Scalable Bit-Blasting with Abstractions

Aina Niemetz[1(✉)], Mathias Preiner[1], and Yoni Zohar[2]

[1] Stanford University, Stanford, USA
niemetz@cs.stanford.edu
[2] Bar-Ilan University, Ramat Gan, Israel

Abstract. The dominant state-of-the-art approach for solving bit-vector formulas in Satisfiability Modulo Theories (SMT) is bit-blasting, an eager reduction to propositional logic. Bit-blasting is surprisingly efficient in practice but does not generally scale well with increasing bit-widths, especially when bit-vector arithmetic is present. In this paper, we present a novel CEGAR-style abstraction-refinement procedure for the theory of fixed-size bit-vectors that significantly improves the scalability of bit-blasting. We provide lemma schemes for various arithmetic bit-vector operators and an abduction-based framework for synthesizing refinement lemmas. We extended the state-of-the-art SMT solver Bitwuzla with our abstraction-refinement approach and show that it significantly improves solver performance on a variety of benchmark sets, including industrial benchmarks that arise from smart contract verification.

1 Introduction

Bit-precise reasoning as provided by Satisfiability Modulo Theories (SMT) for the theory of fixed-size bit-vectors is a key requirement for many applications in computer-aided verification. The dominant, state-of-the-art approach for solving bit-vector formulas is a technique called *bit-blasting* [24], an eager reduction of bit-vector constraints to a propositional satisfiability problem (SAT). Bit-blasting is usually combined with aggressive simplifications of the input constraints prior to the actual reduction step. Even though this eager reduction may come at the cost of significantly increasing the formula size, it is surprisingly efficient in practice—mainly due to the fact that state-of-the-art SAT solvers are usually able to efficiently deal with complex formulas over millions of variables. This size increase, however, is a potential bottleneck and the main reason why bit-blasting does not generally scale well for large bit-widths. This is especially true in the presence of arithmetic operators, which translate to large and complex Boolean circuits on the bit-level. In practice, this scaling issue can already

This work was supported in part by the Stanford Center for Automated Reasoning, the Stanford Center for Blockchain Research, ISF grant number 619/21, and a gift from Amazon Web Services.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14681, pp. 178–200, 2024.
https://doi.org/10.1007/978-3-031-65627-9_9

occur with bit-widths as low as 32 bits, and it is especially severe for applications that reason over considerably larger bit-widths due to the nature of their domain, e.g., 256 bits in the context of smart contract verification [15].

In this paper, we propose a novel abstraction-refinement framework for the theory of fixed-size bit-vectors that significantly improves the scalability of bit-blasting on increasing bit-widths. Rather than providing an alternative to bit-blasting, our approach is explicitly aimed at improving its performance via an abstraction-refinement scheme based on the counterexample-guided abstraction refinement (CEGAR) paradigm [16]. Constructs and operators that are potentially expensive when translated to the bit-level are abstracted with fresh uninterpreted functions (UF), which corresponds to over-approximating the original problem and translates to significantly smaller circuits on the bit-level. When an abstraction is unsatisfiable, so is the original problem. However, when it is satisfiable and inconsistent with the true semantics of the abstracted operators, it must be refined with lemmas to rule out spurious counterexamples. We iteratively repeat the abstraction-refinement process until all abstractions are consistent, and only fall back to bit-blasting an abstracted term when it cannot be further refined, as a last resort. Thus, the main challenge is finding lemmas for abstraction refinement that, ideally, allow to avoid bit-blasting of abstracted terms, entirely. To this extent, this paper makes the following *contributions*:

- We present a modular and configurable CEGAR-style abstraction-refinement framework for the theory of fixed-size bit-vectors, based on bit-blasting.
- We provide a set of refinement lemmas for a restricted but sufficient set of arithmetic bit-vector operators (`bvmul`, `bvudiv`, `bvurem`). This set of lemmas consists of a set of basic, *hand-crafted* lemmas (encoding core properties of abstracted operators) and a set of lemmas synthesized via *abduction*.
- We provide a lemma scoring scheme and an abduction-based framework for synthesizing lemmas, utilizing the syntax-restricted abduction reasoning capabilities of the SMT solver cvc5 [7].
- We extend the open-source SMT solver Bitwuzla [29] with our approach and show that it significantly improves performance on a wide range of benchmarks, including industrial benchmarks from smart contract verification.

Related Work. Developing scalable approaches for solving bit-vector formulas with large bit-widths is a long-standing challenge. Previous efforts to tackle this challenge can be mainly divided into two categories: alternative approaches to bit-blasting that primarily rely on word-level reasoning, and techniques based on bit-blasting that try to reduce the size of the original problem on the bit-level.

Alternative approaches to bit-blasting include: translations to linear integer arithmetic [11] and non-linear integer arithmetic (in combination with CEGAR-style handling of bit-wise operators) [36]; layered CDCL(T)-style approaches that rely on encoding fragments of the input problem into other theories before resorting to bit-blasting [13,21]; instances of the model-constructing satisfiability (mcSAT) calculus [20,35], a generalization of propositional conflict-driven clause learning (CDCL) to SMT; and incomplete techniques such as local search [19,

28,30], which are only able to determine satisfiability. All of these approaches are generally not competitive with bit-blasting.

Techniques based on bit-blasting that aim at mitigating the impact of increasing bit-widths on the bit-level are mainly based on some form of under-approximation. Bryant et al. [14] proposed a combination of under-approximation via restricting the value range of input variables with over-approximation of the unsat core of the under-approximated problem. This over-approximation consists of two strategies: eliminating if-then-else (*ite*) operations, and abstracting bit-vector multiplication $x \cdot y$ with a partially interpreted function of the form $\lambda x.\lambda y.ite(x \approx 0 \vee y \approx 0, 0, ite(x = 1, y, ite(y \approx 1, x, f(x, y)))$ where $f(x, y)$ is a fresh uninterpreted function. An early version of Boolector [12] implemented a refined version of the above under-approximation strategy in [14]. More recently, in the context of quantified bit-vector reasoning, Jonás et al. proposed an abstraction-based approach that reduces the size of the input problem via interpreting bits as don't care bits [22], and an under-approximation-based framework based on bit-width reduction [23] similar to [14].

2 Preliminaries

We assume and briefly review the usual notions and terminology of many-sorted first-order logic with equality (see, e.g., [18,25]). Let S be a set of *sort symbols*, and let Σ be a *signature* containing a set $\Sigma^s \subseteq S$ of sort symbols and a set Σ^f of function symbols $f^{\sigma_1 \cdots \sigma_n \sigma}$ with arity $n \geq 0$ and $\sigma_1, ..., \sigma_n, \sigma \in \Sigma^s$. We usually omit the superscript from function symbols and refer to 0-arity function symbols as *constants*. We assume that Σ includes a designated sort Bool, values $\top$ (true) and $\bot$ (false) of sort Bool, Boolean connectives $\{\wedge, \neg\}$ defined as usual, equality and disequality symbols $\{\approx, \not\approx\}$ of sort $\sigma \times \sigma \to \mathsf{Bool}$ for every $\sigma \in \Sigma^s$, and an if-then-else operator *ite* of sort $\mathsf{Bool} \times \sigma \times \sigma \to \sigma$ for every $\sigma \in \Sigma^s$.

Let $\mathcal{I}$ be a Σ *-interpretation* that maps each $\sigma \in \Sigma^s$ to a non-empty set $\sigma^{\mathcal{I}}$ (the *domain* of $\mathcal{I}$), with $\mathsf{Bool}^{\mathcal{I}} = \{\top, \bot\}$; and each $f^{\sigma_1 \cdots \sigma_n \sigma} \in \Sigma^f$ to a total function $f^{\mathcal{I}} \colon \sigma_1^{\mathcal{I}} \times ... \times \sigma_n^{\mathcal{I}} \to \sigma^{\mathcal{I}}$ if $n > 0$, and to an element in $\sigma^{\mathcal{I}}$ if $n = 0$. The interpretation of Boolean connectives, Boolean values, equality symbols and *ite* symbols is fixed and standard. We use the usual inductive definition of the satisfiability relation $\models$ between Σ-interpretations and Σ-formulas. We write $\varphi[x_1, \ldots, x_n]$ to denote a Σ-formula φ defined over (a subset of) symbols $\{x_1, \ldots, x_n\}$. We further use $\varphi[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$ for the formula obtained from φ by simultaneously replacing each occurrence of x_i with a_i.

A *theory* is a pair (Σ, I) where Σ is some signature, and I is a class of Σ-interpretations. A Σ-formula 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. We assume the usual definition of well-sorted terms, literals, and formulas, and call Σ-formulas T-formulas and Σ-literals T-literals.

We focus on the theory of fixed-size bit-vectors T_{BV} as defined by the SMT-LIB 2 standard [8]. The theory of fixed-size bit-vectors T_{BV} is defined as the pair (Σ_{BV}, I_{BV}). Signature Σ_{BV} includes a unique sort $\sigma_{[w]}$ for each bit-width w,

function symbols overloaded for every $\sigma_{[w]}$, and all *bit-vector values* of sort $\sigma_{[w]}$ for each w. The non-empty class of Σ_{BV}-interpretations I_{BV} (the *models* of T_{BV}) interpret sort and function symbols as specified in SMT-LIB 2.

Without loss of generality, we consider Σ_{BV} to contain a restricted, arbitrary set of bit-vector operators as listed in Table 1. This set is complete in the sense that it suffices to express all bit-vector operators defined in SMT-LIB 2. We further use logical connectives $\{\vee, \Rightarrow, \Leftrightarrow\}$ and bit-vector operator $-$ for subtraction and negation as shorthand when convenient. In the context of this paper it is important to note that both bit-vector subtraction and negation are expressed in terms of bit-vector addition.

We denote a Σ_{BV}-term (or *bit-vector term*) x of width w as $x_{[w]}$ when we want to specify its bit-width explicitly, and will omit w from the notation when it is clear from the context. The width of a bit-vector term is given by function κ, e.g., $\kappa(x_{[w]}) = w$. We refer to the bit at index i of $x_{[w]}$ as $x[i]$ and represent a bit-vector value $v_{[w]}$ as a bit-string of 0 s and 1 s, with the most significant bit (MSB) as the left-most bit $v[msb]$ at index $msb = w - 1$, and the least significant bit (LSB) as the right-most bit $v[lsb]$ at index $lsb = 0$. To simplify the notation, we will sometimes represent a value $v_{[w]}$ as a natural number in $\{0, \ldots, 2^{w-1}\}$.

Table 1. Set of considered bit-vector operators.

Symbol	SMT-LIB Syntax	Sort
$<_u$, $\leq_u$, $>_u$, $\geq_u$	`bvult`, `bvule`, `bvugt`, `bvuge`	$\sigma_{[w]} \times \sigma_{[w]} \rightarrow$ Bool
$\sim$	`bvnot`	$\sigma_{[w]} \rightarrow \sigma_{[w]}$
$\&$, $\mid$, $\oplus$, $\ll$, $\gg$	`bvand`, `bvor`, `bvxor`, `bvshl`, `bvlshr`	$\sigma_{[w]} \times \sigma_{[w]} \rightarrow \sigma_{[w]}$
$+$, $\cdot$, mod, $\div$	`bvadd`, `bvmul`, `bvurem`, `bvudiv`	$\sigma_{[w]} \times \sigma_{[w]} \rightarrow \sigma_{[w]}$
$\circ$	`concat`	$\sigma_{[w]} \times \sigma_{[m]} \rightarrow \sigma_{[w+m]}$
$[u : l]$	`extract` ($l \leq u < w$)	$\sigma_{[w]} \rightarrow \sigma_{[u-l+1]}$

3 Abstraction-Refinement Framework

Our abstraction-refinement framework is integrated into an SMT solver as a CEGAR procedure that combines an abstraction module with the theory solver that is responsible for reasoning about T_{BV}-formulas (the *bit-vector solver*). Since our main goal is to improve the scalability of *bit-blasting*, we assume that the bit-vector solver implements bit-blasting as its main strategy. For simplicity, we further assume that bit-blasting is its only strategy. However, this is not a requirement. Our abstraction-refinement technique can be combined with any complete technique for determining the satisfiability of T_{BV}-formulas that produces models for satisfiable formulas.

Algorithm 1 shows the main abstraction-refinement procedure of our approach. Given a set of bit-vector constraints $\mathcal{A}$, the abstraction module (AM)

Algorithm 1. Abstraction-refinement loop around the T_{BV}-solver.

```
1  function AbstractSolveBV(A)
2      result ← unknown, L ← ∅
3      A' ← AM::abstract(A)                          ▷ generate abstraction
4      repeat
5          A' ← A' ∪ L                               ▷ refine abstraction
6          result, M ← T_BV::solve(A')               ▷ query bit-vector solver
7          if result = unsat then break
8          L ← AM::check(M)                          ▷ check consistency
9      until L = ∅
10     return result
11 end function
```

first generates an abstraction $\mathcal{A}'$ of $\mathcal{A}$ (AM::abstract) by replacing abstracted terms with fresh constants. This abstraction is then iteratively refined with lemmas $\mathcal{L}$, starting from an empty set. First, the bit-vector solver is queried for a satisfiability result of the current abstraction $\mathcal{A}'$ and a model $\mathcal{M}$ of $\mathcal{A}'$ if it is satisfiable (T_{BV}::solve). If $\mathcal{A}'$ is unsatisfiable, the procedure concludes with *unsat*. If $\mathcal{A}'$ is satisfiable, the abstraction module checks the consistency of $\mathcal{M}$ for all abstracted terms with respect to their true semantics (AM::check) as follows. Starting from an empty set of refinement lemmas $\mathcal{L}$, for each abstracted term, function AM::check determines if the model value of its abstraction is consistent. If it is inconsistent, we add a refinement lemma to $\mathcal{L}$ that rules out the inconsistency. When the model values of all abstracted terms have been checked for consistency, AM::check returns the set of refinement lemmas $\mathcal{L}$, which extends abstraction $\mathcal{A}'$ in the next iteration. If model $\mathcal{M}$ is consistent for all abstracted terms (i.e., $\mathcal{L} = \emptyset$), the procedure concludes with *sat*.

Note that conceptually, our term abstractions are uninterpreted functions that map bit-vector arguments to a term of bit-vector sort, e.g., $mul_{32}(x, s)$ of sort $\sigma_{[32]} \times \sigma_{[32]} \rightarrow \sigma_{[32]}$ as abstraction of a bit-vector multiplication $x_{[32]} \cdot s_{[32]}$. When combining bit-vector theory reasoning with UF theory reasoning, from the point of view of the bit-vector solver, these UF are seen as fresh bit-vector constants. However, by construction, our procedure ensures that term abstractions are refined until consistency. Thus, when the UF theory solver is invoked after the bit-vector theory solver, additional UF theory reasoning is not required. Hence, introducing uninterpreted functions is redundant—it is sufficient to introduce a fresh constant of the same bit-vector sort as the abstracted term, e.g., $mul^{x,s}_{[32]}$ for $x_{[32]} \cdot s_{[32]}$. This allows the integration of our approach into any SMT solver that supports bit-vector reasoning, even when UF reasoning is not supported. Preliminary experiments showed that in the context of integrating our techniques in the SMT solver Bitwuzla, using UF as abstractions and scheduling the UF theory solver prior to our abstraction-refinement loop introduced redundant overhead and negatively impacted performance. Our approach, however, allows to freely choose between introducing UF vs. fresh bit-vector constants, depending on what is more beneficial for a specific solver architecture.

One of the main tasks of the *abstraction module* is consistency checking of satisfying assignments of the current abstraction, and refining the abstraction in case of inconsistency. This refinement is driven by a pre-defined *refinement scheme* for each abstracted operator. A refinement scheme is a four-tiered set of lemmas that is checked tier-wise, in ascending order, during consistency checking. We describe the refinement scheme for each operator and their tiers in more detail in Sect. 4.

4 Refinement Schemes

We define four-tiered refinement schemes for bit-vector operators $\diamond \in \{\cdot, \div, \text{mod}\}$, with tiers 1–2 as the main and predefined sets of refinement lemmas that describe properties of the abstracted operators in the usual bit-vector semantics (notably, with respect to overflow semantics). The *first* tier consists of *hand-crafted* lemmas that mostly encode basic properties (described in more detail in Sect. 4.1), while the *second* tier is entirely comprised of lemmas that were *synthesized* via our abduction-based lemma synthesis framework (see Sect. 4.3).

The *third* tier is not pre-defined but encodes so-called *value instantiation lemmas* to rule out the current inconsistent model value as a limited fallback strategy before we have to, as the *fourth* and final tier, resort to *bit-blasting*. For example, for $x_{[32]} \cdot s_{[32]}$ with $\mathcal{M} = \{x = 3, s = 6, mul^{x,s}_{[32]} = 1\}$, we add $(x = 3 \wedge s = 6) \Rightarrow mul^{x,s}_{[32]} = 18$ as value instantiation lemma. Value instantiation lemmas are only added if none of the lemmas in previous tiers were violated. We further limit the number of value instantiation lemmas that are added for an abstracted term since they each only rule out a single spurious model value of the term abstraction (see Sect. 5). Lemmas in tiers 1–2 do not necessarily fully capture all properties of an abstracted operator, and thus, inconsistent assignments may remain uncovered. When this is the case and the number of value instantiation lemmas to add is exhausted, we add a so-called *bit-blasting lemma*, e.g., $mul^{x,s}_{[32]} \approx x \cdot s$, which enforces bit-blasting of the abstracted term.

Note that of the considered arithmetic operators, addition is the only one we do not abstract. Even though addition is more expensive when bit-blasting compared to bit-wise operators, it is considerably cheaper than the operators we abstract. Preliminary experiments showed that the trade-off between abstracting the addition operator (which also occurs in our lemmas) versus bit-blasting addition terms suggests that it is more beneficial to not abstract addition.

Table 2 lists all lemmas of tiers 1–2 for all three operators, with hand-crafted lemmas marked with an asterisk. We use x for the left-hand operand, s for the right-hand operand, and t for the constant introduced to abstract $x \diamond s$. We further indicate with a subscript on the lemma ID if there is a restriction on the bit-widths for which the lemma is correct (see Sect. 4.4). Note that while our abstraction approach does not generally restrict the bit-width of operators to abstract, lemmas that are incorrect for certain bit-widths must be removed from the lemma sets when terms of that size are abstracted. In practice, we only

abstract terms of bit-width 32 and above (see Sect. 5) and thus these restrictions are not applicable. Further, note that in practice we consider both commutative cases (when applicable) while Table 2 only gives one. In the following, we describe our set of hand-crafted lemmas, our lemma scoring scheme and how we derive lemmas via abduction reasoning in more detail.

4.1 Hand-Crafted Lemmas

For each refinement scheme, our set of hand-crafted lemmas mostly contains lemmas that cover basic properties of the abstracted operators (e.g., when one of its operands is a special value). We also include lemmas that describe more elaborate properties based on invertibility conditions [31], i.e., conditions that exactly describe when operand x of operator $\diamond$ has a solution in literal $x \diamond s \approx y$. More formally, an invertibility condition IC for a literal $\varphi[x, s, y]$ is a formula defined over s and y such that $\exists x.\, \varphi \Leftrightarrow IC$. In the following, we summarize the properties encoded by each hand-crafted lemma.

Multiplication. Lemmas 1–2 capture the fact that multiplication by a power of 2 (and its arithmetic negation) can be described as a left shift operation. Lemma 3 states that the result of the multiplication must have at least as many trailing zeros in its binary representation as one of its arguments and is derived from the invertibility condition $(-s \mid s)\ \&\ y \approx y$ for $x \cdot s \approx y$. The left-to-right direction of $\exists x.\, \varphi \Leftrightarrow IC$ gives us (after Skolemization) the implication $x \cdot s \approx y \Rightarrow (-s \mid s)\ \&\ y \approx y$, of which lemma 3 is the right-hand side. Lemma 4 is a parity lemma that states that the result of a multiplication $x \cdot s$ must be odd if both x and s are odd, and even otherwise. Note that properties related to multiplication by special values 1, -1 and 0 are subsumed by lemmas 1, 2 and 3, respectively. Further note that [31] also provides invertibility conditions for literals defined over disequality and inequalities. We only consider invertibility conditions for literals $x \diamond s \approx y$ as this allows to instantiate y in the corresponding lemma with term abstraction t. For literals over predicates other than equality, e.g., $x \diamond s <_u y$, a good strategy for instantiating y in the resulting lemma is not obvious and left to future work.

Division. Lemma 1 states that unsigned division by a power of 2 can be described as a logical right shift operation. Lemmas 2–3 cover special cases: division by itself and division by 0 (the latter is a defined case in SMT-LIB). Lemma 4 states that zero divided by a non-zero value is zero. Lemma 5 captures a natural property of division by a non-zero value: its result is always less than its left-hand argument. Lemma 6 describes the property that division by $\sim$0 (the maximum unsigned value) yields zero if the dividend is less than $\sim$0. Note that for division, we do not utilize the corresponding invertibility conditions from [31] since they introduce new division terms that may not yet appear in the input constraints, which may lead to non-termination of the abstraction procedure.

Table 2. Lemmas for terms $x_{[w]} \diamond s_{[w]}$ with $\diamond \in \{\cdot, \div, \bmod\}$. We use t for the constant introduced to abstract $x \diamond s$, hand-crafted lemmas are marked with $*$, and $i \in [0, w-1]$. Lemma ID subscripts indicate bit-width restrictions for correctness.

bvmul			
1^*	$s \approx 2^i \Rightarrow t \approx x \ll i$	$11_{>1}$	$t \not\approx (1 \mid \sim(x \oplus s))$
2^*	$s \approx -2^i \Rightarrow t \approx -x \ll i$	$12_{>1}$	$t \not\approx (\sim 1 \mid (x \oplus s))$
3^*	$((-s \mid s)\ \&\ t) \approx t$	13	$x \not\approx ((x \ll (s+t)) - 1)$
4^*	$t[0] \approx (x[0]\ \&\ s[0])$	14	$x \not\approx (1 - (x \ll (s-t)))$
$5_{>1}$	$s \not\approx \sim(t \mid (1\ \&\ (x \mid s)))$	15	$s \not\approx (1 + (s \ll (t-x)))$
$6_{>1}$	$(x\ \&\ t) \not\approx (s \mid \sim t)$	16	$s \not\approx (1 - (s \ll (t-x)))$
$7_{>1}$	$t \not\approx ((s \mid 1) \ll (t \ll x))$	17	$s \not\approx (1 + (s \ll (x-t)))$
8	$s \approx (s \ll (x\ \&\ (1 \gg t)))$	$18_{>1}$	$t \not\approx (1 \mid (x+s))$
$9_{\neq 2}$	$t \geq_u (1\ \&\ ((x\ \&\ s) \gg 1))$	19	$x \not\approx \sim(x \ll (s+t))$
10	$x \not\approx (1 \oplus (x \ll (s \oplus t)))$		
bvudiv			
1^*	$s \approx 2^i \Rightarrow t \approx x \gg i$	19	$(x \gg t) \not\approx (s \mid t)$
2^*	$(s \approx x \wedge s \not\approx 0) \Rightarrow t \approx 1$	20	$s \not\approx \sim(s \gg (t \gg 1))$
3^*	$s \approx 0 \Rightarrow t \approx \sim 0$	$21_{>1}$	$x \not\approx \sim(x\ \&\ (t \ll 1))$
4^*	$(x \approx 0 \wedge s \not\approx 0) \Rightarrow t \approx 0$	22	$t \geq_u ((x \ll 1) \gg s)$
5^*	$s \not\approx 0 \Rightarrow t \leq_u x$	23	$x \geq_u (s \ll \sim(x \mid t))$
6^*	$(s \approx \sim 0 \wedge x \not\approx \sim 0) \Rightarrow t \approx 0$	24	$x \geq_u (t \ll \sim(x \mid s))$
7	$x \geq_u -(-s\ \&\ -t)$	25	$x \geq_u (t \oplus (t \gg (s \gg 1)))$
8	$-(s \mid 1) \geq_u t$	26	$x \geq_u (s \oplus (s \gg (t \gg 1)))$
9	$t \not\approx -(s\ \&\ \sim x)$	27	$x \geq_u (s \ll \sim(x \oplus t))$
10	$(s \mid t) \not\approx (x\ \&\ \sim 1)$	28	$x \geq_u (t \ll \sim(x \oplus s))$
11	$(s \mid 1) \not\approx (x\ \&\ \sim t)$	29	$x \not\approx (t + (s \mid (x+s)))$
12	$(x\ \&\ -t) \geq_u (s\ \&\ t)$	$30_{>2}$	$x \not\approx (t + (1 + (1 \ll x)))$
13	$s \geq_u (x \gg t)$	31	$s \geq_u ((x+t) \gg t)$
14	$x \geq_u ((s \gg (s \ll t)) \ll 1)$	$32_{>1}$	$x \not\approx (t + (t + (x \mid s)))$
15	$x \geq_u ((t \ll 1) \gg (t \ll s))$	33	$(s \oplus (x \mid t)) \geq_u (t \oplus 1)$
16	$t \geq_u ((x \gg s) \ll 1)$	34	$t \geq_u (x \gg (s-1))$
17	$x \geq_u ((x \mid t)\ \&\ (s \ll 1))$	35	$(s-1) \geq_u (x \gg t)$
18	$x \geq_u ((x \mid s)\ \&\ (t \ll 1))$	$36_{\neq 2}$	$x \not\approx (1 - (x \ll (x-t)))$
bvurem			
1^*	$s \approx 2^i \Rightarrow t \approx (0_{[\kappa(x)-i]} \circ x[i-1:0])$	9	$x \geq_u (t \mid (x\ \&\ s))$
2^*	$s \not\approx 0 \Rightarrow t \leq_u s$	10	$1 \not\approx (t\ \&\ \sim(x \mid s))$
3^*	$x \approx 0 \Rightarrow t \approx 0$	11	$t \not\approx (\sim x \mid -s)$
4^*	$s \approx 0 \Rightarrow t \approx x$	12	$(t\ \&\ (x \mid s)) \geq_u (t\ \&\ 1)$
5^*	$s \approx x \Rightarrow t \approx 0$	$13_{>2}$	$x \not\approx (-x \mid -\sim t)$
6^*	$x <_u s \Rightarrow t \approx x$	14	$(x + -s) \geq_u t$
7^*	$\sim - s \geq_u t$	15	$(-s \oplus (x \mid s)) \geq_u t$
8	$x \approx (x\ \&\ (s \mid (t \mid -s)))$		

Remainder. Lemma 1 exploits the fact that unsigned division by a power of 2 can be described as a logical right shift operation: the resulting remainder corresponds to the value of the bits that are shifted out. Lemma 2 states that a division by a non-zero divisor yields a remainder that cannot be greater than the divisor. Lemmas 3–5 cover special cases: when one of the operands is zero, and division by itself. Lemma 6 captures the fact that a division with a dividend that is less than the divisor yields the dividend as the remainder. Lemma 7 is derived from invertibility condition $\sim - s \geq_u y$ for $x \bmod s \approx y$ from [31] in a similar manner as the lemma derived from the invertibility condition for multiplication.

Powers of Two Lemmas. The powers of two lemmas for multiplication (lemmas 1–2), division (lemma 1), and remainder (lemma 1) use 2^i to denote a specific power of two. They do not symbolically encode whether a term s represents a power of two since this would require counting the number of trailing zero bits i. Instead, if the current model value of s is a power of two, we instantiate the corresponding lemma with this value. In the worst case, this will add $\kappa(s)$ instantiations of the lemma if all powers of two for bit-width $\kappa(s)$ are enumerated. However, this is rarely the case and the lemmas are cheap in terms of bit-blasting.

4.2 Lemma Scoring Scheme

Compiling a set of lemmas to describe properties of an abstracted operator $\diamond$ requires careful consideration of several key aspects: (*i*) lemmas for $\diamond$ should not introduce new terms that will be abstracted (introducing new terms with $\diamond$ may lead to non-termination of the abstraction procedure and introducing terms with abstracted operators other than $\diamond$ may yield potentially expensive abstractions in case they have to be bit-blasted); (*ii*) lemmas should minimize introducing new terms with potentially expensive operators that are not abstracted (e.g., bit-vector addition); and (*iii*) possible candidate lemmas should be *filtered* based on their *quality* to avoid adding redundant (subsumed) lemmas and to ensure that included lemmas maximize the number of spurious models to rule out.

The former two impose syntax restrictions (see Sect. 4.3), and for the purpose of addressing (*iii*), we define a scoring scheme that measures the quality of a candidate lemma for operator $\diamond$ as follows.

Definition 1 (Lemma Score). *Let $x \diamond s$ be the term to abstract, and let t be the constant abstracting $x \diamond s$ such that $x \diamond s \approx t$. Given a lemma $\ell[x, s, t]$ defined over $\{x, s, t\}$ such that $x \diamond s \approx t \Rightarrow \ell$. We define* $\textsc{Score}(\ell, w)$*, the score of ℓ for a given bit-width w, as the number of triplets (v^x, v^s, v^t) of bit-vector values of bit-width w where $\ell[x \mapsto v^x, s \mapsto v^s, t \mapsto v^t]$ evaluates to $\top$.*

For a term $x_{[4]} \diamond s_{[4]}$, the worst possible lemma score is the number of all possible combinations of triplets ($2^4 \times 2^4 \times 2^4 = 4096$), and the best possible score is the number of possible combinations of x and s ($2^4 \times 2^4 = 256$). Thus, the difference between the worst and best possible lemma score for any $x \diamond s$ is the

number of *incorrect* triplets, i.e., triplets for which $v^x \diamond v^s \not\approx v^t$. Since lemmas over-approximate literals $x \diamond s \approx t$, their score is a measure for the *degree of over-approximation*: a lower score indicates higher quality of a lemma as a higher number of incorrect triplets is ruled out.

For our hand-crafted lemmas for multiplication from Sect. 4.1, for bit-width 4 we compute as scores: {1: 2416, 2: 2791, 3: 1961, 4: 2048}. This indicates that they, individually, rule out 34–55% of incorrect triplets. Further, lemma 3, the lemma derived via the invertibility condition for multiplication over equality, is the strongest lemma of the four. Similarly, our hand-crafted lemmas for division and remainder rule out 6–50% of incorrect triplets for bit-width 4, with lemma 5 the strongest lemma for division, and lemma 7, the lemma derived from an invertibility condition, the strongest for remainder.

Individual lemma scores are a valuable measure of quality for a single lemma. However, triplet coverage for individual lemmas may intersect. Thus, when considered as a set, in a refinement scheme, it is necessary to define a measure for the quality of sets of lemmas to determine if extending the set with additional lemmas improves the number of incorrect triplets that are ruled out.

Definition 2 (Score of Lemma Set). *Given a set of lemmas $\mathcal{L}$ such that for each $\ell[x, s, t] \in \mathcal{L}$, $x \diamond s \approx t \Rightarrow \ell$. We define the score of $\mathcal{L}$ for a given bit-width w* $\textsc{Score}(\mathcal{L}, w)$ *as the number of triplets (v^x, v^s, v^t) of bit-vector values of bit-width w where $\bigwedge_{l \in \mathcal{L}} \ell[x \mapsto v^x, s \mapsto v^s, t \mapsto v^t] = \top$.*

For example, for $x_{[4]} \cdot s_{[4]}$, the score of the set of hand-crafted lemmas is 704, which indicates that it already rules out 88% of the incorrect triplets. Similarly, for division and remainder, for bit-width 4 the sets of hand-crafted lemmas rule out 71% and 91% of incorrect triplets. Note that extending a set of lemmas $\mathcal{L}$ with a lemma $\ell \notin \mathcal{L}$ can improve but not worsen its score. If ℓ is subsumed by $\mathcal{L}$, $\textsc{Score}(\mathcal{L}, w)$ remains unchanged. While our sets of hand-crafted lemmas from Sect. 4.1 already rule out a large number of incorrect triplets, their score also indicates that a considerable number of incorrect triplets is still not covered. We thus, in the following, propose an automated framework for synthesizing lemmas with respect to our sets of hand-crafted lemmas via abductive reasoning.

4.3 Synthesizing Lemmas via Abduction

The lemmas from Sect. 4.1 describe basic properties of the abstracted operators and are hand-crafted but strong, as indicated by their score. However, a considerable number of incorrect triplets is still uncovered for each set. Further, manually crafting lemmas that are effective with respect to an already existing set is challenging for arithmetic bit-vector operators, mainly due to overflow semantics. In this section, we propose an *automated* way to synthesize lemmas with respect to our sets of hand-crafted lemmas via *syntax-restricted abductive* reasoning [34] and focus on synthesizing lemmas for bit-vector operators $\{\cdot, \div, \text{mod}\}$. Our approach, however, can easily be generalized to other operators and theories.

Since we are over-approximating literals $x \diamond s \approx t$, we are trying to find lemmas $\ell[x, s, t]$ such that $(x \diamond s) \approx t \Rightarrow \ell$. Further, as mentioned in Sect. 4.2, we require that ℓ does not contain specific operators (the set of abstracted operators, including $\diamond$ itself) and that the number of occurrences of more expensive operators (such as bit-vector addition) is limited. The best possible over-approximation of operator $\diamond$ would exactly describe the semantics of $\diamond$ without including $\diamond$, which seems unattainable under the given constraints. The worst possible over-approximation, on the other hand, is the formula $\top$. We are thus looking for simple but *non-trivial* lemmas that improve the scores of our initial, hand-crafted lemma sets. We formulate this problem as an instance of the general *abduction* problem, which is defined as follows.

Definition 3 (T_{BV}-Abduct). *Given two quantifier-free T_{BV}-formulas A and B, a T_{BV}-abduct is a quantifier-free formula C such that $A \wedge C \Rightarrow B$ is T_{BV}-valid, and $A \wedge C$ is T_{BV}-satisfiable.*

Definition 4 (Non-trivial Lemma). *Given a T_{BV}-literal φ as $x \diamond s \approx t$, a φ*-lemma *$\ell[x, s, t]$ is a quantifier-free T-formula defined over $\{x, s, t\}$ such that $\varphi \Rightarrow \ell$ is T_{BV}-valid. Lemma ℓ is* non-trivial *if it is not T_{BV}-valid.*

Finding a non-trivial lemma ℓ for a given literal φ amounts to finding an abduct $\neg\ell$ of the formulas $\top$ and $\neg\varphi$.

Lemma 1. *Let φ be a T_{BV}-literal as above. T_{BV}-formula ℓ is a non-trivial φ-lemma if and only if $\neg\ell$ is a T_{BV}-abduct of the formulas $\top$ and $\neg\varphi$.*

Proof. Suppose $\neg\ell$ is a T_{BV}-abduct of $\top$ and $\neg\varphi$. In particular, $\top \wedge \neg\ell \Rightarrow \neg\varphi$, and therefore $\varphi \Rightarrow \ell$, and thus ℓ is a φ-lemma. And since, by Definition 3, $\top \wedge \neg\ell$ is T_{BV}-satisfiable, we get that ℓ is not T_{BV}-valid. For the converse, suppose ℓ is a non-trivial φ-lemma. Then, $\varphi \Rightarrow \ell$ is T_{BV}-valid. In particular, $\top \wedge \neg\ell \Rightarrow \neg\varphi$ is T_{BV}-valid. Further, since ℓ is not T_{BV}-valid, $\top \wedge \neg\ell$ is T_{BV}-satisfiable. □

Since we require certain syntactic restrictions for φ-lemmas, we base our lemma synthesis framework on the syntax-restricted abductive reasoning framework of [34] as implemented in the SMT solver cvc5 [7]. This abduction framework is based on Syntax-Guided Synthesis (SyGuS) [6] and thus guided by a user-defined grammar. Note that, alternatively, our lemma synthesis problem could be directly expressed as a SyGuS problem. However, non-triviality of lemmas requires the introduction of quantifiers in the specification of the formula to synthesize, whereas this quantification is implicit in the abduction formulation.

Our goal is to automatically extend a set of φ-lemmas $\mathcal{L}$ (may be empty) for a given literal φ (as defined above) with a set of lemmas Γ such that each lemma $\ell \in \Gamma$ improves the score of $\mathcal{L}$. Algorithm 2 shows the main procedure of our abduction-based lemma synthesis approach. Function SynthLem takes as input a literal φ, the bit-width w for which φ is defined, a set of initial lemmas $\mathcal{I}$, a set $\mathcal{G}$ of grammars that define syntax restrictions for lemma construction, and a limit n of number of lemmas to synthesize for each grammar. The procedure constructs and returns a set of φ-lemmas $\mathcal{L}$ such that $\mathcal{I} \subseteq \mathcal{L}$ and

Algorithm 2. Synthesizing lemmas. Function SynthLem assumes the availability of an abduction reasoner GetAbduct. Function Score computes the score of a set of lemmas w.r.t. a given bit-width w as in Definition 2.

```
1  function SynthLem(φ, w, I, G, n)
2      L ← I                                          ▷ Populate with initial lemmas
3      for γ in G do
4          Γ ← ∅
5          for i in [1, n] do                         ▷ Synthesize lemmas via abduction
6              a ← GetAbduct(⊤, ¬φ, γ)
7              if a = ⊥ then break
8              Γ ← Γ ∪ {¬a}
9          end for
10         repeat                                     ▷ Merge synthesized lemma with L
11             ℓ_min ← some ℓ ∈ Γ that minimizes Score(L ∪ {ℓ}, w)
12             L ← L ∪ {ℓ_min}, Γ ← Γ \ {ℓ_min}
13         until ℓ_min = ⊤ ∨ Γ = ∅
14     end for
15     return L
16 end function
```

$\mathcal{I} \subset \mathcal{L} \Rightarrow$ Score$(\mathcal{L}, w) <$ Score$(\mathcal{I}, w)$ as follows. The resulting set of lemmas $\mathcal{L}$ is initialized with the given set of initial lemmas $\mathcal{I}$ (in our case our hand-crafted lemmas). Then, for each grammar $\gamma \in \mathcal{G}$, in lines 5–9, first a set of at most n lemmas Γ is generated via abductive reasoning (GetAbduct). From this set, in lines 10–13, $\mathcal{L}$ is extended only with those lemmas ℓ that improve the score of $\mathcal{L}$. Lemmas are synthesized via an incremental abduction engine GetAbduct (in our case cvc5) by iteratively asking for n new T_{BV}-abducts of formulas $\top$ and $\neg\varphi$, constructed from the operators in grammar γ. Function GetAbduct returns $\bot$ if no more abducts are found (line 7), either because the search terminated or a resource limit was reached. Note that we used $\mathsf{n} = 100$ and a time limit of 100 s per call to GetAbduct. Both limits were found to be a good middle ground between generating sufficiently many lemmas while not overwhelming the solver with too many abduction queries.

In the context of synthesizing lemmas for T_{BV} operators, the search for lemmas via abduction is limited to formulas where the bit-width of T_{BV}-terms is explicitly given. Consequently, the T_{BV}-abducts determined via GetAbduct (and thus the resulting lemmas) are only guaranteed to be correct for this specific bit-width. Further, abductive reasoning for theory T_{BV} as in [34] is based on a T_{BV}-solver with the same limitations our abstraction-based approach aims to address: it relies on bit-blasting and thus does not scale well for increasing bit-widths. We thus chose a bit-width of 4 for x, s and t as a reasonable compromise to not overwhelm the abduction engine while avoiding the generation of lemmas that are specific to very small bit-widths. To minimize the risk of including bit-width specific lemmas in the set of synthesized lemmas $\mathcal{L}$, in function SynthLem, before adding lemma ℓ to $\mathcal{L}$, we introduce an additional step

where we verify the correctness of ℓ for bit-widths 4–10. And finally, before incorporating synthesized lemmas in our refinement schemes, we verify each lemma up to a certain, large bit-width (see Sect. 4.4). Note that while the additional verification step during synthesis encountered lemmas that were only valid for bit-width 4, no lemmas that passed this verification step failed verification for larger bit-widths. Further note that bit-vector multiplication is commutative. As an optimization we thus add the corresponding symmetric cases of hand-crafted lemmas to the set of initial lemmas $\mathcal{I}$ when applicable.

Our abduction-based lemma synthesis procedure requires the definition of a set of grammars $\mathcal{G}$ to describe syntax restrictions for constructing lemmas. Since the search space for SyGuS-based abduction heavily depends on such an input grammar, we opted for diversification via a set of grammars rather than a single, larger grammar. Set $\mathcal{G}$ consists of the of grammars γ_0 to γ_6 defined via a common grammar $\gamma_c = \{x, s, t, \approx, \not\approx, <_u, \leq_u, 0, 1\}$ as follows:

$$\begin{aligned}
\gamma_0 &= \gamma_c \cup \{\sim, \&, |, \oplus\} & \gamma_4 &= \gamma_3 \cup \{\oplus\} \\
\gamma_1 &= \gamma_c \cup \{-, \sim, \&, |\} & \gamma_5 &= \gamma_4 \cup \{+\} \\
\gamma_2 &= \gamma_1 \cup \{\oplus\} & \gamma_6 &= \gamma_c \cup \{-, +, -_+, \ll, \gg\} \\
\gamma_3 &= \gamma_1 \cup \{\ll, \gg\}
\end{aligned}$$

Note that in grammars γ_0 to γ_6 above, we use symbol '$-$' for negation and '$-_+$' for subtraction to ensure that they are distinguishable. Further note that we include bit-vector addition (and operators such as subtraction and negation that can be rewritten as addition) even though it is an arithmetic operation and thus one of the more expensive operators when bit-blasting. Preliminary experiments showed that including addition, negation and subtraction in some of the grammars is beneficial for finding useful lemmas.

Extending our set of hand-crafted lemmas from Sect. 4.1 with the lemmas synthesized via abduction as given in Table 2 improves the score for multiplication from 704 to 490, which corresponds to ruling out 94% of incorrect triplets for our final set of tier 1 and tier 2 lemmas. Similarly, the score for division improves from 1366 to 394 (96% coverage of incorrect triplets), and the score for remainder improves from 616 to 400 (96% coverage of incorrect triplets).

Finally, it is important to note that we synthesized lemmas via abduction in an offline manner, as opposed to during the solving process. That is, after automatically generating the lemmas, they were incorporated into the solver together with the hand-crafted lemmas. Thus, the set of incorporated tier 1 and tier 2 lemmas is fixed and independent from the input problem.

4.4 Lemma Verification

We verified the correctness of lemmas ℓ from Table 2 for bit-widths from 1–256 by checking for literal $x \diamond s \approx t$ if formula $x \diamond s \approx t \wedge \neg\ell$ is T-unsatisfiable. Given that the lemmas based on powers of two are well-known and universally valid properties of the corresponding bit-vector operators, we omit the additional

131,584 benchmarks required to check each instance of these lemmas up to bit-width 256. For the remaining lemmas, we generated 16,896 benchmarks and used the SMT solvers Bitwuzla [29], cvc5 [7], Yices [17], and Z3 [27] for verification. We ran these verification tasks on a cluster of 22 machines with Intel(R) Xeon(R) Gold 6348 CPUs. For each solver and benchmark pair, we used a CPU time limit of 8 h and a memory limit of 8GB. For a given bit-width, we consider a lemma to be correct if at least one solver determined *unsat*, and as incorrect if at least one solver determined *sat*. Overall, all solver-benchmark pairs required 1,112 d of CPU time. We did not encounter any disagreements between solvers and were able to complete all verification tasks, with Yices individually solving 96.49%, Bitwuzla 96.47%, cvc5 96.29%, and Z3 95.05% of all tasks.

We were able to verify the correctness of all hand-crafted lemmas for bit-widths 1–256, and of all synthesized lemmas for bit-widths 3–256. Synthesized lemmas are correct by construction for bit-width 4, which is confirmed by this experiment. However, some of the synthesized lemmas do not hold for very small bit-widths, as indicated by the bit-width restrictions given in Table 2. As mentioned above, if terms of such a restricted size are abstracted, these lemmas must not be considered for refinement. However, in the context of integrating our abstraction approach into Bitwuzla, all lemmas are applicable since we only abstract terms of size 32 and above (see Sect. 5).

Verification of the correctness of our lemmas up to bit-width 256 establishes sufficient confidence of their correctness for bit-widths larger than 256. We leave the task of formally proving their correctness for all bit-widths to future work. A recent technique for reasoning over bit-vectors with parametric bit-width based on a reduction to the quantified combination of the theories of uninterpreted functions and non-linear arithmetic was proposed in [32]. However, preliminary experiments showed that except for a small number of lemmas, verification of our lemmas using this technique is not feasible.

5 Integration

We extended the state-of-the-art SMT solver Bitwuzla [29] with our proposed framework. Bitwuzla supports quantified and quantifier-free bit-vector reasoning in combination with arrays, floating-point arithmetic and uninterpreted functions and was the best performing solver across supported logics in the SMT competition in 2023 [5]. Further, Bitwuzla reduces floating-point arithmetic to the theory of bit-vectors, which allows us to also apply our approach to floating-point arithmetic problems that do not involve bit-vector constraints.

Bitwuzla implements a lazy, CEGAR-based SMT paradigm called *lemmas on demand* [10,26], but with a bit-vector abstraction (and thus a T_{BV}-solver) instead of a propositional abstraction at its core. In this bit-vector abstraction, non-T_{BV}-atoms are abstracted as Boolean constants and non-T_{BV}-terms are abstracted as bit-vector constants. These abstracted terms are then handled by the corresponding theory solvers. This architecture allows an easy and seamless integration of our abstraction module. The interaction between the T_{BV}-solver of

Algorithm 3. The lemmas on demand loop of Bitwuzla with multiple theory solvers, extended with our abstraction module AM (highlighted in blue).

```
1  function SOLVE(A)
2      r ← UNKNOWN, L ← ∅
3      repeat
4          A ← AM::ABSTRACT(A ∪ L)
5          r, M ← T_BV::SOLVE(A)              ▷ Solve Bit-Vector Abstraction of A
6          if r = UNSAT then break end if
7          if (L ← T_FP::CHECK(M)) ≠ ∅ then continue end if        ▷ FP Solver
8          if (L ← AM::CHECK(M)) ≠ ∅ then continue end if
9          if (L ← T_A::CHECK(M)) ≠ ∅ then continue end if         ▷ Arrays Solver
10         if (L ← T_UF::CHECK(M)) ≠ ∅ then continue end if        ▷ UF Solver
11         L ← T_Q::CHECK(M)                                      ▷ Quantifiers Solver
12     until L = ∅
13     return r
14 end function
```

Bitwuzla and our abstraction module AM is implemented as shown in Algorithm 3. Prior to sending assertions to the T_{BV}-solver, the abstraction module processes each assertion and introduces abstractions for all relevant bit-vector terms. After the T_{BV}-solver determines that the set of abstracted assertions is satisfiable, the abstraction module checks if all abstracted bit-vector terms are consistent and adds refinement lemmas when needed.

Note that the order in which the theory solvers and the abstraction module are called is not arbitrary. The T_{FP}-solver word-blasts floating-point constraints to T_{BV} and, thus, introduces new bit-vector terms. Hence, the abstraction module is called after the T_{FP}-solver to ensure that for pure T_{FP}-formulas, the T_{FP}-solver first generates word-blasting lemmas so that the abstraction module has bit-vector terms to abstract. For the arrays (T_A) and UF (T_{UF}) theory solvers and the quantifiers module (T_Q), on the other hand, we have to ensure that the bit-vector abstraction is consistent before checking the theory axioms based on the current bit-vector abstraction model $\mathcal{M}$. In preliminary experiments, the abstraction module was called after the T_A- and T_{UF}-solvers, which resulted in a degraded performance for problems involving these theories. This was a consequence of the T_A- and T_{UF}-solvers generating substantially more lemmas due to an inconsistent bit-vector abstraction. Similarly, when quantifiers are involved, the quantifiers module is called last to ensure that the bit-vector abstraction of all ground terms and formulas is consistent.

As an additional extension, we also implemented a more coarse-grained abstraction approach that *abstracts assertions* as fresh Boolean constants. This is not a novel technique and has been proposed in earlier literature [24]. However, it can be easily implemented in our proposed abstraction framework with a simple refinement scheme for assertions. The goal of this refinement scheme is to incrementally add assertions as refinements that evaluate to $\bot$ under the current

model of the bit-vector abstraction. This is combined with our main approach of term abstraction in an interleaved manner by limiting the number of assertion refinements added per refinement iteration. When adding assertions as refinement, the abstraction module abstracts all relevant bit-vector terms occurring in these assertions, and before new assertions are added, it ensures that the current set of term abstractions is consistent. Only when all currently abstracted terms are consistent, more assertions may be added as refinement. The termination criteria are the same as with term abstraction only. If all of the remaining assertions evaluate to $\top$ under the current model, we conclude with *sat*. If a subset of the added assertions is already *unsatisfiable*, we found an *unsat core* and conclude with *unsat*.

Configuration. The number of assertion refinements per iteration is configurable and set to 100 refinements per iteration. Similarly, the minimum bit-width of terms defined over $\{\cdot, \div, \text{mod}\}$ that we abstract is configurable and limited to terms of size 32 and above. Further, since value instantiation lemmas only rule out one spurious model, our implementation limits the number of value instantiations per abstraction t based on its bit-width to $\kappa(t)/8$ instantiations. For example, for an abstracted term t of bit-width 32, we add at most four value instantiations before we add a bit-blasting lemma as final refinement for t.

6 Evaluation

We evaluate the performance of our bit-vector abstraction approach as integrated in Bitwuzla on five different benchmark sets: *certora* (1,988 benchmarks), *ethereum* (3,173 benchmarks), *syrew* (15,000 benchmarks), *ff* (1,224 benchmarks), and *smtlib* (155,269 benchmarks). Benchmark sets *certora* and *ethereum* are industrial benchmarks that arise from smart contract verification applications [15], provided by Certora [1] and the Ethereum Foundation [3]. The *certora* set consists of SMT queries generated by the Certora Prover [2] and is split into sets $certora_1$ and $certora_2$. The *ethereum* set contains benchmarks generated by hevm [4], a symbolic execution engine for the Ethereum virtual machine. Benchmarks in these sets are specifically encoded over bit-vectors of size 256, in combination with arrays, uninterpreted functions, and quantifiers.

Benchmark set *syrew* serves as a more controlled and balanced set to specifically evaluate the effectiveness of our abstraction approach for each abstracted operator. We generated three sets of equivalence checks, each only involving one of the abstracted operators. For that purpose, we enumerated T_{BV}-terms and T_{BV}-formulas that are equivalent for bit-width 4 with the SyGuS-solver of cvc5. For each set, we enumerated 500 equivalence checks using as SyGuS grammar $\{0, 1, x, s, t, \approx, \not\approx, <_u, \leq_u, \sim, \&, \ll, \gg\}$, extended with only one of $\{\cdot, \div, \text{mod}\}$. The resulting 1,500 benchmarks were then instantiated for bit-widths 2^i with $i \in [4, 13]$ yielding 15,000 benchmarks in total, the majority unsatisfiable.

The *ff* benchmark set originates from [33] and consists of translation validation problems of zero-knowledge proof compilers in two sets: an encoding in

the theory of finite fields T_{FF} and a translation to T_{BV} that exclusively uses arithmetic bit-vector operators $\{+, \cdot, \text{mod}\}$ over bit-vectors of size 510.

Benchmark set *smtlib* contains all non-incremental benchmarks of all logics in the SMT-LIB [9] benchmark library supported by Bitwuzla. This includes all quantified and quantifier-free logics involving the theories of bit-vectors, arrays, floating-point arithmetic and uninterpreted functions (24 in total). Note that this also includes floating-point arithmetic logics that do not involve the theory of bit-vectors since Bitwuzla word-blasts floating-point terms to bit-vector terms.

We implemented our novel *term abstraction* technique in our main configuration Abstr-t. We additionally distinguish two configurations that enable *assertion abstraction* as described in Sect. 5: configuration Abstr-a, which enables *assertion abstraction* only, and configuration Abstr-ta, which enables *both* term and assertion abstraction. We evaluate these configurations against Bitwuzla version 0.3.2, cvc5 version 1.1.0, and Z3 version 4.12.4 (in their default configuration, using bit-blasting for T_{BV}). Both cvc5 and Z3 are industrial-strength SMT solvers that support a wide range of theories, including the theories supported by Bitwuzla. We further compare against cvc5-ib, a configuration of cvc5 that reduces bit-vector problems to non-linear integer arithmetic problems via *int-blasting* [36]. Note that on the *ff* benchmark set, we evaluate these configurations only on the T_{BV} subset, and additionally compare against a dedicated T_{FF}-solver implementation of cvc5 (cvc5-ff) on the T_{FF} subset.

We ran all experiments on a cluster of 25 machines with Intel(R) Xeon E5-2620 v4 CPUs. For each solver and benchmark pair, we allocated one CPU core and 8GB of memory with a time limit of 1200 s. In case that a solver terminated with an error or ran into the memory limit on a specific benchmark, we counted its runtime on that benchmark as 1200 s as a penalty.

Table 3 summarizes the results for each solver grouped by benchmark set and ordered by number of solved benchmarks. Overall, Abstr-t significantly outperforms all other bit-blasting solvers and the int-blasting solver cvc5-ib on all benchmark sets. Our abstraction approach considerably reduces the memory usage across all sets, solving more benchmarks with a lower number of memory outs. Only on the *certora* sets, cvc5-ib has a smaller memory footprint, which is due to the more memory-efficient translation of bit-vector to integer arithmetic.

The *certora* set is divided into the *certora*$_1$ and *certora*$_2$ subsets, which correspond to the use of two different encodings arising from the same application. Both sets rely on 256-bit bit-vectors and uninterpreted functions and make heavy use of arithmetic operators. Set *certora*$_1$ is a proprietary and more diverse set of benchmarks and is sampled from a different (and more diverse) set of smart contracts than *certora*$_2$. It uses an older, less optimized encoding that involves quantifiers and overflow predicates, while *certora*$_2$ does not rely on quantifiers and was successfully optimized for existing bit-blasting solvers, which struggled on the older encoding. This can be seen in Table 3, where the best non-abstraction-based bit-blasting configuration (Bitwuzla) solves only 13% of *certora*$_1$ but 74% of *certora*$_2$. Benchmarks in the *certora*$_1$ set usually contain a large number of assertions (15k on average, up to 100k) and are thus

Table 3. Number of solved benchmarks (Solved), timeouts (TO), memory outs (MO), penalized runtime (T), memory usage of all benchmarks (M), and runtime T_c on commonly solved benchmarks, grouped by benchmark set and solvers. Note that the number (x/y) for each benchmark set indicates the number of commonly solved instances x and the total number of benchmarks y in the set.

Benchmarks	Solver	Solved	TO	MO	T [s]	M [GB]	T_c [s]
certora$_1$ (10/850)	Abstr-ta	573	231	46	448k	2,492	234
	Abstr-a	386	140	324	681k	5,201	963
	Abstr-t	258	155	437	760k	4,807	83
	cvc5-ib	147	674	0	879k	667	52
	Bitwuzla	111	86	653	915k	6,182	192
	cvc5	90	113	610	923k	6,064	341
	Z3	30	447	373	989k	4,944	484
certora$_2$ (227/1,138)	Abstr-ta	866	264	8	370k	1,024	11k
	Abstr-t	866	263	9	384k	1,402	17k
	Abstr-a	844	269	25	433k	2,661	19k
	Bitwuzla	843	266	29	439k	2,944	23k
	cvc5	705	223	210	603k	4,027	22k
	cvc5-ib	666	472	0	643k	106	15k
	Z3	612	492	34	679k	1,866	24k
ethereum (3,138/3,173)	Abstr-t	3,173	0	0	407	11	102
	Bitwuzla	3,173	0	0	720	29	228
	Z3	3,169	4	0	6k	107	679
	cvc5	3,158	0	1	18k	36	377
	cvc5-ib	3,141	20	0	39k	21	128
syrew (5,528/15,000)	Abstr-t	14,142	583	276	1,225k	4,409	2k
	Bitwuzla	11,961	744	2,296	3,955k	23,483	24k
	Z3	9,992	833	4,175	6,198k	39,506	78k
	cvc5	9,003	797	5,200	7,498k	48,421	109k
	cvc5-ib	7,974	5,137	1,632	8,836k	19,850	180k
ff (12/1,224)	cvc5-ff	973	129	122	313k	1,364	0
	Abstr-t	480	729	15	913k	2,762	0
	cvc5-ib	304	822	98	1,104k	1,074	0
	Bitwuzla	223	71	930	1,211k	8,360	277
	Z3	145	56	1,023	1,299k	8,893	3
	cvc5	40	0	1,184	1,422k	9,523	589
smtlib (125,037/155,269)	Abstr-t	148,554	1,944	152	8,770k	8,566	64k
	Bitwuzla	148,492	1,966	193	8,748k	8,953	64k
	Z3	145,121	4,846	565	13,528k	18,278	693k
	cvc5	144,829	3,775	285	13,513k	11,029	213k
	cvc5-ib	127,144	24,479	194	39,647k	15,233	5,666k

good candidates for evaluating assertion abstraction in combination with term abstraction. Benchmarks in the *certora*$_2$ set, on the other hand, usually contain a significantly smaller number of assertions (less than 1k per benchmark). Hence,

on the *certora* benchmark sets, in addition to configuration Abstr-t, we also evaluate the two configurations Abstr-a and Abstr-ta that enable assertion abstraction. On both sets, Abstr-t considerably improves over bit-blasting. On the *certora*$_1$ set, Abstr-a outperforms Abstr-t, and combining assertion and term abstraction (Abstr-ta) significantly outperforms either, both in terms of solved benchmarks and memory usage. We observed that in the majority of cases where Abstr-ta improves over Abstr-t, the benchmark is unsatisfiable and the size of the unsatisfiable core is only a small fraction of the overall number of assertions. On the *certora*$_2$ set, however, Abstr-a is less effective since these benchmarks contain a significantly smaller number of assertions. Configuration Abstr-ta still improves over Abstr-t in terms of overall memory usage.

Note that for the benchmark sets *ethereum*, *ff* and *syrew*, enabling assertion abstraction was not applicable for a majority of the benchmarks due to the low number of assertions (less than 100 per benchmark). On benchmark set *smtlib*, the effects of assertion abstraction were overall inconclusive. Thus, due to space constraints, for the remaining sets, we exclude configurations Abstr-a and Abstr-ta from the evaluation.

On the *ethereum* set, both Abstr-t and Bitwuzla solve all benchmarks. However, Abstr-t is more than 40% faster and requires 60% less memory. On the commonly 3,138 solved benchmarks, Abstr-t is the fastest solver, closely followed by cvc5-ib. Both outperform the other bit-blasting solvers. Note that on this benchmark set, cvc5 and cvc5-ib returned with errors due to unsupported cases of equality over constant arrays on 14 and 12 benchmarks, respectively.

On the *syrew* set, Abstr-t significantly outperforms all other solvers and is more than 3× faster with a 5× lower memory usage compared to the second best solver Bitwuzla. On the commonly solved 5,528 benchmarks, Abstr-t is 12–90× faster than the competition. The int-blasting configuration cvc5-ib comes in last, mainly due to the occurrence of *bit-wise* operations. Bit-wise operators do not have a direct translation to integers and require cvc5-ib to resort to abstraction schemes, which is more expensive than the direct translation via bit-blasting.

On the *ff* benchmark set, as expected, the native finite field solver cvc5-ff solves the most benchmarks overall. However, Abstr-t significantly improves over bit-blasting (Bitwuzla) and int-blasting (cvc5-ib) with the least number of memory outs overall. Surprisingly, Abstr-t is able to solve 36 benchmarks that cvc5-ff cannot. None of the other solvers solves benchmarks that cvc5-ff cannot.

On the *smtlib* set, Abstr-t improves over Bitwuzla in 10 out of the 24 logics in terms of number of solved benchmarks, with 6 of them being floating-point arithmetic logics. Most notably, Abstr-t was able to improve the number of solved instances X and runtime in percent Y on commonly solved instances (X, Y%) over Bitwuzla in logics `FP` (+5, −16%), `BVFP` (0, −45%), `QF_ABVFP` (+1, −33%), `QF_ABVFPLRA` (0, −23%), `QF_BVFP` (+1, −45%), `QF_BVFPLRA` (+9, −46%), `QF_FP` (+23, −13%), and `QF_FPLRA` (+1, -7%).

The only significant loss of -13 benchmarks is in the `QF_BV` logic, which is also the only logic where Abstr-t is significantly slower (33%) on commonly solved instances compared to Bitwuzla. This slowdown can be primarily attributed to the two benchmark families *Sage2* and *uclid*. On these two families, on the commonly solved instances, Abstr-t is slower by 40% and 4,100%, respectively. This slowdown is unexpected and needs further investigation. Nevertheless, in logic `QF_BV`, Abstr-t is able to solve more unsatisfiable benchmarks with less memory outs compared to Bitwuzla and outperforms cvc5, cvc5-ib and Z3 by a significant margin (more than 1,400 solved benchmarks).

Table 4. Number of overall abstracted terms and abstraction refinements on solved benchmarks grouped by abstracted operator and refinement tier (1: hand-crafted, 2: abduction, 3: value instantiation, 4: bit-blasting).

Terms		Refinement Tier				
Operator	*Abstracted*	*1*	*2*	*3*	*4*	*Total*
·	367,101	579,369	67,221	650,086	134,525	1,431,201
÷	55,461	126,223	109,137	73,019	7,024	315,403
mod	62,328	161,270	5,614	30,350	1,326	198,560

We further performed an analysis of term abstractions and abstraction refinements for all benchmarks solved by Abstr-t in all benchmark sets. Table 4 summarizes our findings, grouped by refinement tier and abstracted operator. Overall, Abstr-t abstracted 367,101 multiplication terms, 55,461 unsigned division terms, and 62,328 unsigned remainder terms. Out of these, only 134,525 (37%) multiplications, 7,024 (13%) divisions, and 1,326 (2%) remainders were bit-blasted as last resort via adding tier-4 lemmas. For the remaining 63%/87%/98% of multiplication/division/remainder terms, refinement with tier 1–3 lemmas only was sufficient to solve the benchmarks. Out of the solved benchmarks where Abstr-t abstracted any bit-vector terms, 80% were solved without bit-blasting any of the abstracted terms. For the remaining 20% of solved benchmarks, 78% of abstracted terms were bit-blasted.

For the benchmarks solved with abstraction, Abstr-t required on average 37 refinement iterations (median 4). Further, all lemmas except `bvudiv` lemma 21 and `bvurem` lemma 11 from Table 2 were used for solving these instances. Tier-1/2/3/4 lemmas were used in 76%/27%/30%/20% of solved instances.

We further evaluated the usefulness of the abduction-based lemmas (tier 2) by disabling these lemmas on the *syrew* benchmark set. Without these lemmas, Abstr-t solves 336 less benchmarks, has 2× more memory outs, and is 23% slower on commonly solved instances while consuming 61% more memory. Without tier-3 lemmas the number of solved instances for benchmark sets $certora_1/certora_2/syrew/ff/smtlib$ change by $-12\%/-1\%/-1\%/-6\%/+0.01\%$. The artifact of this paper is archived and available in the Zenodo open-access repository at https://zenodo.org/record/10913320.

7 Conclusion

We have presented a novel abstraction-refinement approach to improve the scalability of bit-blasting arithmetic terms with large bit-widths. We have introduced a lemma scoring scheme and an abduction-based framework for synthesizing refinement lemmas, which we include in our four-tiered refinement schemes. We have extended the state-of-the-art SMT solver Bitwuzla with our techniques and showed that this significantly improves solver performance on a diverse set of benchmarks coming from a variety of applications, including smart contract verification and zero-knowledge proofs. Incorporating existing under-approximation techniques with our approach is an interesting direction for future work.

References

1. Certora (2024). https://www.certora.com/
2. Certora prover white paper (2024). https://docs.certora.com/en/latest/docs/whitepaper/index.html
3. Ethereum foundation (2024). https://ethereum.foundation/
4. hevm symbolic execution engine smt queries (2024). https://github.com/msooseth/eth-bench-smt-queries
5. SMT competition 2023 (2024). https://github.com/smt-comp/2023
6. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, pp. 1–8. IEEE (2013). https://ieeexplore.ieee.org/document/6679385/
7. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
8. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017). http://smt-lib.org
9. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2023)
10. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, pp. 236–249. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_18
11. Bozzano, M., et al.: Encoding RTL constructs for MathSAT: a preliminary report. Electron. Notes Theor. Comput. Sci. **144**(2), 3–14 (2006)
12. Brummayer, R.: Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays. Ph.D. thesis, Informatik, Johannes Kepler University Linz (2009)
13. Bruttomesso, R., et al.: A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, pp. 547–560. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_54
14. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: An abstraction-based decision procedure for bit-vector arithmetic. Int. J. Softw. Tools Technol. Transf. **11**(2), 95–104 (2009). https://doi.org/10.1007/S10009-009-0101-X

15. Buterin, V.: Ethereum whitepaper (2023). https://ethereum.org/en/whitepaper/
16. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification, pp. 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2000). https://doi.org/10.1007/10722167_15
17. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification, pp. 737–744. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
18. Enderton, H.B.: A mathematical introduction to logic. Academic Press (1972)
19. Fröhlich, A., Biere, A., Wintersteiger, C., Hamadi, Y.: Stochastic local search for satisfiability modulo theories. Proc. AAAI Conf. Artif. Intell. **29**(1) (2015). https://doi.org/10.1609/aaai.v29i1.9372
20. Graham-Lengrand, S., Jovanović, D., Dutertre, B.: Solving Bitvectors with MCSAT: explanations from bits and pieces. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part I, pp. 103–121. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_7
21. Hadarean, L., Bansal, K., Jovanović, D., Barrett, C., Tinelli, C.: A tale of two solvers: eager and lazy approaches to bit-vectors. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification, pp. 680–695. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_45
22. Jonáš, M., Strejček, J.: Abstraction of bit-vector operations for BDD-based SMT solvers. In: Fischer, B., Uustalu, T. (eds.) Theoretical Aspects of Computing – ICTAC 2018: 15th International Colloquium, Stellenbosch, South Africa, October 16–19, 2018, Proceedings, pp. 273–291. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-02508-3_15
23. Jonáš, M., Strejček, J.: Speeding up quantified bit-vector SMT Solvers by Bit-Width Reductions and Extensions. In: Pulina, L., Seidl, M. (eds.) Theory and Applications of Satisfiability Testing – SAT 2020: 23rd International Conference, Alghero, Italy, July 3–10, 2020, Proceedings, pp. 378–393. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51825-7_27
24. Kroening, D., Strichman, O.: Decision Procedures. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
25. Manzano, M.: Introduction to many-sorted logic. In: Many-sorted logic and its applications, pp. 3–86. John Wiley & Sons, Inc., New York, NY, USA (1993)
26. Moura, L.D., Rueß, H.: Lemmas on demand for satisfiability solvers. In: The 5th International Symposium on the Theory and Applications of Satisfiability Testing, SAT 2002, Cincinnati, USA, May 15, 2002 (2002)
27. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
28. Niemetz, A., Preiner, M.: Ternary propagation-based local search for more bit-precise reasoning. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020, pp. 214–224. IEEE (2020). https://doi.org/10.34727/2020/ISBN.978-3-85448-042-6_29, https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_29
29. Niemetz, A., Preiner, M.: Bitwuzla. In: Enea, C., Lal, A. (eds.) Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II, pp. 3–17. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37703-7_1

30. Niemetz, A., Preiner, M., Biere, A.: Propagation based local search for bit-precise reasoning. Formal Methods Syst. Des. **51**(3), 608–636 (2017). https://doi.org/10.1007/S10703-017-0295-6, https://doi.org/10.1007/s10703-017-0295-6
31. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: On solving quantified bit-vector constraints using invertibility conditions. Formal Methods Syst. Des. **57**(1), 87–115 (2021)
32. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C.W., Tinelli, C.: Towards satisfiability modulo parametric bit-vectors. J. Autom. Reason. **65**(7), 1001–1025 (2021). https://doi.org/10.1007/S10817-021-09598-9
33. Ozdemir, A., Kremer, G., Tinelli, C., Barrett, C.: Satisfiability modulo finite fields. In: Enea, C., Lal, A. (eds.) Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II, pp. 163–186. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37703-7_8
34. Reynolds, A., Barbosa, H., Larraz, D., Tinelli, C.: Scalable algorithms for abduction via enumerative syntax-guided synthesis. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part I, pp. 141–160. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_9
35. Zeljić, A., Wintersteiger, C.M., Rümmer, P.: Deciding bit-vector formulas with mcSAT. In: Creignou, N., Le Berre, D. (eds.) Theory and Applications of Satisfiability Testing – SAT 2016, pp. 249–266. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_16
36. Zohar, Y.: Bit-precise reasoning via int-blasting. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation: 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16–18, 2022, Proceedings, pp. 496–518. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_24