv k1 0)
bv-lshr-by-const-0	(bvlshr x (int_to_pbv k1 0))	x
bv-lshr-by-const-2	(bvlshr x (int_to_pbv k1 k1))	(int_to_pbv k1 0)
bv-ashr-by-const-0	(bvashr x (int_to_pbv k1 0))	x
bv-bitwise-idemp-2	(bvor x x)	x
bv-or-one	(bvor x (bvnot x))	(bvnot (int_to_pbv k1 0))
bv-xor-duplicate	(bvxor x x)	(int_to_pbv k1 0)
bv-xor-zero	(bvxor x (int_to_pbv k1 0))	x
bv-bitwise-not-or	(bvor x (bvnot x))	(bvnot (int_to_pbv k1 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 k1 0) x)	(not (= x (int_to_pbv k1 0)))
bv-ult-zero-2	(bvult x (int_to_pbv k1 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 k1 0))	(= x (int_to_pbv k1 0))
bv-zero-ule	(bvule (int_to_pbv k1 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 k1 0))	(bvnot (int_to_pbv k1 0))
bv-udiv-one	(bvudiv x (int_to_pbv k1 1))	x
bv-urem-one	(bvurem x (int_to_pbv k1 1))	(int_to_pbv k1 0)
bv-urem-self	(bvurem x x)	(int_to_pbv k1 0)
bv-shl-zero	(bvshl (int_to_pbv k1 0) x)	(int_to_pbv k1 0)
bv-lshr-zero	(bvlshr (int_to_pbv k1 0) x)	(int_to_pbv k1 0)
bv-ashr-zero	(bvashr (int_to_pbv k1 0) x)	(int_to_pbv k1 0)
bv-ult-one	(bvult x (int_to_pbv k1 1))	(= x (int_to_pbv k1 0))

Table 4: Rewrite rules for the bit-vector rewriter. k1 is the bitwidth of x. In addition, we employed a lexicographic ordering between terms, $>_{lex}$, and normalized commutative operations (such as $+^{\mathcal{B}}$ &, etc.) according to this relation. For example, if $t_1 <_{lex} t_2$ then we rewrite $t_1 +^{\mathcal{B}} t_2$ to itself and $t_2 +^{\mathcal{B}} t_1$ to $t_1 +^{\mathcal{B}} t_2$.

Algorithm 3 Translation function $\mathcal{T}$. We use x for PBV constants, z for Int constants, $\mathsf{uts}(k, z)$ for $2 \cdot (z \bmod \mathsf{pow}_2(k-1)) - z$, symbol $\bowtie \in \{<, \leq, >, \geq\}$, and $\bullet$ for *ite*, logical connectives and symbols in Σ_{IA}.

function $\mathcal{T}(\varphi)$
  return $\mathrm{C}(\varphi \wedge \textsc{Range}(\varphi) \wedge \textsc{Type}(\varphi))$

function $\mathrm{C}(e)$
  match e:

z	$\rightarrow$	z
$\lvert t \rvert$	$\rightarrow$	$\kappa(t)$
$topbv(k, t)$	$\rightarrow$	$\mathrm{C}(t) \bmod \mathsf{pow}_2(k)$
x	$\rightarrow$	$\chi(x)$
$t_1 \approx t_2$	$\rightarrow$	$\mathrm{C}(t_1) \approx \mathrm{C}(t_2)$
$\bowtie_u (t_1, t_2)$	$\rightarrow$	$\mathrm{C}(t_1) \bowtie \mathrm{C}(t_2)$
$\bowtie_s (t_1, t_2)$	$\rightarrow$	$\mathsf{uts}(\kappa(t_1), \mathrm{C}(t_1)) \bowtie \mathsf{uts}(\kappa(t_2), \mathrm{C}(t_2))$
$t_1 +^{\mathcal{B}} t_2$	$\rightarrow$	$(\mathrm{C}(t_1) + \mathrm{C}(t_2)) \bmod \mathsf{pow}_2(\kappa(t_1))$
$t_1 -^{\mathcal{B}} t_2$	$\rightarrow$	$(\mathrm{C}(t_1) - \mathrm{C}(t_2)) \bmod \mathsf{pow}_2(\kappa(t_1))$
$t_1 \cdot^{\mathcal{B}} t_2$	$\rightarrow$	$(\mathrm{C}(t_1) \cdot \mathrm{C}(t_2)) \bmod \mathsf{pow}_2(\kappa(t_1))$
$t_1 \mathrm{div}^{\mathcal{B}} t_2$	$\rightarrow$	$ite(\mathrm{C}(t_2) \approx 0, \mathsf{pow}_2(\kappa(t_1)) - 1, \mathrm{C}(t_1) \mathrm{div} \mathrm{C}(t_2))$
$t_1 \mathrm{mod}^{\mathcal{B}} t_2$	$\rightarrow$	$ite(\mathrm{C}(t_2) \approx 0, \mathrm{C}(t_1), \mathrm{C}(t_1) \bmod \mathrm{C}(t_2))$
$\sim t$	$\rightarrow$	$\mathsf{pow}_2(\kappa(t)) - (\mathrm{C}(t) + 1)$
$-^{\mathcal{B}} t$	$\rightarrow$	$(\mathsf{pow}_2(\kappa(t)) - \mathrm{C}(t)) \bmod \mathsf{pow}_2(\kappa(t))$
$t_1 \ll t_2$	$\rightarrow$	$(\mathrm{C}(t_1) \cdot \mathsf{pow}_2(\mathrm{C}(t_2))) \bmod \mathsf{pow}_2(\kappa(t_1))$
$t_1 \gg t_2$	$\rightarrow$	$\mathrm{C}(t_1) \mathrm{div} \mathsf{pow}_2(\mathrm{C}(t_2))$
$t_1 \,\&\, t_2$	$\rightarrow$	$\&^{\mathbb{N}}(\kappa(t_1), \mathrm{C}(t_1), \mathrm{C}(t_2))$
$t_1 \mid t_2$	$\rightarrow$	$\mathrm{C}(t_1) + \mathrm{C}(t_2) - \&^{\mathbb{N}}(\kappa(t_1), \mathrm{C}(t_1), \mathrm{C}(t_2))$
$t_1 \oplus t_2$	$\rightarrow$	$\mathrm{C}(t_1) + \mathrm{C}(t_2) - 2 \cdot \&^{\mathbb{N}}(\kappa(t_1), \mathrm{C}(t_1), \mathrm{C}(t_2))$
$t_1 \circ t_2$	$\rightarrow$	$\mathrm{C}(t_1) \cdot \mathsf{pow}_2(\kappa(t_2)) + \mathrm{C}(t_2)$
$t[i:j]$	$\rightarrow$	$(\mathrm{C}(t) \mathrm{div} \mathsf{pow}_2(j)) \bmod \mathsf{pow}_2(i - j + 1)$
$ext_z(n, t)$	$\rightarrow$	$\mathrm{C}(t)$
$ext_s(n, t)$	$\rightarrow$	$ite(\quad \mathrm{C}(t[\kappa(t) - 1]) \approx 1,$ $(\mathsf{pow}_2(n) - 1) \cdot \mathsf{pow}_2(\kappa(t)) + \mathrm{C}(t), \mathrm{C}(t))$
$\bullet(t_1, \ldots, t_n)$	$\rightarrow$	$\bullet(\mathrm{C}(t_1), \ldots, \mathrm{C}(t_n))$

$C(\textsc{Range}(\varphi))$ is $0 \leq y' < \mathsf{pow}_2(k_y) \wedge 0 \leq z' < \mathsf{pow}_2(k_z) \wedge 0 \leq w' < \mathsf{pow}_2(k_w)$, where $k_y = \kappa(y)$ and $k_z = \kappa(z)$. The result of $\textsc{Type}(\varphi)$ is given in Example 1, and then C replaces $|y|$, $|z|$, and $|w|$ by k_y, k_z, and k_w, respectively.

4.3 The Arithmetic Rewriter $\mathcal{RW}_{\mathcal{A}}$

Our translation function $\mathcal{T}$ makes heavy use of mod and pow_2 operations. In practice, some of these applications can be safely eliminated to optimize

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

function RANGE(e)
  match e:

x	$\rightarrow$	$0 \leq \chi(x) < \mathsf{pow}_2(\kappa(x))$
$\lvert t \rvert$	$\rightarrow$	$e \approx \mathrm{K}(t) \wedge \textsc{Range}(t)$
z	$\rightarrow$	$\top$
$\diamond(t_1, \ldots, t_n)$	$\rightarrow$	$\bigwedge_{i=1}^{n} \textsc{Range}(t_i)$

the encoding. For example, consider the term $(x +^{\mathcal{B}} x) +^{\mathcal{B}} x$, where $\kappa(x) = k$. The result of C on this term is $(((x' + x') \bmod 2^k) + x') \bmod 2^k$, where $x' = \chi(x)$. Instead, we directly translate it to $((x' + x') + x') \bmod 2^k$. The complete set of optimizations is presented in Figure 3.

$\sharp^1_{rr}$	$((x \bmod \mathsf{pow}_2(k)) \sharp (y \bmod \mathsf{pow}_2(k))) \bmod \mathsf{pow}_2(k)$	$\rightarrow$	$(x \sharp y) \bmod \mathsf{pow}_2(k)$
$\sharp^2_{rr}$	$((x \bmod \mathsf{pow}_2(k)) \sharp y) \bmod \mathsf{pow}_2(k)$	$\rightarrow$	$(x \sharp y) \bmod \mathsf{pow}_2(k)$
$\sharp^3_{rr}$	$(x \sharp (y \bmod \mathsf{pow}_2(k))) \bmod \mathsf{pow}_2(k)$	$\rightarrow$	$(x \sharp y) \bmod \mathsf{pow}_2(k)$
i^1_{rr}	$(ite(b, x \bmod \mathsf{pow}_2(k), y \bmod \mathsf{pow}_2(k))) \bmod \mathsf{pow}_2(k)$	$\rightarrow$	$ite(b, x, y) \bmod \mathsf{pow}_2(k)$
i^2_{rr}	$(ite(b, x, y \bmod \mathsf{pow}_2(k))) \bmod \mathsf{pow}_2(k)$	$\rightarrow$	$ite(b, x, y) \bmod \mathsf{pow}_2(k)$
i^3_{rr}	$(ite(b, x \bmod \mathsf{pow}_2(k), y)) \bmod \mathsf{pow}_2(k)$	$\rightarrow$	$ite(b, x, y) \bmod \mathsf{pow}_2(k)$

Figure 3: $\mathcal{RW}_{\mathcal{A}}$ Optimizations. The first three rules apply to arithmetic operations $\sharp \in \{+, -, \cdot\}$.

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

After translating a Σ_{PBV}-formula to a $\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$-formula, we now check if it is $T_{IA}(\mathsf{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$-satisfiable with a dedicated CEGAR-style procedure $\textsc{Solve}_\star$ that iteratively refines over-approximations of pow_2 and $\&^{\mathbb{N}}$ as given in Algorithm 5. The procedure relies on a set of quantifier-free lemmas $\mathcal{L}$, which fully specifies the semantics of operators pow_2 and $\&^{\mathbb{N}}$ and is constructed by instantiating the (implicitly) universally quantified lemma schemas $\mathcal{L}_{\mathsf{pow}_2}$ (Table 5) and $\mathcal{L}_{\&^{\mathbb{N}}}$ (Table 6) for all pow_2-terms and $\&^{\mathbb{N}}$-terms in φ. It further assumes the availability of a subprocedure SOLVE to determine $T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$-satisfiability of a set of $\Sigma_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$-formulas Γ. Recall that in $T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$, symbols pow_2 and $\&^{\mathbb{N}}$ are uninterpreted and thus, Γ is an over-approximation of φ in $T_{IA}(\mathsf{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$. 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 with all lemmas that are not satisfied by $\mathcal{I}$.

Algorithm 5 Procedure for $T_{IA}(\mathsf{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$-satisfiability. We assume SOLVE is a procedure for $T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$-satisfiability that returns "sat" or "unsat" and an interpretation $\mathcal{I}$ for satisfiable formulas.

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

The set of lemmas $\mathcal{L}_{\mathsf{pow}_2}$ in Table 5 includes 4 lemmas from [32], 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 $T_{IA}(\mathsf{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$. Lemma bound strengthens a lemma from recent work proposing a CEGAR-style approach for a new SMT theory for integer arithmetic with exponentiation [16]. From there, we get that when $x \geq 3$, then $2 \cdot x + 1$ is a lower bound for $\mathsf{pow}_2(x)$. 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 value-based lemmas, instantiated with $v = x^{\mathcal{I}}$, to block concrete model values.

The set of lemmas $\mathcal{L}_{\&^{\mathbb{N}}}$ in Table 6 includes 8 lemmas that were introduced in [32] and capture basic properties of the $\&^{\mathbb{N}}$ operator. In addition, we consider 4 lemmas that did not appear in [32]. 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 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 $(x \text{ div } 2^i) \bmod 2$ as in Table 6. We observe that for a bit-width $m \geq n$ with $0 \leq x < 2^n$ and $0 \leq y < 2^n$, $\&^{\mathbb{N}}(m, x, y) \approx \&^{\mathbb{N}}(n, x, y)$, e.g., $\&^{\mathbb{N}}(5, 1, 1) \approx \&^{\mathbb{N}}(4, 1, 1)$. Lemma sum$_\geq$ is a generalization of this observation. It is instantiated with $v = k^{\mathcal{I}}$.

Note that procedure SOLVE$_\star$ resembles the lazy approach of [40], 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 [40] since there, all exponents were constants. Thus, exponentiation was simply evaluated. Finally, our sets of lemmas for $\&^{\mathbb{N}}$ and pow_2 expand both

Name	Lemma	Source
	$\mathsf{pow}_2(x) \in S$	
positive	$0 \leq x \Rightarrow \mathsf{pow}_2(x) > 0$	[32]
even	$x \geq 1 \Rightarrow \mathsf{pow}_2(x) \bmod 2 \approx 0$	[32]
div	$x \geq 0 \Rightarrow x \operatorname{div} \mathsf{pow}_2(x) \approx 0$	[32]
neg	$x < 0 \Rightarrow \mathsf{pow}_2(x) \approx 0$	new
bound	$(x \geq v \wedge v \geq 7) \Rightarrow v \cdot x + v^2 < \mathsf{pow}_2(x)$	new
value	$(0 \leq x \wedge x \approx v) \Rightarrow \mathsf{pow}_2(x) \approx 2^v$	new
	$\mathsf{pow}_2(x) \in S$, $\mathsf{pow}_2(y) \in S$	
monotonicity	$(0 \leq x \wedge x < y) \Rightarrow \mathsf{pow}_2(x) < \mathsf{pow}_2(y)$	[32]

Table 5: The set of lemma schemas defined by $\mathcal{L}_{\mathsf{pow}_2}(S)$ over a set of pow_2-terms S. The variables x and y in the formulas above, which are implicitly universally quantified, are instantiated for each pow_2-term in S. Symbol v is a schema variable standing for an arbitrary numeral.

the axioms from [32] and the lemmas from [40].

Theorem 1. *Let φ be a Σ_{PBV}-formula and let $\varphi' = \mathcal{RW}_{\mathcal{A}}(\mathcal{T}(\mathcal{RW}_{\mathcal{B}}(\varphi)))$. If $\textsc{Solve}_\star(\varphi')$ terminates, then φ is admissibly T_{PBV}-satisfiable if and only if $\textsc{Solve}_\star(\varphi')$ returns "sat".*

Putting all components described in this section together, we obtain an incomplete but sound procedure $\textsc{Solve}_\star(\mathcal{RW}_{\mathcal{A}}(\mathcal{T}(\mathcal{RW}_{\mathcal{B}}(\varphi))))$ for admissible T_{PBV}-satisfiability. The correctness argument for our procedure is based on the correctness arguments in [32, 40]. Extensions and generalizations compared to [32, 40] are sound, thus correctness is preserved.

4.5 Proof of Theorem 1

Proof. The arguments for correctness of this algorithm are similar to the correctness arguments from [32, 40]: assuming a T_{PBV}-interpretation that satisfies a Σ_{PBV}-formula φ, while also being admissible w.r.t. φ, we can construct a $T_{IA}(\mathsf{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$-interpretation that satisfies the translation of φ, as well as the relevant lemmas from Tables 5 and 6, and vice verse. Compared to [32], the assumption of the admissibility of ω^b is replaced by the satisfaction of $\textsc{Type}(\varphi)$. What is left to show is:

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.

Name	Lemma	Source
	$\&^{\mathbb{N}}(k,x,y) \in S$	
base	$(k > 0 \wedge x \approx 1 \wedge y \approx 1) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx 1$	[32]
max	$(k > 0 \wedge \langle x \rangle_k \wedge y \approx \mathsf{pow}_2(k) - 1) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx x$	[32]
min	$y \approx 0 \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx 0$	[32]
idem	$(k > 0 \wedge \langle x \rangle_k \wedge x \approx y) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx x$	[32]
contra	$(x + y) \bmod \mathsf{pow}_2(k) \approx \mathsf{pow}_2(k) - 1 \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx 0$	[32]
range	$0 \leq x \wedge 0 \leq y \Rightarrow 0 \leq \&^{\mathbb{N}}(k,x,y) \leq min(x,y)$	[32]
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	$(x \approx w \wedge y \approx z) \Rightarrow \&^{\mathbb{N}}(k,x,y) \approx \&^{\mathbb{N}}(k,z,w)$	[32]
diff	$(k > 0 \wedge z \approx w \wedge x \not\approx y \wedge \langle x \rangle_k \wedge \langle y \rangle_k \wedge \langle z \rangle_k) \Rightarrow (\&^{\mathbb{N}}(k,x,z) \not\approx y \vee \&^{\mathbb{N}}(k,y,w) \not\approx x)$	[32]

Table 6: The set of lemmas defined by $\mathcal{L}_{\&^{\mathbb{N}}}(S)$ over a set of $\&^{\mathbb{N}}$-terms S. The variables k, x, z, and w in the formulas above, which are implicitly universally quantified, are instantiated for each $\&^{\mathbb{N}}$-term in S. We use $\langle t \rangle_k$ to denote $0 \leq t < \mathsf{pow}_2(k)$, $min(x,y)$ to abbreviate $ite(x < y, x, y)$, $\mathsf{ex}_i(x)$ for $(x \text{ div } 2^i) \bmod 2$. Symbol v is a schema variable standing for an arbitrary numeral.

4. The rules of the $\mathcal{RW}_{\mathcal{B}}$ rewriter are T_{PBV}-valid.
5. The rules of the $\mathcal{RW}_{\mathcal{A}}$ rewriter are T_{IA}-valid.
6. All lemmas of Tables 5 and 6 are $T_{IA}(\mathsf{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-valid.

We prove the first item in Section 4.5.1. The second item is proven in Section 4.5.2. The third follows directly from the semantics specified by the SMT-LIB standard. For the forth, notice that we only use rewrite rules that are already used in cvc5, and are bit-width independent. The rewrites themselves appear in Table 4. For their correctness verification, see, e.g., [22]. The fifth is proven in Section 4.5.3. For the sixth, most lemmas are taken from [32]. From the new lemmas that we add, correctness is trivial for lemmas neg and value in the case of pow_2, and also for empty, lsb and one in the case of $\&^{\mathbb{N}}$. We prove the correctness of the remaining lemmas in Section 4.5.4. □

4.5.1 Translating right shift

It suffices to show that in every $T_{IA}(\mathsf{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-interpretation in which $\mathrm{C}(x) \geq 0$, $\mathsf{pow}_2(\mathrm{C}(y)) > 0$, and and $\mathsf{pow}_2(k) > \mathrm{C}(x)$ are satisfied, the expressions $\mathrm{C}(x)$ div $\mathsf{pow}_2(\mathrm{C}(y))$ and $(\mathrm{C}(x)$ div $\mathsf{pow}_2(\mathrm{C}(y)))$ mod $\mathsf{pow}_2(k)$ are interpreted the same. This is indeed the case, as every such interpretation satisfied $\mathrm{C}(x)$ div $\mathsf{pow}_2(\mathrm{C}(y)) \leq \mathrm{C}(x) < \mathsf{pow}_2(k)$.

4.5.2 Elimination of $|$ and $\oplus$

We start with Lemma 1:

Lemma 1. *Suppose* $k > 0$ *and* $0 \leq x, y < 2^k$. *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 the fixed interpretation given to them by all $T_{IA}(\mathsf{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-interpretations. This saves us from superscripting interpretations to these symbols, and is not needed since their interpretations are fixed.

1. First case: 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, $x + y - \&^{\mathbb{N}}(k,x,y) \geq 0$.

   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 $\&^{\mathbb{N}}(k,x,y) = \Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 1, 0)$. Notice that we can also have $y = \Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 1, 0) + \Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0)$.

   Now, we clearly have that $x + y - \&^{\mathbb{N}}(k,x,y)$ equals $\Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(x) + \Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 10) + \Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0) - \Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 1, 0)$, which is: $\Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(x) + \Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0)$. This is equal to $\Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = 1, 1, 0) + \Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0)$. Clearly, this must be at most $\Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = 1, 1, 0) + \Sigma_{i=0}^{k-1} 2^i \cdot ite(\mathsf{ex}_i(x) = 0, 1, 0)$, which equals $\Sigma_{i=0}^{k-1} 2^i$, which must be strictly smaller than 2^k.

2. Second case: 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 $x + y - 2 \cdot \&^{\mathbb{N}}(k,x,y) \geq 0$.

   As for the other direction: Clearly, since by the semantics of $\&^{\mathbb{N}}$, we always have $\&^{\mathbb{N}}(k,x,y) \geq 0$, we have: $x + y - 2 \cdot \&^{\mathbb{N}}(k,x,y) \leq x + y - \&^{\mathbb{N}}(k,x,y)$. By the previous item, $x + y - \&^{\mathbb{N}}(k,x,y) < 2^k$.

□

We now justify the elimination of $|$. In [20], there is the following equation, that we rely on:

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

for all bit-vectors x and y. We therefore 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 1.

The justification for the elimination of $\oplus$ is similar to that of $|$. In [20], there is also the following equation, that we rely on:

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

for all bit-vectors x and y. We therefore get, combining (1) and (2) that

$$x \oplus y \approx ((x +^{\mathcal{B}} y) -^{\mathcal{B}} (x \,\&\, y) -^{\mathcal{B}} (x \,\&\, y)$$

and so $x \oplus y \approx x +^{\mathcal{B}} y -^{\mathcal{B}} (2 \cdot^{\mathcal{B}} x \,\&\, y)$. 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: $0 \leq (x+y-2\cdot\&^{\mathbb{N}}(k, x, y)) < 2^k$, as stated in the second item of Lemma 1.

4.5.3 Correctness Of The Arithmetic Rewriter $\mathcal{RW}_{\mathcal{A}}$

In this section, we specify the optimizations that we have implemented to the translation of Algorithm 3. Figure 3 summarizes all the optimizations integrated into our approach.

We prove the soundness of these rewrite rules, by considering more generic lemmas, without mentioning, and so the lemmas that we show here are more general than the rewrite rules.

Since in all rules, the right-hand side of mod is $\mathsf{pow}_2(k)$ for some bit-width k, and TYPE always adds a constraint that $k > 0$, we may assume in the following lemmas that $C > 0$.

The following lemma establishes the validity of Rewrite $+^1_{rr}$ presented in Figure 3. The validity of $+^2_{rr}$ and $+^3_{rr}$ is shown similarly.

Lemma 2. $((A \bmod C) + (B \bmod C)) \bmod C = (A + B) \bmod C$

Proof. From the quotient remainder theorem, we can write A and B as:

$$A = C \cdot Q_1 + R_1$$

$$B = C \cdot Q_2 + R_2$$

where $0 \leq R_1 < C$, $0 \leq R_2 < C$ and Q_1, Q_2 are integers.
Let's calculate Left Hand Side:

$$((A \bmod C) + (B \bmod C)) \bmod C = (R_1 + R_2) \bmod C$$

Let's calculate Right Hand Side:

$$\begin{aligned}(A+B) \bmod C &= (C \cdot Q_1 + R_1 + C \cdot Q_2 + R_2) \bmod C \\ &= (C \cdot Q_1 + C \cdot Q_2 + R_1 + R_2) \bmod C \\ &= (C(Q_1 + Q_2) + R_1 + R_2) \bmod C \\ &= (R_1 + R_2) \bmod C\end{aligned}$$

So, we show that both, Left Hand Side and Right Hand Side are equal to $(R_1 + R_2) \bmod C$. □

The following lemma establishes the validity of Rewrite $+^2_{rr}$ presented in Figure 3.

Lemma 3. $((A \bmod C) + B) \bmod C = (A + B) \bmod C$

Proof. From the quotient remainder theorem, we can write A and B as:

$$A = C \cdot Q_1 + R_1$$

$$B = C \cdot Q_2 + R_2$$

where $0 \leq R_1 < C$, $0 \leq R_2 < C$ and Q_1, Q_2 are integers.
Let's calculate Left Hand Side:

$$\begin{aligned}((A \bmod C) + B) \bmod C &= (R_1 + B) \bmod C \\ &= (R_1 + C \cdot Q_2 + R_2) \bmod C \\ &= (C \cdot Q_2 + R_1 + R_2) \bmod C \\ &= (R_1 + R_2) \bmod C\end{aligned}$$

Let's calculate Right Hand Side:

$$\begin{aligned}(A+B) \bmod C &= (C \cdot Q_1 + R_1 + C \cdot Q_2 + R_2) \bmod C \\ &= (C \cdot Q_1 + C \cdot Q_2 + R_1 + R_2) \bmod C \\ &= (C(Q_1 + Q_2) + R_1 + R_2) \bmod C \\ &= (R_1 + R_2) \bmod C\end{aligned}$$

So, we show that both, Left Hand Side and Right Hand Side are equal to $(R_1 + R_2) \bmod C$. □

The following lemma establishes the validity of Rewrite $+^3_{rr}$ presented in Figure 3.

Lemma 4. $(A + (B \bmod C)) \bmod C = (A + B) \bmod C$

Proof. From the quotient remainder theorem, we can write A and B as:

$$A = C \cdot Q_1 + R_1$$

$$B = C \cdot Q_2 + R_2$$

where $0 \leq R_1 < C$, $0 \leq R_2 < C$ and Q_1, Q_2 are integers.
Let's calculate Left Hand Side:

$$\begin{aligned}(A + (B \bmod C)) \bmod C &= (A + R_2) \bmod C \\ &= (C \cdot Q_1 + R_1 + R_2) \bmod C \\ &= (R_1 + R_2) \bmod C\end{aligned}$$

Let's calculate Right Hand Side:

$$\begin{aligned}(A + B) \bmod C &= (C \cdot Q_1 + R_1 + C \cdot Q_2 + R_2) \bmod C \\ &= (C \cdot Q_1 + C \cdot Q_2 + R_1 + R_2) \bmod C \\ &= (C(Q_1 + Q_2) + R_1 + R_2) \bmod C \\ &= (R_1 + R_2) \bmod C\end{aligned}$$

So, we show that both, Left Hand Side and Right Hand Side are equal to $(R_1 + R_2) \bmod C$. □

The following lemma establishes the validity of Rewrite $-^1_{rr}$ presented in Figure 3. The validity of $-^2_{rr}$ and $-^3_{rr}$ is shown similarly.

Lemma 5. $((A \bmod C) - (B \bmod C)) \bmod C = (A - B) \bmod C$

Proof. From the quotient remainder theorem, we can write A and B as:

$$A = C \cdot Q_1 + R_1$$

$$B = C \cdot Q_2 + R_2$$

where $0 \leq R_1 < C$, $0 \leq R_2 < C$ and Q_1, Q_2 are integers.
Let's calculate Left Hand Side:

$$((A \bmod C) - (B \bmod C)) \bmod C = (R_1 - R_2) \bmod C$$

Let's calculate Right Hand Side:

$$\begin{aligned}(A - B) \bmod C &= (C \cdot Q_1 + R_1 - (C \cdot Q_2 + R_2)) \bmod C \\ &= (C \cdot Q_1 + R_1 - C \cdot Q_2 - R_2) \bmod C \\ &= (C \cdot Q_1 - C \cdot Q_2 + R_1 - R_2) \bmod C \\ &= (C(Q_1 - Q_2) + R_1 - R_2) \bmod C \\ &= (R_1 - R_2) \bmod C\end{aligned}$$

So, we show that both, Left Hand Side and Right Hand Side are equal to $(R_1 - R_2) \bmod C$. □

The following lemma establishes the validity of Rewrite $-^2_{rr}$ presented in Figure 3.

Lemma 6. $((A \bmod C) - B) \bmod C = (A - B) \bmod C$

Proof. From the quotient remainder theorem, we can write A and B as:

$$A = C \cdot Q_1 + R_1$$

$$B = C \cdot Q_2 + R_2$$

where $0 \leq R_1 < C$, $0 \leq R_2 < C$ and Q_1, Q_2 are integers.
Let's calculate Left Hand Side:

$$\begin{aligned}((A \bmod C) - B) \bmod C &= (R_1 - B) \bmod C \\ &= (R_1 - (C \cdot Q_2 + R_2)) \bmod C \\ &= (R_1 - C \cdot Q_2 - R_2) \bmod C \\ &= (R_1 - R_2) \bmod C\end{aligned}$$

Let's calculate Right Hand Side:

$$\begin{aligned}(A - B) \bmod C &= (C \cdot Q_1 + R_1 - (C \cdot Q_2 + R_2)) \bmod C \\ &= (C \cdot Q_1 + R_1 - C \cdot Q_2 - R_2) \bmod C \\ &= (C \cdot Q_1 - C \cdot Q_2 + R_1 - R_2) \bmod C \\ &= (C(Q_1 - Q_2) + R_1 - R_2) \bmod C \\ &= (R_1 - R_2) \bmod C\end{aligned}$$

So, we show that both, Left Hand Side and Right Hand Side are equal to $(R_1 - R_2) \bmod C$. □

The following lemma establishes the validity of Rewrite $-^3_{rr}$ presented in Figure 3.

Lemma 7. $(A - (B \bmod C)) \bmod C = (A - B) \bmod C$

Proof. From the quotient remainder theorem, we can write A and B as:

$$A = C \cdot Q_1 + R_1$$

$$B = C \cdot Q_2 + R_2$$

where $0 \leq R_1 < C$, $0 \leq R_2 < C$ and Q_1, Q_2 are integers.
Let's calculate Left Hand Side:

$$\begin{aligned}(A - (B \bmod C)) \bmod C &= (A - R_2) \bmod C \\ &= (C \cdot Q_1 + R_1 - R_2) \bmod C \\ &= (R_1 - R_2) \bmod C\end{aligned}$$

Let's calculate Right Hand Side:

$$\begin{aligned}(A - B) \bmod C &= (C \cdot Q_1 + R_1 - (C \cdot Q_2 + R_2)) \bmod C \\ &= (C \cdot Q_1 + R_1 - C \cdot Q_2 - R_2) \bmod C \\ &= (C \cdot Q_1 - C \cdot Q_2 + R_1 - R_2) \bmod C \\ &= (C(Q_1 - Q_2) + R_1 - R_2) \bmod C \\ &= (R_1 - R_2) \bmod C\end{aligned}$$

So, we show that both, Left Hand Side and Right Hand Side are equal to $(R_1 - R_2) \bmod C$. □

The following lemma establishes the validity of Rewrite $\cdot^1_{rr}$ presented in Figure 3. The validity of $\cdot^2_{rr}$ and $\cdot^3_{rr}$ is shown similarly.

Lemma 8. $((A \bmod C) \cdot (B \bmod C)) \bmod C = (A \cdot B) \bmod C$

Proof. From the quotient remainder theorem, we can write A and B as:

$$A = C \cdot Q_1 + R_1$$

$$B = C \cdot Q_2 + R_2$$

where $0 \leq R_1 < C$, $0 \leq R_2 < C$ and Q_1, Q_2 are integers.
Let's calculate Left Hand Side:

$$((A \bmod C) \cdot (B \bmod C)) \bmod C = (R_1 \cdot R_2) \bmod C$$

Let's calculate Right Hand Side:

$$\begin{aligned}(A \cdot B) \bmod C &= ((C \cdot Q_1 + R_1) \cdot (C \cdot Q_2 + R_2)) \bmod C \\ &= (C \cdot Q_1 \cdot C \cdot Q_2 + C \cdot Q_1 \cdot R_2 + C \cdot Q_2 \cdot R_1 + R_1 \cdot R_2) \bmod C \\ &= (C(Q_1 \cdot C \cdot Q_2 + Q_1 \cdot R_2 + Q_2 \cdot R_1) + R_1 \cdot R_2) \bmod C \\ &= (R_1 \cdot R_2) \bmod C\end{aligned}$$

So, we show that both, Left Hand Side and Right Hand Side are equal to $(R_1 \cdot R_2) \bmod C$. □

The following lemma establishes the validity of Rewrite $\cdot^2_{rr}$ presented in Figure 3.

Lemma 9. $((A \bmod C) \cdot B) \bmod C = (A \cdot B) \bmod C$

Proof. From the quotient remainder theorem, we can write A and B as:

$$A = C \cdot Q_1 + R_1$$

$$B = C \cdot Q_2 + R_2$$

where $0 \leq R_1 < C$, $0 \leq R_2 < C$ and Q_1, Q_2 are integers.
Let's calculate Left Hand Side:

$$\begin{aligned}((A \bmod C) \cdot B) \bmod C &= (R_1 \cdot B) \bmod C \\ &= (R_1 \cdot (C \cdot Q_2 + R_2)) \bmod C \\ &= (C \cdot Q_2 \cdot R_1 + R_1 \cdot R_2)) \bmod C \\ &= (R_1 \cdot R_2)) \bmod C\end{aligned}$$

Let's calculate Right Hand Side:

$$\begin{aligned}(A \cdot B) \bmod C &= ((C \cdot Q_1 + R_1) \cdot (C \cdot Q_2 + R_2)) \bmod C \\ &= (C \cdot Q_1 \cdot C \cdot Q_2 + C \cdot Q_1 \cdot R_2 + C \cdot Q_2 \cdot R_1 + R_1 \cdot R_2) \bmod C \\ &= (C(Q_1 \cdot C \cdot Q_2 + Q_1 \cdot R_2 + Q_2 \cdot R_1) + R_1 \cdot R_2) \bmod C \\ &= (R_1 \cdot R_2)) \bmod C\end{aligned}$$

So, we show that both, Left Hand Side and Right Hand Side are equal $(R_1 \cdot R_2) \bmod C$. □

The following lemma establishes the validity of Rewrite $\cdot^3_{rr}$ presented in Figure 3.

Lemma 10. $(A \cdot (B \bmod C)) \bmod C = (A \cdot B) \bmod C$

Proof. From the quotient remainder theorem, we can write A and B as:

$$A = C \cdot Q_1 + R_1$$

$$B = C \cdot Q_2 + R_2$$

where $0 \leq R_1 < C$, $0 \leq R_2 < C$ and Q_1, Q_2 are integers.
Let's calculate Left Hand Side:

$$\begin{aligned}(A \cdot (B \bmod C)) \bmod C &= (A \cdot R_2) \bmod C \\ &= ((C \cdot Q_1 + R_1) \cdot R_2) \bmod C \\ &= (C \cdot Q_1 \cdot R_2 + R_1 \cdot R_2) \bmod C \\ &= (R_1 \cdot R_2) \bmod C\end{aligned}$$

Let's calculate Right Hand Side:

$$\begin{aligned}(A \cdot B) \bmod C &= ((C \cdot Q_1 + R_1) \cdot (C \cdot Q_2 + R_2)) \bmod C \\ &= (C \cdot Q_1 \cdot C \cdot Q_2 + C \cdot Q_1 \cdot R_2 + C \cdot Q_2 \cdot R_1 + R_1 \cdot R_2) \bmod C \\ &= (C(Q_1 \cdot C \cdot Q_2 + Q_1 \cdot R_2 + Q_2 \cdot R_1) + R_1 \cdot R_2) \bmod C \\ &= (R_1 \cdot R_2) \bmod C\end{aligned}$$

So, we show that both, Left Hand Side and Right Hand Side are equal to $(R_1 \cdot R_2) \bmod C$. □

The following lemma establishes the validity of Rewrite i^1_{rr} presented in Figure 3. The proofs for i^2_{rr} and i^3_{rr} are similar.

Lemma 11. $(ite(D, A \bmod C, B \bmod C) \bmod C = ite(D, A, B) \bmod C$.

Proof. Assume w.l.g. that D holds. Then the left-hand side equals $(A \bmod C) \bmod C)$ and the right-hand side equals $A \bmod C$, which are equal, as $0 \leq A \bmod C < C$. □

The following lemma establishes the validity of Rewrite i^2_{rr} presented in Figure 3.

Lemma 12. $(ite(D, A, B \bmod C) \bmod C = ite(D, A, B) \bmod C$.

Proof. If D holds then this is trvial. Otherwise, Then the left-hand side equals $(B \bmod C) \bmod C)$ and the right-hand side equals $B \bmod C$, which are equal, as $0 \leq B \bmod C < C$. □

The following lemma establishes the validity of Rewrite i^3_{rr} presented in Figure 3.

Lemma 13. $(ite(D, A \bmod C, B) \bmod C = ite(D, A, B) \bmod C$.

Proof. If D does not hold then this is trvial. Otherwise, Then the left-hand side equals $(A \bmod C) \bmod C)$ and the right-hand side equals $A \bmod C$, which are equal, as $0 \leq A \bmod C < C$. □

4.5.4 Correctness of Lemmas

The following lemma proves the validity of bound.

Lemma 14. *Suppose* $v \geq 7$. *Then:* $x \geq v \Rightarrow v \cdot x + v^2 < \mathsf{pow}_2(x)$

Proof. We first prove that $x \geq 7 \Rightarrow 2 \cdot x^2 < 2^x$. We do so by induction on x. If $x = 7$ then we get $98 < 128$ which holds. Now 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 $2 \cdot (x+1)^2 < 2^x + 2^x = 2^{x+1}$.

Now, let $v \geq 7$ and suppose $x \geq v$. Then, $v \cdot x \leq x^2$ and $v^2 \leq x^2$, and so $v \cdot x + v^2 \leq 2 \cdot x^2 < 2^x$. □

The following lemma proves the validity of $\mathrm{sum}_{\geq}$.

Lemma 15. *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 for every such i we also have $\mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i = 0$. □

4.6 Type Checker

Recall that the function $\mathcal{T}(\varphi)$ from algorithm 3 returns a conjunction of three formulas: $\mathrm{C}(\varphi)$, which is the translation of the input formula to integers, $\textsc{Range}(\varphi)$, which adds range constraints in order to encode the bit-widths into integer ranges, and $\textsc{Type}(\varphi)$, that enforces restrictions on the bit-widths themselves. As it turns out, the last conjunct, namely $\textsc{Type}(\varphi)$ is useful in its own right. Indeed, our framework allows users to textually describe formulas over parametric bit-vectors with unfeasible bit-widths. Figure 4 contains an example for such a case. The formula declares a single integer variable k and a parametric bit-vector variable x of bit-width k. It then asserts that x is equal to $x \circ x$. However, because $x \circ x$ has bit-width $2k$ while x has bit-width k, the comparison is type-inconsistent.

```
(set-logic ALL)
(declare-const k Int)
(declare-const x (_ BitVec k))
(assert (= (concat x x) x))
```

Figure 4: An example of a formula with invalid bit-widths due to symbolic parameter `k`.

By default, running our solver on such cases yields an unsat result. This makes sense, since indeed, the described constraint is admissibly-unsatisfiable. But, a more informative output can be given to the user, namely that unsatisfiability does not necessarily arise from the formula itself, but from its associated bit-widths, or, from the types of the bit-vectors. We therefore implemented an optional light-weight type checker, to check precisely these cases. The behavior of using this optional flag is described in Algorithm 6. When the type checker is turned on, only $\textsc{Type}(\varphi)$ is first checked for satisfiability. Only if it is satisfiable, then the full translation is checked. If $\textsc{Type}(\varphi)$ is unsatisfiable, we can then report a type error to the user. This approach follows the type constraints from [8]. While in that paper, the canonical example was identifying possible division by zeros in integer and real arithmetic, here we use the same approach for constructing formulas with parametric bit-vectors of incompatible bit-widths.

Algorithm 6 The optional type checking flag.

function TYPE CHECKER(φ):
  $\phi \leftarrow \textsc{Type}(\varphi)$
  if ϕ is not T_{NIA}-satisfiable **then**
    Throw Exception "Type Checker Error."
  run $\textsc{Solve}_\star(PBV)\varphi$ from Algorithm 5

5 Evaluation

We have implemented our approach by extending two tools: the generic SMT solver API smt-switch [24] and the SMT solver cvc5 [1]. In smt-switch, we extended the SMT-LIB parser to support parsing Σ_{PBV}-formulas, implemented our bit-vector rewriter $\mathcal{RW}_\mathcal{B}$ (Section 4.1), the translation function $\mathcal{T}$ (Algorithm 3), and the arithmetic rewriter optimization $\mathcal{RW}_\mathcal{A}$ (Section 4.3). 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 [40], 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 [3, 5] 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 benchmarks 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 [23]. The evaluation of [32] included 180 such conditions. Here, we include 20 additional conditions that were not supported by the implementation of [32] (17 of the new conditions involve multiple bit-widths and 3 include terms of the form $topbv(k, k)$).

ic (180 benchmarks). Set *ic* consists of benchmarks to verify the correctness of the *invertibility conditions* formalized in Niemetz et al. [29], which utilizes these conditions for quantifier instantiation. They are also instrumental to a local search procedure for fixed-size bit-vectors [29]. The evaluation of [32] included 160 (out of 180) correctness checks of invertibility conditions. Here, we also include the remaining 20 checks that encode invertibility conditions over the $\circ$ operator, which were not supported by the implementation of [32].

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 [32]. They were automatically generated using CVC4SY [38], 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 [33], 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 [33], 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 [28]. These lemmas were verified to be correct up to bit-width 256 [33] 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 [32].

icfb (46 benchmarks). Set *icfb* originates from a local search procedure [27] that generalizes the technique from [29] 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 ze-

roes, and where thus excluded from this set. Further note that in the original publication [27], one of the invertibility conditions for operator $<_{\mathrm{u}}$ was given incorrectly (the implementation and verification of the rule used the correct definition).[1] 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 of our approach on satisfiable benchmarks, we created a benchmark set *mut* by mutating benchmarks from the other sets. Our goal is to mimic realistic cases of introducing bugs in verification conditions, e.g., by using a bitwise disjunction instead of conjunction or a right shift instead of a left shift. We generated this set as follows. Let (f, g) be a pair of Σ_{PBV}-terms with the same arity, and let φ be a Σ_{PBV}-formula that contains at least one occurrence of f. Given a pair (f, g), a mutation of φ is obtained by replacing the first occurrence of f in φ by g. We generated mutations for pairs $(f, g) \in \{(\&, |), (+^{\mathcal{B}}, -^{\mathcal{B}}), (-^{\mathcal{B}}, \sim), (\ll, \gg), (\mathit{topbv}(k, 1), \mathit{topbv}(k, 0)), (\mathit{topbv}(k, k), \mathit{topbv}(k, 0))\}$ and its permutation (g, f) for each benchmark. In most cases, the resulting formulas are satisfiable. Note that we include unsatisfiable mutations but filtered out duplicate benchmarks.

Attribute	BASELINE	EAGER	PBV
Target Theory	$T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}}, \|^{\mathbb{N}}, \oplus^{\mathbb{N}})$	$T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$	$T_{IA}(\mathsf{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$
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	✗	✓	✓
$\mathcal{RW}_{\mathcal{B}}$	✗	✓	✓
$\mathcal{RW}_{\mathcal{A}}$	✗	✓	✓

Table 7: Configurations considered in our experimental setup.

5.2 Primary Experimental Setup

For our experimental evaluation, we consider three configurations: configuration BASELINE, which corresponds to the best configuration of [32]; configuration PBV, which implements our techniques as described in Section 4; and configuration EAGER, which corresponds to extending configuration

[1] The correction is given in `https://bitwuzla.github.io/data/fmcad2020/errata.txt`.

Baseline with our rewriter $\mathcal{RW}_{\mathcal{B}}$ and $\mathcal{RW}_{\mathcal{A}}$ as detailed in Table 7 and Figure 3 , while still handling operators pow_2 and $\&^{\mathbb{N}}$ eagerly. Recall that the implementation of [32] utilized translations that were tailored to each considered benchmark set each benchmark set, which does not allow a faithful comparison to our Pbv-configurations. Thus, we reimplemented the best configuration *comb* from [32] 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 7 outlines the differences of each configuration of the primary experimental evaluation. Configurations Baseline and Eager rely on an *eager* translation to target theories $T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}}, |^{\mathbb{N}}, \oplus^{\mathbb{N}})$ and $T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$. The introduced uninterpreted functions are axiomatized by means of quantified formulas that are added as preamble to each translation. This happens regardless of whether the operator actually occurs in the formula for configuration Baseline, thus potentially adding redundant axioms. In contrast, configuration Pbv employs a translation to theory $T_{IA}(\mathsf{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$, 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 described in section Section 4.4, that are also incorporated in Pbv. Notice that operators $|$ and $\oplus$ are handled natively in configuration Baseline, while configurations Eager and Pbv use the elimination rules described in Section 4.2.

Remark 3. *Even though our evaluation includes benchmark sets from the evaluation in [32], the numbers are not directly comparable due to the following reasons. Each benchmark in set ic describes an equivalence, which was split into two benchmarks in [32], one for each direction of the equivalence. Additionally, the notion of* conditional inverses *was introduced, which 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 for parametric bit-vectors is in a fully automated setting. Further, the translation of [32] utilized quantifier-specific optimizations and included instantiation patterns as well as concrete instantiations for some of the axioms. However, quantifier instantiation patterns are often tailored to a specific solving procedure, and concrete instantiations are only sound for unsatisfiable benchmarks (which was the focus of [32]). In configuration Baseline, we therefore do not use any of these optimizations.*

We ran our experiments on a cluster of 22 machines equipped with Intel

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-solved		709k	634k	**375k**	355k
time-common	1890	13152	**229**	335	
mem-solved		49151	25972	**25749**	31569
mem-common	1890	55352	4977	**4860**	

Table 8: 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.

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 enough to justify the additional resource usage overhead. Further, no configuration exceeded the memory limit of 8GB during solving.

5.3 Primary Results

Table 8 summarizes the results for each benchmark set described in Section 5.1. In addition to the results of the configurations introduced in Section 5.2, we also report the results for the virtual best solver over these configurations in column VBS.

Overall, configuration Pbv significantly outperforms configurations Baseline and Eager in number of solved instances and runtime. Further, comparing Pbv against Eager indicates that using a dedicated procedure for 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 also beneficial for the eager translation. On the 1890s instances commonly solved by all three configurations, Eager is almost 4× faster than Baseline while Pbv is more 12× faster than Eager.

Benchmarks	#	Baseline	Eager	Pbv	VBS
alive	17	0	9	10	10
ic	20	0	5	8	8
rewrite	0	0	0	0	0
syrew	0	0	0	0	0
lemmas	2	0	0	0	0
icfb	16	0	8	10	10
mut	316	0	88	150	151
total	371	0	110	178	179
sat		0	0	20	20
unsat		0	110	158	159
time-solved		0	277	**30**	26
time-common	0	0	0	0	0
mem-solved		0	796	**666**	659
mem-common	0	0	0	0	0

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

On the 3060 instances commonly solved by Eager and Pbv, configuration Pbv is more than 18× faster. Notice that configurations Baseline and Eager do not solve any satisfiable benchmarks. This is due to the quantified axiomatization of the uninterpreted functions introduced in the target theory. In fact, the presence of these axiomatizations already prevents both Baseline and Eager from reporting that the formula $\top$ is satisfiable.

One of the capabilities that Baseline does not offer is support for benchmarks that involve multiple bit-widths. Our benchmark sets include in total 371 of such benchmarks. The performance outcomes for this benchmark set are summarized in Table 9. 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 configurations Eager and Pbv but has a bigger impact on Eager. Without rewriting, Eager solves 474 less instances, while Pbv only loses 72 instances.

Proving the correctness of the invertibility conditions from [30] independent of the bit-width (benchmark set *ic*) has previously been attempted with two different approaches: *(i)* the eager T_{PBV} approach described in [32] and *(ii)* a bit-width independent formalization in Coq [14]. Out of the 81 invertibility conditions proven across our three configurations, 12 conditions were proven correct that approaches *(i)* and *(ii)* were not able to prove. Of these

12 instances, 9 involve multiple bit-widths which both *(i)* and *(ii)* cannot handle (8 solved by PBV and 1 by PBV$^{\mathcal{R}-}$ that will be described in Table 13). The remaining 3 conditions involve operators that are not supported by *(ii)* (signed inequalities, multiplication and division). The new status for the verification of invertability conditions is shown in Table 10.

$\ell[x]$	$\approx$	$\not\approx$	$<_u$	$>_u$	$\leq_u$	$\geq_u$	$<_s$	$>_s$	$\leq_s$	$\geq_s$
$-^{\mathcal{B}}x \bowtie t$	✓✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓	✓	✓✓	✓
$\sim x \bowtie t$	✓✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓	✓	✓✓	✓✓
$x \,\&\, s \bowtie t$	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✗	✗	✗	✗
$x \mid s \bowtie t$	✓	✓✓	✓✓	✓	✓✓	✓✓	✗	✗	✗	✗
$x \ll s \bowtie t$	✓	✓	✓✓	✓	✓✓	✓	✗	✗	✗	✗
$s \ll x \bowtie t$	✓✓	✓	✓	✓	✓✓	✓	✗	✓	✗	✓
$x \gg s \bowtie t$	✓✓	✓	✓✓	✓	✓✓	✓✓	✓	✗	✓	✗
$s \gg x \bowtie t$	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓	✓	✓	✓
$x \gg_a s \bowtie t$	✓	✓	✓✓	✓✓	✓✓	✓	✗	✓	✗	✓
$s \gg_a x \bowtie t$	✓✓	✓	✓	✓	✓✓	✓	✗	✗	✗	✓
$x +^{\mathcal{B}} s \bowtie t$	✓✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓	✓	✓✓	✓✓
$x \cdot^{\mathcal{B}} s \bowtie t$	✗	✓	✓✓	✗	✓✓	✗	✗	✗	✗	✗
$x \mathrm{div}^{\mathcal{B}} s \bowtie t$	✓	✓✓	✓✓	✓✓	✓✓	✗	✓	✓	✓	✓
$s \mathrm{div}^{\mathcal{B}} x \bowtie t$	✓	✓	✓✓	✓✓	✓✓	✓✓	✓	✓	✓✓	✗
$x \mathrm{mod}^{\mathcal{B}} s \bowtie t$	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✗	✓	✗	✓
$s \mathrm{mod}^{\mathcal{B}} x \bowtie t$	✗	✓✓	✓✓	✓✓	✓✓	✓	✓	✗	✓	✗
$x \circ s \bowtie t$	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗
$s \circ x \bowtie t$	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗

Table 10: Invertibility conditions verification. ✓ means the condition was fully proved by [32], ✓ indicates that the condition was fully proved by at least one of the *PBV* configurations, ✓ means that the condition was fully proved by coq, ✗ means that it was not proved.

Proving the correctness of the compiler optimizations from [23] independent of the bit-width (benchmark set *alive*) has previously been attempted with the eager T_{PBV} approach described in [32]. Out of the 124 optimizations proven across our three configurations, 44 optimizations were proven correct that the previous approach was not able to prove. Of these 44 instances, 10 involve multiple bit-widths which previous approach cannot handle. The performance outcomes for alive benchmark set are shown in Table 11.

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 up to a fixed bit-width [27, 33]. Configuration PBV was able to confirm that the incorrect condition benchmark from *icfb* — presented in [27] and described in Section 5.1 — is indeed incorrect. However, it was

Family	Considered	Proved			
		[32]	Eager	Pbv	VBS
AddSub (52)	19	9	8	12	16
MulDivRem (29)	13	3	8	7	10
AndOrXor (162)	130	60	61	77	88
Select (51)	27	16	13	8	16
Shifts (17)	11	0	3	3	3
LoadStoreAlloca (9)	0	0	0	0	0
Total (320)	200	88	93	107	133

Table 11: Alive optimizations verification using Baseline, Eager and Pbv.

Attribute	OR-E	XOR-E	SH-M-E	ALL-E	POW2++	PIAND++
Target Theory	T_1					
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	✗	✗	✗	✗	✗	✗
$\mathcal{RW}_{\mathcal{B}}$	✗	✗	✗	✗	✗	✗
$\mathcal{RW}_{\mathcal{A}}$	✗	✗	✗	✗	✗	✗

Table 12: Intermediate configurations with $T_1 = T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}}, |^{\mathbb{N}}, \oplus^{\mathbb{N}})$.

unable to prove the corrected condition within the given time limit.

5.4 More Detailed Results

To better understand the improvements in our solver, we introduced additional configurations that we have evaluated. The configurations are listed in Tables 12 and 13.

Table 12 presents various changes to Baseline, each designed to evaluate the impact of individual optimization components on solver performance.

We begin by assessing the effect of our elimination techniques introduced in the translation process. Specifically, we investigate three elimination strategies: (1) the elimination of bitwise-OR operations by translating them to integer-AND, (2) the elimination of bitwise-XOR operations via integer-AND, and (3) the elimination of modulo operations in left bitwise shift expressions. These techniques are detailed in Section 4.2. To evaluate

Attribute	POW2-L	PIAND-L	EAGER$^{\not\mathcal{RW}}$	EAGER$^{\mathcal{RW}}$	EAGER^{-}	PBV$^{\mathcal{RW}}$	PBV$^{\not\mathcal{RW}}$	PBV^{-}
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 $\&^{\mathbb{N}}$	✗	✓	✗	✗	✗	✓	✓	✓
|-elimination	✗	✗	✓	✓	✓	✓	✓	✓
⊕-elimination	✗	✗	✓	✓	✓	✓	✓	✓
≫ without mod	✗	✗	✓	✓	✓	✓	✓	✓
New lemmas for $\&^{\mathbb{N}}$	✗	✓	✓	✓	✓	✓	✓	✓
New lemmas for pow_2	✓	✗	✓	✓	✓	✓	✓	✓
No redundant axioms	✗	✗	✓	✓	✓	✓	✓	✓
$\mathcal{RW}_{\mathcal{B}}$	✗	✗	✗	✗	✓	✗	✗	✓
$\mathcal{RW}_{\mathcal{A}}$	✗	✗	✗	✓	✗	✓	✗	✗

Table 13: Intermediate configurations. $T_3 = T_{IA}(\mathsf{pow}_{2\star}, \&^{\mathbb{N}})$, that is, pow_2 has a fixed interpretation, while $\&^{\mathbb{N}}$ is freely interpreted. $T_4 = T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}}_{\star})$, that is, $\&^{\mathbb{N}}$ has a fixed interpretation, while pow_2 is freely interpreted. $T_5 = T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$. $T_2 = T_{IA}(\mathsf{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$.

them individually, we introduce configurations OR-E, XOR-E, and SH-M-E, corresponding to the eliminations of bitwise-OR, bitwise-XOR, and modulo in left shift, respectively. In addition, we define ALL-E, which applies all three eliminations simultaneously.

We also investigate the contribution of new lemmas added to the solver in the eager setting. These include lemmas for the functions pow_2 and $\&^{\mathbb{N}}$, as described in Section 4.4. Configurations POW2++ and PIAND++ are introduced to isolate the effects of the new lemmas for pow_2 and $\&^{\mathbb{N}}$, respectively, allowing us to evaluate each in isolation.

Table 13 presents a range of configurations for evaluating the lazy approach and the impact of the rewriters $\mathcal{RW}_{\mathcal{A}}$ and $\mathcal{RW}_{\mathcal{B}}$ in both lazy and eager settings. We begin by introducing POW2-L and PIAND-L, which are based on BASELINE but apply lazy reasoning to pow_2 and $\&^{\mathbb{N}}$, respectively. Specifically, POW2-L operates in the $T_{IA}(\mathsf{pow}_{2\star}, \&^{\mathbb{N}})$ theory, where pow_2 is interpreted and $\&^{\mathbb{N}}$ is left uninterpreted. Conversely, PIAND-L is defined over the $T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}}_{\star})$ theory, where $\&^{\mathbb{N}}$ is interpreted and pow_2 remains uninterpreted.

To assess the effectiveness of $\mathcal{RW}_{\mathcal{A}}$ and $\mathcal{RW}_{\mathcal{B}}$ in the lazy setting, we introduce a series of configurations based on PBV, which includes both rewriters. We define PBV^{-} as the variant with $\mathcal{RW}_{\mathcal{B}}$ only (i.e., without $\mathcal{RW}_{\mathcal{A}}$), PBV$^{\mathcal{RW}}$ as the variant with $\mathcal{RW}_{\mathcal{A}}$ only (i.e., without $\mathcal{RW}_{\mathcal{B}}$), and PBV$^{\not\mathcal{RW}}$ without either rewriter. All of these configurations operate under the $T_{IA}(\mathsf{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$ theory, in which both pow_2 and $\&^{\mathbb{N}}$ are interpreted.

Analogously, we introduce eager counterparts — EAGER^{-}, EAGER$^{\mathcal{RW}}$, and EAGER$^{\not\mathcal{RW}}$ —that interpret both pow_2 and $\&^{\mathbb{N}}$ eagerly and follow the same rewriter inclusion pattern. These configurations use the $T_{IA}(\mathsf{pow}_2, \&^{\mathbb{N}})$ theory.

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
time-solved		15771	14863	**13154**	13181	15680	13947	13955
time-common	1838	10018	10217	5875	**5812**	10260	10724	
mem-solved		57550	54182	51684	52451	**50870**	51346	58537
mem-common	1838	41920	42143	29940	**29364**	38278	43538	

Table 14: Overall number of solved benchmarks for each configuration. VBS corresponds to the virtual best solver over all three configurations.

We present the evaluation results for the extended set of configurations. The results are in Tables 14 and 15. VBS corresponds to the virtual best solver over all configurations in the table.

Table 14 presents the evaluation results for the various configurations described in Table 12. Overall, the configuration SH-M-E yields the most significant improvement, whereas OR-E and XOR-E have only a minor impact. Combining all eliminations in ALL-E shows a slight improvement. The configurations POW2++ and PIAND++, which introduce new lemmas in the eager approach, also have limited impact and, in some cases, perform worse than BASELINE. In terms of runtime, SH-M-E substantially reduces execution time solving common benchmarks in nearly half the time while ALL-E provides a modest improvement. Regarding memory usage, SH-M-E shows the most significant reduction across all configurations, with ALL-E again offering a small benefit. These configurations have no effect on the benchmarks in *alive* and *icfb*. Furthermore, since all configurations are eager, they are unable to solve satisfiable benchmarks.

Table 15 presents the evaluation results for the various configurations described in Table 13. Overall, all configurations demonstrate significant improvement. The best results are achieved by the configurations that use the lazy approach for both pow_2 and &. These configurations are also the only ones that successfully solve satisfiable benchmarks. In contrast, the eager configurations and those that apply the lazy approach to only one of the operations fail to solve any satisfiable benchmarks.

In particular, POW2-L solves 12 additional benchmarks in the *ic* set, 5 new lemmas from the *lemmas* set, and more than 60 benchmarks from the

Benchmarks	#	POW2-L	PIAND-L	EAGER$^{\mathcal{R}\text{-}}$	EAGER$^{\mathcal{R}}$	EAGER$^{\text{-}}$	PBV$^{\mathcal{R}}$	PBV$^{\mathcal{R}\text{-}}$	PBV$^{\text{-}}$	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
time-solved		**1366**	15765	14829	12738	10352	9805	3512	3511	5161
time-common	1609	386	8636	4855	3547	2485	2107	**190**	249	
mem-solved		**12391**	58938	48953	44266	36934	36417	26244	25108	32677
mem-common	1609	6019	34114	21534	17857	14341	13779	4195	**3944**	

Table 15: Overall number of solved benchmarks for each configuration. VBS corresponds to the virtual best solver over all three configurations.

mut set, compared to BASELINE. Likewise, PIAND-L solves over 100 additional benchmarks in *syrew* and more than 30 in *rewrite*. All configurations based on PBV, even when either $\mathcal{RW}_{\mathcal{B}}$ or $\mathcal{RW}_{\mathcal{A}}$ is disabled, show significant improvement across all benchmark sets. The table indicates that each of these rewriters contributes meaningfully to the solver's performance.

In terms of runtime, POW2-L greatly reduces the solver runtime and solves the common benchmarks up to ten times faster than the eager approach. The best lazy configuration improves even further, cutting runtime by half compared to POW2-L. Regarding memory usage, POW2-L already shows a substantial improvement, and the full lazy approach improves memory efficiency even further.

Additionally, EAGER$^{\mathcal{R}\text{-}}$, which is BASELINE augmented with all our improvements except $\mathcal{RW}_{\mathcal{A}}$ and $\mathcal{RW}_{\mathcal{B}}$ in the eager approach, demonstrates a notable improvement over BASELINE. This indicates that our combined enhancements can still be beneficial even in the eager approach.

Finally, we evaluate the impact of $\mathcal{RW}_{\mathcal{A}}$ and $\mathcal{RW}_{\mathcal{B}}$ in both the lazy and eager approaches. This comparison is summarized in Figure 5. Figure 5 demonstrating that in both the eager and lazy settings, the inclusion of $\mathcal{RW}_{\mathcal{A}}$ and $\mathcal{RW}_{\mathcal{B}}$ leads to improved solver performance over the baseline configuration without these rewriters.

Additionally, all configurations of the lazy approach perform better than all configurations of the eager approach. In fact, even the lazy configuration without any additional rewrites outperforms the eager configuration that includes all available rewrites. The difference between the two approaches is significant, with the lazy configuration solving nearly five times more benchmarks than the eager one.

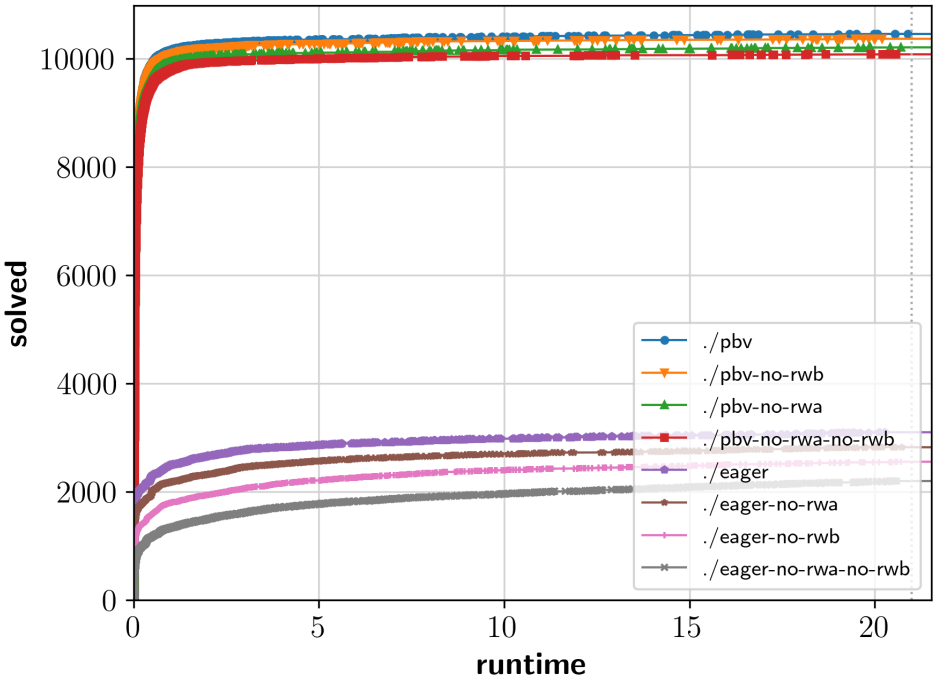

Figure 5: Eager and Pbv verification. no-rwa means the solver does not use $\mathcal{RW}_{\mathcal{A}}$, no-rwb means the solver does not use $\mathcal{RW}_{\mathcal{B}}$.

6 Conclusion and Further Research

We have presented a new formalization of a theory of parametric bit-vectors T_{PBV} and proposed a dedicated, lazy procedure for solving T_{PBV}-formulas. Our techniques significantly improve over existing approaches for parametric bit-vectors over a wide range of benchmarks.

In future work, we plan to explore proof production for parametric bit-vectors, building on the algorithms presented in this thesis. This capability will enable the integration of our solver into widely used proof assistants such as Coq [13], Lean [25] and Isabelle [34]. This will improve their automation.

7 Bibliography

[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 SMT-LIB Standard: Version 2.7. Technical report, Department of Computer Science, The University of Iowa, 2025. Available at `www.SMT-LIB.org`.

[3] 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.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2023.

[6] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021. `doi:10.3233/FAIA201017`.

[7] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.

[8] Sergey Berezin, Clark W. Barrett, Igor Shikanian, Marsha Chechik, Arie Gurfinkel, and David L. Dill. A practical approach to partial functions in CVC lite. In *D/PDPAR@IJCAR*, volume 125 of *Electronic Notes in Theoretical Computer Science*, pages 13–23. Elsevier, 2004.

[9] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

[10] 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.

[11] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[12] 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.

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

[14] 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.

[15] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

תקציר

התיאוריה של ביט-וקטורים ב-SMT-LIB היא לביט-וקטורים בעלי אורך קבוע. עם זאת, מספר יישומים חשובים יכולים להפיק תועלת מהוכחה על ביט-וקטורים ברוחב סימבולי, כלומר ביט-וקטור בעל אורך משתנה. עבודה אחרונה הציגה גישה לפתרון נוסחאות לביט-וקטורים באמצעות תרגום לאריתמטיקה של מספרים שלמים עם כמתים ופונקציות לא מפורשות. הגישה הוכחה כמוצלחת עבור מספר יישומים, כולל אימות של תנאי היפוך, אופטימיזציות של מהדר ונוסחאות משוכתבות מפותרנים אוטומטים. במאמר זה אנו מרחיבים ומשפרים את הגישה הקיימת במספר היבטים. תיאורטית, אנו משפרים את יכולת ההבעה על ידי הגדרת תיאוריה חדשה של ביט וקטורים פרמטריים. מבחינה אלגוריתמיתף אנו מציגים אלגוריתם עצלן שנמנע מכמתים ופונקציות לא מפורשות. מבחינה אפירית, אנו מראים שיפור משמעותי על ידי יישום והערכה של גישתנו והשוואתה לגישה הקודמת.

עבודה זו נעשתה בהדרכתו של ד"ר יוני זוהר מן המחלקה למדעי המחשב של אוניברסיטת בר אילן.

אוניברסיטת בר אילן

הסקה בביט-וקטורים בגודל פרמטרי

צבי ברגר

עבודה זו מוגשת כחלק מהדרישות לשם קבלת תואר מוסמך במחלקה למדעי המחשב
של אוניברסיטת בר אילן

רמת גן תשפ"ה