










Towards Bit-Width-Independent Proofs in SMT Solvers

Aina Niemetz¹ , Mathias Preiner¹ , Andrew Reynolds² , Yoni Zohar¹  ,
Clark Barrett¹ , and Cesare Tinelli² 

¹ Stanford University, Stanford, USA
yoniz@cs.stanford.edu

² The University of Iowa, Iowa City, USA

Abstract. Many SMT solvers implement efficient SAT-based procedures for solving fixed-size bit-vector formulas. These approaches, however, cannot be used directly to reason about bit-vectors of symbolic bit-width. To address this shortcoming, we propose a translation from bit-vector formulas with parametric bit-width to formulas in a logic supported by SMT solvers that includes non-linear integer arithmetic, uninterpreted functions, and universal quantification. While this logic is undecidable, this approach can still solve many formulas by capitalizing on advances in SMT solving for non-linear arithmetic and universally quantified formulas. We provide several case studies in which we have applied this approach with promising results, including the bit-width independent verification of invertibility conditions, compiler optimizations, and bit-vector rewrites.

1 Introduction

Satisfiability Modulo Theories (SMT) solving for the theory of fixed-size bit-vectors has received a lot of interest in recent years. Many applications rely on bit-precise reasoning as provided by SMT solvers, and the number of solvers that participate in the corresponding divisions of the annual SMT competition is high and increasing. Although theoretically difficult (e.g., [14]), bit-vector solvers are in practice highly efficient and typically implement SAT-based procedures. Reasoning about fixed-size bit-vectors suffices for many applications. In hardware verification, the size of a circuit is usually known in advance, and in software verification, machine integers are treated as fixed-size bit-vectors, where the width depends on the underlying architecture. Current solving approaches, however, do not generalize beyond this limitation, i.e., they cannot reason about parametric circuits or machine integers of arbitrary size. This is a serious limitation when one wants to prove properties that are bit-width independent. Further, when reasoning about machine integers of a fixed but large size, as employed, for example,

This work was supported in part by DARPA (awards N66001-18-C-4012 and FA8650-18-2-7861), ONR (award N68335-17-C-0558), NSF (award 1656926), and the Stanford Center for Blockchain Research.

in smart contract languages such as Solidity [28], current approaches do not perform as well in the presence of expensive operations such as multiplication [15].

To address this limitation we propose a general method for reasoning about bit-vector formulas with parametric bit-width. The essence of the method is to replace the translation from fixed-size bit-vectors to propositional logic (which is at the core of state-of-the-art bit-vector solvers) with a translation to the quantified theories of integer arithmetic and uninterpreted functions. We obtain a fully automated verification process by capitalizing on recent advances in SMT solving for these theories.

The reliability of our approach depends on the correctness of the SMT solvers in use. Interactive theorem provers, or proof assistants, such as Isabelle and Coq [20,29], on the other hand, target applications where trust is of higher importance than automation, although substantial progress towards increasing the latter has been made in recent years [5]. Our long-term goal is an efficient automated framework for proving bit-width independent properties within a trusted proof assistant, which requires both a formalization of such properties in the language of the proof assistant and the development of efficient automated techniques to reason about these properties. This work shows that state-of-the-art SMT solving combined with our encoding techniques make the latter feasible. The next steps towards this goal are described in the final section of this paper.

Translating a formula from the theory of fixed-size bit-vectors to the theory of integer arithmetic is not straightforward. This is due to the fact that the semantics of bit-vector operators are defined modulo the bit-width n , which must be expressed using exponentiation terms 2^n . Most SMT solvers, however, do not support unrestricted exponentiation. Furthermore, operators such as bit-wise *and* and *or* do not have a natural representation in integer arithmetic. While they are definable in the theory of integer arithmetic using β -function encodings (e.g., [10]), such a translation is expensive as it requires an encoding of sequences into natural numbers. Instead, we introduce an uninterpreted function (UF) for each of the problematic operators and axiomatize them with quantified formulas, which shifts some of the burden from arithmetic to UF reasoning. We consider two alternative axiomatizations: a complete one relying on induction, and a partial (hand-crafted) one that can be understood as an under-approximation.

To evaluate the potential of our approach, we examine three case studies that arise from real applications where reasoning about bit-width independent properties is essential. Niemetz et al. [19] defined invertibility conditions for bit-vector operators, which they then used to solve quantified bit-vector formulas. However, correctness of the conditions was only checked for specific bit-widths: from 1 to 65. As a first case study, we consider the bit-width independent verification of these invertibility conditions, which [19] left to future work. As a second case study, we examine the bit-width independent verification of compiler optimizations in LLVM. For that, we use the Alive tool [17], which generates verification conditions for such optimizations in the theory of fixed-size bit-vectors. Proving the correctness of these optimizations for arbitrary bit-widths would ensure their correctness for any language and underlying architecture rather than specific

ones. As a third case study, we consider the bit-width independent verification of rewrite rules for the theory of fixed-size bit-vectors. SMT solvers for this theory heavily rely on such rules to simplify the input. Verifying their correctness is essential and is typically done by hand, which is both tedious and error-prone.

To summarize, this paper makes the following contributions.

- In Sect. 3, we study complete and incomplete encodings of bit-vector formulas with parametric bit-width into integer arithmetic.
- In Sect. 4, we evaluate the effectiveness of both encodings in three case studies.
- As part of the invertibility conditions case study, we introduce *conditional inverses* for bit-vector constraints, thus augmenting [19] with concrete parametric solutions.

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

Symbol	SMT-LIB Syntax	Sort
$\approx, \not\approx$	$=$, distinct	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \mathbf{Bool}$
$\langle \! \! \rangle_u^{\mathbf{BV}}, \langle \! \! \rangle_s^{\mathbf{BV}}, \langle \! \! \rangle_u^{\mathbf{BV}}, \langle \! \! \rangle_s^{\mathbf{BV}}$	bvult, bvugt, bvslt, bvsgt	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \mathbf{Bool}$
$\leq_u^{\mathbf{BV}}, \geq_u^{\mathbf{BV}}, \leq_s^{\mathbf{BV}}, \geq_s^{\mathbf{BV}}$	bvule, bvuge, bvslle, bvsgle	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \mathbf{Bool}$
$\sim^{\mathbf{BV}}, \neg^{\mathbf{BV}}$	bvnot, bvneg	$\sigma_{[n]} \rightarrow \sigma_{[n]}$
$\&^{\mathbf{BV}}, ^{\mathbf{BV}}, \oplus^{\mathbf{BV}}$	bvand, bvor, bvxor	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \sigma_{[n]}$
$\ll^{\mathbf{BV}}, \gg^{\mathbf{BV}}, \gg_a^{\mathbf{BV}}$	bvshl, bvlsht, bvashr	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \sigma_{[n]}$
$+^{\mathbf{BV}}, \cdot^{\mathbf{BV}}, \text{mod}^{\mathbf{BV}}, \text{div}^{\mathbf{BV}}$	bvadd, bvmul, bvurem, bvudiv	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \sigma_{[n]}$
$[u : l]^{\mathbf{BV}}$	extract ($0 \leq l \leq u < n$)	$\sigma_{[n]} \rightarrow \sigma_{[u-l+1]}$
$\circ^{\mathbf{BV}}$	concatenation	$\sigma_{[n]} \times \sigma_{[m]} \rightarrow \sigma_{[n+m]}$

Related Work. Bit-width independent bit-vector formulas were studied by Picora [22], who introduced a formal language for bit-vectors of parametric width, along with a semantics and a decision procedure. The language we use here is a simplified variant of that language. A unification-based algorithm for bit-vectors of symbolic lengths is discussed by Bjørner and Picora [4]. Bit-width independent formulas are related to parametric Boolean functions and circuits. An inductive approach for reasoning about such formalisms was developed by Gupta and Fisher [11, 12] by considering a Boolean function for the base case of a circuit and another one for its inductive step. Reasoning about equivalence of such circuits can be embedded in the framework of [22].

2 Preliminaries

We briefly review the usual notions and terminology of many-sorted first-order logic with equality (denoted by \approx). See [10, 30] for more detailed information. 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 way. Constants are treated as 0-ary functions. We assume that Σ includes a Boolean sort **Bool** and the Boolean constants \top (true) and \perp (false). Functions returning **Bool** are also called *predicates*.

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 define $\mathbf{x} = (x_1, \dots, x_n)$ as a tuple of variables and write $Q\mathbf{x}\varphi$ with $Q \in \{\forall, \exists\}$ for a *quantified* formula $Qx_1 \cdots Qx_n\varphi$. For a Σ -term or Σ -formula e , we denote the *free variables* of e (defined as usual) as $\text{FV}(e)$ and use $e[\mathbf{x}]$ to denote that the variables in \mathbf{x} occur free in e . For a tuple of Σ -terms $\mathbf{t} = (t_1, \dots, t_n)$ and a tuple of Σ -variables $\mathbf{x} = (x_1, \dots, x_n)$, we write $e\{\mathbf{x} \mapsto \mathbf{t}\}$ for the term or formula obtained from e by simultaneously replacing each occurrence of x_i in e by t_i .

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 $f^\mathcal{I}: \sigma_1^\mathcal{I} \times \dots \times \sigma_n^\mathcal{I} \rightarrow \sigma^\mathcal{I}$ if $n > 0$, and to an element in $\sigma^\mathcal{I}$ if $n = 0$. We use the usual inductive definition of a 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 $\mathcal{I}' \in I$. 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 . We will sometimes omit T when the theory is understood from context.

The theory $T_{\text{BV}} = (\Sigma_{\text{BV}}, I_{\text{BV}})$ of fixed-size bit-vectors as defined in the SMT-LIB 2 standard [3] 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 $\sigma_{[n]}$. 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 $\{0, \dots, 2^{n-1}\}$. All interpretations $\mathcal{I} \in I_{\text{BV}}$ 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_{BV}^f are overloaded for every $\sigma_{[n]} \in \Sigma_{\text{BV}}^s$. We denote a Σ_{BV} -term (or *bit-vector term*) t of width n as $t_{[n]}$ when we want to specify its bit-width explicitly. We refer to the i -th bit of $t_{[n]}$ as $t[i]$ with $0 \leq i < n$. We interpret $t[0]$ as the least significant bit (LSB), and $t[n-1]$ as the most significant bit (MSB), and denote bit ranges over k from index j down to i as $t[j : i]$. The unsigned interpretation of a bit-vector $t_{[n]}$ as a natural number is given by $[t]_{\mathbb{N}} = \sum_{i=0}^{n-1} t[i] \cdot 2^i$, and its signed interpretation as an integer is given by $[t]_{\mathbb{Z}} = -t[n-1] \cdot 2^{n-1} + [t[n-2 : 0]_{\text{BV}}]_{\mathbb{N}}$.

Without loss of generality, we consider a restricted set of bit-vector function and predicate symbols (or *bit-vector operators*) as listed in Table 1. The selection of operators in this set is arbitrary but complete in the sense that it suffices to

express all bit-vector operators defined in SMT-LIB 2. We use $\max_{\text{s}[k]}^{\text{BV}}$ ($\min_{\text{s}[k]}^{\text{BV}}$) for the *maximum* or *minimum signed value* of width k , e.g., $\max_{\text{s}[4]}^{\text{BV}} = 0111$ and $\min_{\text{s}[4]}^{\text{BV}} = 1000$.

The theory $T_{\text{IA}} = (\Sigma_{\text{IA}}, I_{\text{IA}})$ of integer arithmetic is also defined as in the SMT-LIB 2 standard. The signature Σ_{IA} includes a single sort `Int`, function and predicate symbols $\{+, -, \cdot, \text{div}, \text{mod}, |\dots|, <, \leq, >, \geq\}$, and a constant symbol for every integer value. We further extend Σ_{IA} to include exponentiation, denoted in the usual way as a^b . All interpretations $\mathcal{I} \in I_{\text{IA}}$ are identical except for the values they assign to variables. We write T_{UFIA} to denote the (combined) theory of uninterpreted functions with integer arithmetic. Its signature is the union of the signature of T_{IA} with a signature containing a set of (freely interpreted) function symbols, called *uninterpreted functions*.

2.1 Parametric Bit-Vector Formulas

We are interested in reasoning about (classes of) Σ_{BV} -formulas that hold independently of the sorts assigned to their variables or terms. We formalize the notion of parametric Σ_{BV} -formulas in the following.

We fix two sets X^* and Z^* of variable and constant symbols, respectively, of bit-vector sort of undetermined bit-width. The bit-width is provided by the first component of a separate function pair $\omega = (\omega^b, \omega^N)$ which maps symbols $x \in X^* \cup Z^*$ to Σ_{IA} -terms. We refer to $\omega^b(x)$ as the *symbolic bit-width* assigned by ω to x . The second component of ω is a map ω^N from symbols $z \in Z^*$ to Σ_{IA} -terms. We call $\omega^N(z)$ the *symbolic value* assigned by ω to z . Let $\mathbf{v} = \text{FV}(\omega)$ be the set of (integer) free variables occurring in the range of either ω^b or ω^N . We say that ω is *admissible* if for every interpretation $\mathcal{I} \in I_{\text{IA}}$ that interprets each variable in \mathbf{v} as a positive integer, and for every $x \in X^* \cup Z^*$, \mathcal{I} also interprets $\omega^b(x)$ as a positive integer.

Let φ be a formula built from the function symbols of Σ_{BV} and $X^* \cup Z^*$, ignoring their sorts. We refer to φ as a *parametric Σ_{BV} -formula*. One can interpret φ as a class of fixed-size bit-vector formulas as follows. For each symbol $x \in X^*$ and integer $n > 0$, we associate a unique variable x_n of (fixed) bit-vector sort $\sigma_{[n]}$. Given an admissible ω with $\mathbf{v} = \text{FV}(\omega)$ and an interpretation \mathcal{I} that maps each variable in \mathbf{v} to a positive integer, let $\varphi|_{\omega[\mathcal{I}]}$ be the result of replacing all symbols $x \in X^*$ in φ by the corresponding bit-vector variable x_k and all symbols $z \in Z^*$ in φ by the bit-vector constant of sort $\sigma_{[k]}$ corresponding to $\omega^N(z)^{\mathcal{I}} \bmod 2^k$, where in both cases k is the value of $\omega^b(x)^{\mathcal{I}}$. We say a formula φ is *well sorted under ω* if ω is admissible and $\varphi|_{\omega[\mathcal{I}]}$ is a well-sorted Σ_{BV} -formula for all \mathcal{I} that map variables in \mathbf{v} to positive integers.

Example 1. Let X^* be the set $\{x\}$ and Z^* be the set $\{z_0, z_1\}$, where $\omega^N(z_0) = 0$ and $\omega^N(z_1) = 1$. Let φ be the formula $(x +^{\text{BV}}x) +^{\text{BV}}z_1 \not\approx z_0$. We have that φ is well sorted under (ω^b, ω^N) with $\omega^b = \{x \mapsto a, z_0 \mapsto a, z_1 \mapsto a\}$ or $\omega^b = \{x \mapsto 3, z_0 \mapsto 3, z_1 \mapsto 3\}$. It is not well sorted when $\omega^b = \{x \mapsto a_1, z_0 \mapsto a_1, z_1 \mapsto a_2\}$ since $\varphi|_{\omega[\mathcal{I}]}$ is not a well sorted Σ_{BV} -formula whenever $a_1^{\mathcal{I}} \neq a_2^{\mathcal{I}}$. Note that an

ω where $\omega^b(\mathbf{x}) = a_1 - a_2$ is not admissible, since $(a_1 - a_2)^{\mathcal{I}} \leq 0$ is possible even when $a_1^{\mathcal{I}} > 0$ and $a_2^{\mathcal{I}} > 0$.

Notice that symbolic constants such as the maximum unsigned constant of a symbolic length w can be represented by introducing $\mathbf{z} \in Z^*$ with $\omega^b(\mathbf{z}) = w$ and $\omega^N(\mathbf{z}) = 2^w - 1$. Furthermore, recall that signature Σ_{BV} includes the bit-vector extract operator, which is parameterized by two natural numbers u and l . We do not lift the above definitions to handle extract operations having symbolic ranges, e.g., where u and l are Σ_{IA} -terms. This is for simplicity and comes at no loss of expressive power, since constraints involving extract can be equivalently expressed using constraints involving concatenation. For example, showing that every instance of a constraint $s \approx t[u : l]^{\text{BV}}$ holds, where $0 < l \leq u < n - 1$, is equivalent to showing that $t \approx y_1 \circ^{\text{BV}}(y_2 \circ^{\text{BV}} y_3) \Rightarrow s \approx y_2$ holds for all y_1, y_2, y_3 , where y_1, y_2, y_3 have sorts $\sigma_{[n-1-u]}$, $\sigma_{[u-l+1]}$, $\sigma_{[l]}$, respectively. We may reason about a formula involving a symbolic range $\{l, \dots, u\}$ of t by considering a parametric bit-vector formula that encodes a formula of the latter form, where the appropriate symbolic bit-widths are assigned to symbols introduced for y_1, y_2, y_3 .

We assume the above definitions for parametric Σ_{BV} -formulas are applied to parametric Σ_{BV} -terms as well. Furthermore, for any admissible ω , we assume ω can be extended to terms t of bit-vector sort that are well sorted under ω such that $t|_{\omega[\mathcal{I}]}$ has sort $\sigma_{[\omega^b(t)^{\mathcal{I}}]}$ for all \mathcal{I} that map variables in $\text{FV}(\omega)$ to positive integers. Such an extension of ω to terms can be easily computed in a bottom-up fashion by computing ω for each child and then applying the typing rules of the operators in Σ_{BV} . For example, we may assume $\omega^b(t) = \omega^b(t_2)$ if t is of the form $t_1 +^{\text{BV}} t_2$ and is well sorted under ω , and $\omega^b(t) = \omega^b(t_1) + \omega^b(t_2)$ if t is of the form $t_1 \circ^{\text{BV}} t_2$.

Finally, we extend the notion of validity to parametric bit-vector formulas. Given a formula φ that is well sorted under ω , we say φ is T_{BV} -valid under ω if $\varphi|_{\omega[\mathcal{I}]}$ is T_{BV} -valid for all \mathcal{I} that map variables in $\text{FV}(\omega)$ to positive integers.

3 Encoding Parametric Bit-Vector Formulas in SMT

Current SMT solvers do not support reasoning about parametric bit-vector formulas. In this section, we present a technique for encoding such formulas as formulas involving non-linear integer arithmetic, uninterpreted functions, and universal quantifiers. In SMT parlance, these are formulas in the UFNIA logic. Given a formula φ that is well sorted under some mapping ω , we describe this encoding in terms of a translation \mathcal{T} , which returns a formula ψ that is valid in the theory of uninterpreted functions with integer arithmetic only if φ is T_{BV} -valid under ω . We describe several variations on this translation and discuss their relative strengths and weaknesses.

Overall Approach. At a high level, our translation produces an implication whose antecedent requires the integer variables to be in the correct ranges (e.g., $k > 0$ for every bit-width variable k), and whose conclusion is the result of converting each (parametric) bit-vector term of bit-width k to an integer term.

Operations on parametric bit-vector terms are converted to operations on the integers modulo 2^k , where k can be a symbolic constant. We first introduce uninterpreted functions that will be used in our translation. Note that SMT solvers may not support the full set of functions in our extended signature Σ_{IA} , since they typically do not support exponentiation. Since translation requires a limited form of exponentiation we introduce an uninterpreted function symbol pow2 of sort $\text{Int} \rightarrow \text{Int}$, whose intended semantics is the function $\lambda x.2^x$ when the argument x is non-negative. Second, for each (non-predicate) n -ary (with $n > 0$) function f^{BV} of sort $\sigma_1 \times \dots \times \sigma_n \rightarrow \sigma$ in the signature of fixed-size bit-vectors Σ_{BV} (excluding bit-vector extraction), we introduce an uninterpreted function f^{N} of arity $n + 1$ and sort $\text{Int} \times \text{Int} \times \dots \times \text{Int} \rightarrow \text{Int}$, where the extra argument is used to specify the bit-width. For example, for $+^{\text{BV}}$ with sort $\sigma_{[n]} \times \sigma_{[n]} \rightarrow \sigma_{[n]}$, we introduce $+^{\text{N}}$ of sort $\text{Int} \times \text{Int} \times \text{Int} \rightarrow \text{Int}$. In its intended semantics, this function adds the second and third arguments, both integers, and returns the result modulo 2^k , where k is the first argument. The signature Σ_{BV} contains one function, bit-vector concatenation \circ^{BV} , whose two arguments may have different sorts. For this case, the first argument of \circ^{N} indicates the bit-width of the third argument, i.e., $\circ^{\text{N}}(k, x, y)$ is interpreted as the concatenation of x and y , where y is an integer that encodes a bit-vector of bit-width k ; the bit-width for x is not specified by an argument, as it is not needed for the elimination of this operator we perform later. We introduce uninterpreted functions for each bit-vector predicate in a similar fashion. For instance, \geq_u^{N} has sort $\text{Int} \times \text{Int} \times \text{Int} \rightarrow \text{Bool}$ and encodes whether its second argument is greater than or equal to its third argument, when these two arguments are interpreted as unsigned bit-vector values whose bit-width is given by its first argument. Depending on the variation of the encoding, our translation will either introduce quantified formulas that fully axiomatize the behavior of these uninterpreted functions or add (quantified) lemmas that state key properties about them, or both.

Translation Function. Figure 1 defines our translation function \mathcal{T}_A , which is parameterized by an axiomatization mode A . Given an input formula φ that is well sorted under ω , it returns the implication whose antecedent is an *axiomatization* formula $\text{AX}_A(\varphi, \sigma)$ and whose conclusion is the result of converting φ to its encoded version via the conversion function CONV . The former is dependent upon the axiomatization mode A which we discuss later. We assume without loss of generality that φ contains no applications of bit-vector extract , which can be eliminated as described in the previous section, nor does it contain concrete bit-vector constants, since these can be equivalently represented by introducing a symbol in Z^* with the appropriate concrete mappings in ω^b and ω^N .

In the translation, we use an auxiliary function CONV which converts parametric bit-vector expressions into integer expressions with uninterpreted functions. Parametric bit-vector variables x (that is, symbols from X^*) are replaced by unique integer variables of type Int , where we assume a mapping χ maintains this correspondence, such that range of χ does not include any variable that occurs in $\text{FV}(\omega)$. Parametric bit-vector constants z (that is, symbols from set Z^*) are replaced by the term $\omega^N(z) \bmod \text{pow2}(\omega^b(z))$. The ranges of the maps

$\mathcal{T}_A(\varphi, \omega)$:		
Return $\text{AX}_A(\varphi, \omega) \Rightarrow \text{CONV}(\varphi, \omega)$.		
$\text{CONV}(e, \omega)$:		
Match e :		
x	$\rightarrow \chi(x)$	if $x \in X^*$
z	$\rightarrow \omega^N(z) \bmod \text{pow2}(\omega^b(z))$	if $z \in Z^*$
$t_1 \approx t_2$	$\rightarrow \text{CONV}(t_1, \omega) \approx \text{CONV}(t_2, \omega)$	
$f^{\text{BV}}(t_1, \dots, t_n)$	$\rightarrow \text{ELIM}(f^{\text{N}}(\omega^b(t_n), \text{CONV}(t_1, \omega), \dots, \text{CONV}(t_n, \omega)))$	
$\bowtie(\varphi_1, \dots, \varphi_n)$	$\rightarrow \bowtie(\text{CONV}(\varphi_1, \omega), \dots, \text{CONV}(\varphi_n, \omega))$	$\bowtie \in \{\wedge, \vee, \Rightarrow, \neg, \Leftrightarrow\}$
$\text{ELIM}(e)$:		
Match e :		
$+^{\text{N}}(k, x, y)$	$\rightarrow (x + y) \bmod \text{pow2}(k)$	
$-^{\text{N}}(k, x, y)$	$\rightarrow (x - y) \bmod \text{pow2}(k)$	
$\cdot^{\text{N}}(k, x, y)$	$\rightarrow (x \cdot y) \bmod \text{pow2}(k)$	
$\text{div}^{\text{N}}(k, x, y)$	$\rightarrow \text{ite}(y \approx 0, \text{pow2}(k) - 1, x \text{ div } y)$	
$\text{mod}^{\text{N}}(k, x, y)$	$\rightarrow \text{ite}(y \approx 0, \text{pow2}(k) - 1, x \bmod y)$	
$\sim^{\text{N}}(k, x)$	$\rightarrow \text{pow2}(k) - (x + 1)$	
$-^{\text{N}}(k, x)$	$\rightarrow (\text{pow2}(k) - x) \bmod \text{pow2}(k)$	
$\ll^{\text{N}}(k, x, y)$	$\rightarrow (x \cdot \text{pow2}(y)) \bmod \text{pow2}(k)$	
$\gg^{\text{N}}(k, x, y)$	$\rightarrow (x \text{ div } \text{pow2}(y)) \bmod \text{pow2}(k)$	
$\circ^{\text{N}}(k, x, y)$	$\rightarrow x \cdot \text{pow2}(k) + y$	
$\bowtie_u^{\text{N}}(k, x, y)$	$\rightarrow x \bowtie y$	$\bowtie \in \{<, \leq, >, \geq\}$
$\bowtie_s^{\text{N}}(k, x, y)$	$\rightarrow \text{uts}_k(x) \bowtie \text{uts}_k(y)$	$\bowtie \in \{<, \leq, >, \geq\}$
e	$\rightarrow e$	otherwise

Fig. 1. Translation \mathcal{T}_A for parametric bit-vector formulas, parametrized by axiomatization mode A . We use $\text{uts}_k(x)$ as shorthand for $2 \cdot (x \bmod \text{pow2}(k - 1)) - x$.

in ω may contain arbitrary Σ_{IA} -terms. In practice, our translation handles only cases where these terms contain symbols supported by the SMT solver, as well as terms of the form 2^t , which we assume are replaced by $\text{pow2}(t)$ during this translation. For instance, if $\omega^b(z) = w + v$ and $\omega^N(z) = 2^w - 1$, then $\text{CONV}(z)$ returns $(\text{pow2}(w) - 1) \bmod \text{pow2}(w + v)$. Equalities are processed by recursively running the translation on both sides. The next case handles symbols from the signature Σ_{BV} , where symbols f^{BV} are replaced with the corresponding uninterpreted function f^{N} . We take as the first argument $\omega^b(t_n)$, indicating the symbolic bit-width of the last argument of e , and recursively call CONV on t_1, \dots, t_n . In all cases, $\omega^b(t_n)$ corresponds to the bit-width that the uninterpreted function f^{N} expects based on its intended semantics (the bit-width of the second argument for bit-vector concatenation, or of an arbitrary argument for all other functions and predicates). Finally, if the top symbol of e is a Boolean connective we apply the conversion function recursively to all its children.

We run ELIM for all applications of uninterpreted functions f^{N} introduced during the conversion, which eliminates functions that correspond to a majority of the bit-vector operators. These functions can be equivalently expressed using integer arithmetic and pow2 . The ternary addition operation $+^{\text{N}}$, that represents

addition of two bit-vectors with their width k specified as the first argument, is translated to integer addition modulo $\text{pow2}(k)$. Similar considerations are applied for $-^{\mathbb{N}}$ and $\cdot^{\mathbb{N}}$. For $\text{div}^{\mathbb{N}}$ and $\text{mod}^{\mathbb{N}}$, our translation handles the special case where the second argument is zero, where the return value in this case is the maximum value for the given bit-width, i.e. $\text{pow2}(k) - 1$. The integer operators corresponding to unary (arithmetic) negation and bit-wise negation can be eliminated in a straightforward way. The semantics of various bitwise shift operators can be defined arithmetically using division and multiplication with $\text{pow2}(k)$. Concatenation can be eliminated by multiplying its first argument x by $\text{pow2}(k)$, where recall k is the bit-width of the second argument y . In other words, it has the effect of shifting x left by k bits, as expected. The unsigned relation symbols can be directly converted to the corresponding integer relation. For the elimination of signed relation symbols we use an auxiliary helper `uts` (unsigned to signed), defined in Fig. 1, which returns the interpretation of its argument when seen as a signed value. The definition of `uts` can be derived based on the semantics for signed and unsigned bit-vector values in the SMT LIB standard. Based on this definition, we have that integers v and u that encode bit-vectors of bit-width k satisfy $<_s^{\mathbb{N}}(k, u, v)$ if and only if $\text{uts}_k(u) < \text{uts}_k(v)$.

As an example of our translation, let $\varphi = (x + \text{BV}x) + \text{BV}z_1 \not\approx z_0, \omega^{\mathbb{N}}(z_0) = 0, \omega^{\mathbb{N}}(z_1) = 1, \text{ and } \omega^b(x) = \omega^b(z_0) = \omega^b(z_1) = a$ from Example 1. $\text{CONV}(\varphi, (\omega^b, \omega^{\mathbb{N}}))$ is $\text{ELIM}(+\mathbb{N}(a, \text{ELIM}(+\mathbb{N}(a, \chi(x), \chi(x))), 1 \bmod \text{pow2}(a))) \not\approx 0 \bmod \text{pow2}(a)$. After applying `ELIM` and simplifying, we get $(\chi(x) + \chi(x) + 1) \bmod \text{pow2}(a) \not\approx 0$.

Thanks to `ELIM`, we can assume that all formulas generated by `CONV` contain only uninterpreted function symbols in the set $\{\text{pow2}, \&^{\mathbb{N}}, |\mathbb{N}, \oplus^{\mathbb{N}}\}$. Thus, we restrict our attention to these symbols only in our axiomatization AX_A , described next.

Table 2. Full axiomatization of pow2 , $\&^{\mathbb{N}}$, and $\oplus^{\mathbb{N}}$. The axiomatization of $|\mathbb{N}$ is omitted, and is dual to that of $\&^{\mathbb{N}}$. We use $\text{ex}_i(x)$ for $(x \text{ div } \text{pow2}(i)) \bmod 2$.

\diamond	$\text{AX}_{\text{full}}^{\diamond}$
pow2	$\text{pow2}(0) \approx 1 \wedge \forall k. k > 0 \Rightarrow \text{pow2}(k) \approx 2 \cdot \text{pow2}(k - 1)$
$\&^{\mathbb{N}}$	$\forall k, x, y. \&^{\mathbb{N}}(k, x, y) \approx$ $\text{ite}(k > 1, \&^{\mathbb{N}}(k - 1, x \bmod \text{pow2}(k - 1), y \bmod \text{pow2}(k - 1)), 0) +$ $\text{pow2}(k - 1) \cdot \min(\text{ex}_{k-1}(x), \text{ex}_{k-1}(y))$
$\oplus^{\mathbb{N}}$	$\forall k, x, y. \oplus^{\mathbb{N}}(k, x, y) \approx$ $\text{ite}(k > 1, \oplus^{\mathbb{N}}(k - 1, x \bmod \text{pow2}(k - 1), y \bmod \text{pow2}(k - 1)), 0) +$ $\text{pow2}(k - 1) \cdot \text{ex}_{k-1}(x) - \text{ex}_{k-1}(y) $

Axiomatization Modes. We consider four different axiomatization modes A , which we call full, partial, combined, and `qf` (quantifier-free). For each of these axiomatizations, we define $\text{AX}_A(\varphi, \omega)$ as the conjunction:

Table 3. Partial axiomatization of pow2 , $\&^{\mathbb{N}}$, and $\oplus^{\mathbb{N}}$. The axioms for $|\mathbb{N}$ are omitted, and are dual to those for $\&^{\mathbb{N}}$. We use $\text{max}_k^{\mathbb{N}}$ for $\text{pow2}(k) - 1$.

\diamond	axiom	$\text{AX}_{\text{partial}}^{\diamond}$
pow2	base cases	$\text{pow2}(0) \approx 1 \wedge \text{pow2}(1) \approx 2 \wedge \text{pow2}(2) \approx 4 \wedge \text{pow2}(3) \approx 8$
	weak monotonicity	$\forall i \forall j. i \leq j \Rightarrow \text{pow2}(i) \leq \text{pow2}(j)$
	strong monotonicity	$\forall i \forall j. i < j \Rightarrow \text{pow2}(i) < \text{pow2}(j)$
	modularity	$\forall i \forall j \forall x. (x \cdot \text{pow2}(i)) \bmod \text{pow2}(j) \not\approx 0 \Rightarrow i < j$
	never even	$\forall i \forall x. \text{pow2}(i) - 1 \not\approx 2 \cdot x$
	always positive	$\forall i. \text{pow2}(i) \geq 1$
	div 0	$\forall i. i \text{ div } \text{pow2}(i) \approx 0$
$\&^{\mathbb{N}}$	base case	$\forall x \forall y. \&^{\mathbb{N}}(1, x, y) \approx \min(\text{ex}_0(x), \text{ex}_0(y))$
	max	$\forall k \forall x. \&^{\mathbb{N}}(k, x, \text{max}_k^{\mathbb{N}}) \approx x$
	min	$\forall k \forall x. \&^{\mathbb{N}}(k, x, 0) \approx 0$
	idempotence	$\forall k \forall x. \&^{\mathbb{N}}(k, x, x) \approx x$
	contradiction	$\forall k \forall x. \&^{\mathbb{N}}(k, x, \sim^{\mathbb{N}}(k, x)) \approx 0$
	symmetry	$\forall k \forall x \forall y. \&^{\mathbb{N}}(k, x, y) \approx \&^{\mathbb{N}}(k, y, x)$
	difference	$\forall k \forall x \forall y \forall z. x \not\approx y \Rightarrow \&^{\mathbb{N}}(k, x, z) \not\approx y \vee \&^{\mathbb{N}}(k, y, z) \not\approx x$
	range	$\forall k \forall x \forall y. 0 \leq \&^{\mathbb{N}}(k, x, y) \leq \min(x, y)$
	$\oplus^{\mathbb{N}}$	base case
zero		$\forall k \forall x. \oplus^{\mathbb{N}}(k, x, x) \approx 0$
one		$\forall k \forall x. \oplus^{\mathbb{N}}(k, x, \sim^{\mathbb{N}}(k, x)) \approx \text{max}_k^{\mathbb{N}}$
symmetry		$\forall k \forall x \forall y. \oplus^{\mathbb{N}}(k, x, y) \approx \oplus^{\mathbb{N}}(k, y, x)$
range		$\forall k \forall x \forall y. 0 \leq \oplus^{\mathbb{N}}(k, x, y) \leq \text{max}_k^{\mathbb{N}}$

$$\bigwedge_{x \in \text{FV}(\varphi)} 0 \leq \chi(x) < \text{pow2}(\omega^b(x)) \wedge \left(\bigwedge_{w \in \text{FV}(\omega)} w > 0 \right) \wedge \text{AX}_A^{\text{pow2}} \wedge \text{AX}_A^{\&^{\mathbb{N}}} \wedge \text{AX}_A^{|\mathbb{N}} \wedge \text{AX}_A^{\oplus^{\mathbb{N}}}$$

The first conjunction states that all integer variables introduced for parametric bit-vector variables x reside in the range specified by their bit-width. The second conjunction states that all free variables in ω (denoting bit-widths) are positive. The remaining four conjuncts denote the axiomatizations for the four uninterpreted functions that may occur in the output of the conversion function. The definitions of these formulas are given in Tables 2 and 3 for full and partial respectively. For each axiom, i, j, k denote bit-widths and x, y denote integers that encode bit-vectors of size k . We assume guards on all quantified formulas (omitted for brevity) that constrain i, j, k to be positive and x, y to be in the range $\{0, \dots, \text{pow2}(k) - 1\}$. Each table entry lists a set of formulas (interpreted conjunctively) that state properties about the intended semantics of these operators. The formulas for axiomatization mode full assert the intended semantics of these operators, whereas those for partial assert several properties of them. Mode combined asserts both, and mode qf takes only the formulas in partial that are quantifier-free. In particular, $\text{AX}_A^{\text{pow2}}$ corresponds to the base cases listed in partial, and $\text{AX}_{\text{qf}}^{\diamond}$ for the other operators is simply \top . The partial axiomatization of these operations mainly includes natural properties of them. For example, we include some base cases for each operation, and also the ranges of its inputs and output. For some proofs, these are sufficient. For $\&^{\mathbb{N}}$, $|\mathbb{N}$ and $\oplus^{\mathbb{N}}$, we also included their behavior for specific cases, e.g., $\&^{\mathbb{N}}(k, a, 0) = 0$ and its

variants. Other axioms (e.g., “never even”) were added after analyzing specific benchmarks to identify sufficient axioms for their proofs.

Our translation satisfies the following key properties.

Theorem 2. *Let φ be a parametric bit-vector formula that is well sorted under ω and has no occurrences of bit-vector extract or concrete bit-vector constants. Then:*

1. φ is T_{BV} -valid under ω if and only if $\mathcal{T}_{\text{full}}(\varphi, \omega)$ is T_{UFIA} -valid.
2. φ is T_{BV} -valid under ω if and only if $\mathcal{T}_{\text{combined}}(\varphi, \omega)$ is T_{UFIA} -valid.
3. φ is T_{BV} -valid under ω if $\mathcal{T}_{\text{partial}}(\varphi, \omega)$ is T_{UFIA} -valid.
4. φ is T_{BV} -valid under ω if $\mathcal{T}_{\text{qf}}(\varphi, \omega)$ is T_{UFIA} -valid.¹

The proof of Property 1 is carried out by translating every interpretation \mathcal{I}_{BV} of T_{BV} into a corresponding interpretation \mathcal{I}_{N} of T_{UFIA} such that \mathcal{I}_{BV} satisfies φ iff \mathcal{I}_{N} satisfies $\mathcal{T}_{\text{full}}(\varphi)$. The converse translation can be achieved similarly, where appropriate bit-widths are determined by the range axioms $0 \leq \chi(x) < \text{pow}2(\omega^b(x))$ that occur in $\mathcal{T}_{\text{full}}(\varphi, \omega)$. The rest of the properties follow from Property 1, by showing that the axioms in Table 3 are valid in every interpretation of T_{UFIA} that satisfies $\text{AX}_{\text{full}}(\varphi, \omega)$.

4 Case Studies

We apply the techniques from Sect. 3 to three case studies: (i) verification of invertibility conditions from Niemetz et al. [19]; (ii) verification of compiler optimizations as generated by Alive [17]; and (iii) verification of rewrite rules that are used in SMT solvers. For these case studies, we consider a set of verification conditions that originally use fixed-size bit-vectors, and exclude formulas involving multiple bit-widths.

For each formula ϕ , we first extract a parametric version φ by replacing each variable in ϕ by a fresh $x \in X^*$ and each (concrete) bit-vector constant by a fresh $z \in Z^*$. We define $\omega^b(x) = \omega^b(z) = k$ for a fresh integer variable k , and let $\omega^N(z)$ be the integer value corresponding to the bit-vector constant it replaced. Notice that, although omitted from the presentation, our translation can be easily extended to handle quantified bit-vector formulas, which appear in some of the case studies. We then define $\omega = (\omega^b, \omega^N)$ and invoke our translation from Sect. 3 on the parametric bit-vector formula φ . If the resulting formula is valid, the original verification condition holds independent of the original bit-width. In each case study, we report on the success rates of determining the validity of these formulas for axiomatization modes full, partial, combined, and qf. Overall, axiomatization mode combined yields the best results.

All experiments described below require tools with support for the SMT logic UFNIA. We used all three participants in the UFNIA division of the 2018 SMT

¹ A detailed proof, along with further details that were omitted from this paper can be found in its extended version at <https://arxiv.org/abs/1905.10434>.

competition: CVC4 [2] (GitHub master 6eb492f6), Z3 [8] (version 4.8.4), and Vampire [13] (GitHub master d0ea236). Z3 and CVC4 use various strategies and techniques for quantifier instantiation including E-matching [18], and enumerative [24] and conflict-based [27] instantiation. For non-linear integer arithmetic, CVC4 uses an approach based on incremental linearization [6, 7, 26]. Vampire is a superposition-based theorem prover for first-order logic based on the AVATAR framework [31], which has been extended also to support some theories including integer arithmetic [23]. We performed all experiments on a cluster with Intel Xeon E5-2637 CPUs with 3.5 GHz and 32 GB of memory and used a time limit of 300 s (wallclock) and a memory limit of 4 GB for each solver/benchmark pair. We consider a bit-width independent property to be proved if at least one solver proved it for at least one of the axiomatization modes.²

4.1 Verifying Invertibility Conditions

Niemetz et al. [19] present a technique for solving quantified bit-vector formulas that utilizes *invertibility conditions* to generate symbolic instantiations. Intuitively, an invertibility condition ϕ_c for a literal $\ell[x]$ is the exact condition under which $\ell[x]$ has a solution for x , i.e., $\phi_c \Leftrightarrow \exists x.\ell[x]$. For example, consider bit-vector literal $x \&^{\text{BV}}s \approx t$ with $x \notin \text{FV}(s) \cup \text{FV}(t)$; then, the invertibility condition for x is $t \&^{\text{BV}}s \approx t$.

The authors define invertibility conditions for a representative set of literals having a single occurrence of x , that involve the bit-vector operators listed in Table 1, excluding extraction, as the invertibility condition for the latter is trivially \top . A considerable number of these conditions were determined by leveraging syntax-guided synthesis (SyGuS) techniques [1]. The authors further verified the correctness of all conditions for bit-widths 1 to 65. However, a bit-width-independent formal proof of correctness of these conditions was left to future work. In the following, we apply the techniques of Sect. 3 to tackle this problem. Note that for this case study, we exclude operators involving multiple bit-widths, namely bit-vector extraction and concatenation. For the former, all invertibility conditions are \top , and for the latter a hand-written proof of the correctness of its invertibility conditions can be achieved easily.

Proving Invertibility Conditions. Let $\ell[x]$ be a bit-vector literal of the form $\diamond x \boxtimes t$ or $x \diamond s \boxtimes t$ (dually, $s \diamond x \boxtimes t$) with operators \diamond and relations \boxtimes as defined in Table 1. To prove the correctness of an invertibility condition ϕ_c for x independent of the bit-width, we have to prove the validity of the formula:

$$\phi_c \Leftrightarrow \exists x.\ell[x] \tag{1}$$

where occurrences of s and t are implicitly universally quantified. We then want to prove that Eq. 1 is T_{BV} -valid under ω . Considering the two directions of (1) separately, we get:

² All benchmarks, results, log files, and solver configurations are available at <http://cvc4.cs.stanford.edu/papers/CADE2019-BVPROOF/>.

$$\exists x.\ell[x, s, t] \Rightarrow \phi_c[s, t] \quad (\text{rtl})$$

$$\phi_c[s, t] \Rightarrow \exists x.\ell[x, s, t] \quad (\text{ltr})$$

The validity of (rtl) is equivalent to the unsatisfiability of the quantifier-free formula:

$$\ell[x, s, t] \wedge \neg\phi_c[s, t] \quad (\text{rtl}')$$

Eliminating the quantifier in (ltr) is much trickier. It typically amounts to finding a symbolic value for x such that $\ell[x, s, t]$ holds provided that $\phi_c[s, t]$ holds. We refer to such a symbolic value as a *conditional inverse*.

Conditional Inverses. Given an invertibility condition ϕ_c for x in bit-vector literal $\ell[x]$, we say that a term α_c is a *conditional inverse* for x if $\phi_c \Rightarrow \ell[\alpha_c]$ is T_{BV} -valid. For example, the term s itself is a conditional inverse for x in the literal $(x \mid^{\text{BV}} s) \leq_u^{\text{BV}} t$: given that there exists some x such that $(x \mid^{\text{BV}} s) \leq_u^{\text{BV}} t$, we have that $(s \mid^{\text{BV}} s) \leq_u^{\text{BV}} t$. When a conditional inverse α_c for x is found, we may replace (ltr) by:

$$\phi_c \Rightarrow \ell[\alpha_c] \quad (\text{ltr}')$$

Clearly, (ltr') implies (ltr). However, the converse may not hold, i.e., if (ltr') is refuted, (ltr) is not necessarily refuted. Notice that if the invertibility condition for x is \top , the conditional inverse is in fact unconditional. The problem of finding a conditional inverse for a bit-vector literal $x \diamond s \bowtie t$ (dually, $s \diamond x \bowtie t$) can be defined as a SyGuS problem by asking whether there exists a binary bit-vector function C such that the (second-order) formula $\exists C \forall s \forall t. \phi_c \Rightarrow C(s, t) \diamond s \bowtie t$ is satisfiable. If such a function C is found, then it is in fact a conditional inverse for x in $\ell[x]$. We synthesized conditional inverses for x in $\ell[x]$ for bit-width 4 with variants of the grammars used in [19] to synthesize invertibility conditions. For each grammar we generated 160 SyGuS problems, one for each combination of bit-vector operator and relation from Table 1 (excluding extraction and concatenation), counting commutative cases only once. We used the SyGuS feature of the SMT solver CVC4 [25] to solve these problems, and out of 160, we were able to synthesize candidate conditional inverses for 143 invertibility conditions. For 12 out of these 143, we found that the synthesized terms were not conditional inverses for every bit-width, by checking (ltr') for bit-widths up to 64.

Results. Table 4 provides detailed information on the results for the axiomatization modes full, partial, and qf discussed in Sect. 3. We use \rightarrow and \leftarrow to indicate that only direction left-to-right ((ltr) or (ltr')) or right-to-left (rtl'), respectively, were proved, and \checkmark and \times to indicate that both or none, respectively, of the directions were proved. Additionally, we use \rightarrow_{α_c} (resp. $\rightarrow_{\text{no } \alpha_c}$) to

indicate that for direction left-to-right, formula (ltr') (resp. (ltr)) was proved with (resp. without) plugging in a conditional inverse.

Overall, out of 160 invertibility conditions, we were able to fully prove 110, and for 19 (17) conditions we were able to prove only direction (rtl') (ltr'). For direction right-to-left, 129 formulas (rtl') overall were successfully proved to be unsatisfiable. Out of these 129, 32 formulas were actually trivial since the invertibility condition ϕ_c was \top . For direction left-to-right, overall, 127 formulas were proved successfully, and out of these, 102 (94) were proved using (resp. not using) a conditional inverse. Furthermore, 33 formulas could only be proved when using a conditional inverse. Thus, using conditional inverses was helpful for proving the correctness of invertibility conditions.

Considering the different axiomatization modes, overall, with 104 fully proved and only 17 unproved instances, combined performed best. Interestingly, even though axiomatization qf only includes some of the base cases of axiomatization partial, it still performs well. This may be due to the fact that in many cases, the correctness of the invertibility condition does not rely on any particular property of the operators involved. For example, the invertibility condition ϕ_c for literal $x \&^{\text{BV}} s \approx t$ is $t \&^{\text{BV}} s \approx t$. Proving the correctness of ϕ_c amounts to coming up with the right substitution for x , without relying on any particular axiomatization of $\&^{\text{N}}$. In contrast, the invertibility condition ϕ_c for literal $x \&^{\text{BV}} s \not\approx t$ is $t \not\approx 0 \vee s \not\approx 0$. Proving the correctness of ϕ_c relies on axioms regarding $\&^{\text{BV}}$ and \sim^{BV} . Specifically, we have found that from partial, it suffices to keep “min” and “idempotence” to prove ϕ_c . Overall, from the 2696 problems that this case study included, CVC4 proved 50.3%, Vampire proved 31.4%, and Z3 proved 33.8%, while 23.5% of the problems were proved by all solvers.

Table 4. Invertibility condition verification using axiomatization modes combined, full, partial, and qf. Column $\rightarrow_{\alpha_c}(\rightarrow_{\text{no } \alpha_c})$ counts left-to-right proved with (without) conditional inverse.

Axiomatization	✓	←	→	✗	\rightarrow_{α_c}	$\rightarrow_{\text{no } \alpha_c}$
full	64	18	22	56	72	51
partial	76	14	26	44	78	81
qf	40	22	22	76	50	51
combined	104	21	18	17	99	79
Total (160)	110	19	17	14	102	94

4.2 Verifying Alive Optimizations

Lopes et al. [17] introduces Alive, a tool for proving the correctness of compiler peephole optimizations. Alive has a high-level language for specifying optimizations. The tool takes as input a description of an optimization in this high-level language and then automatically verifies that applying the optimization to an arbitrary piece of source code produces optimized target code that is equivalent

under a given precondition. It can also automatically translate verified optimizations into C++ code that can be linked into LLVM [16]. For each optimization, Alive generates four constraints that encode the following properties, assuming that the precondition of the optimization holds:

1. *Memory* Source and Target yield the same state of memory after execution.
2. *Definedness* The target is well-defined whenever the source is.
3. *Poison* The target produces so-called poison values (caused by LLVM’s *nsw*, *nw*, and *exact* attributes) only when the source does.
4. *Equivalence* Source and target yield the same result after execution.

From these verification tasks, Alive can generate benchmarks in SMT-LIB 2 format in the theory of fixed-size bit-vectors, with and without quantifiers. For each task, types are instantiated with all possible valid type assignments (for integer types up to a default bound of 64 bits). In the following, we apply our techniques from Sect. 3 to prove Alive verification tasks independently from the bit-width. For this, as in the Alive paper, we consider the set of optimizations from the *instcombine* optimization pass of LLVM, provided as Alive translations (433 total).³ Of these 433 optimizations, 113 are dependent on a specific bit-width; thus we focus on the remaining 320. We further exclude optimizations that do not comply with the following criteria:

- In each generated SMT-LIB 2 file, only a single bit-width is used.
- All SMT-LIB 2 files generated for a property (instantiated for all possible valid type assignments) must be identical modulo the bit-width (excluding, e.g., bit-width dependent constants other than 0, 1, (un)signed min/max, and the bit-width).

As a useful exception to the first criterion, we included instances where all terms of bit-width 1 can be interpreted as Boolean terms. Overall, we consider bit-width independent verification conditions 1–4 for 180 out of 320 optimizations. None of these include memory operations or poison values, and only some have definedness constraints (and those are simple). Hence, the generated verification conditions 1–3 are trivial. We thus only consider the equivalence verification conditions for these 180 optimizations.

Results. Table 5 summarizes the results of verifying the equivalence constraints for the selected 180 optimizations from the *instcombine* LLVM optimization pass. It first lists all families, showing the number of bit-width independent optimizations per family (320 total). The next column indicates how many in each family were in the set of 180 considered optimizations, and the remaining columns show how many of those considered were proved with each axiomatization mode.

³ At <https://github.com/nunoplopes/alive/tree/master/tests/instcombine>.

Table 5. Alive optimizations verification using axiomatizations combined, full, partial and qf.

Family	Considered	Proved				
		full	partial	qf	combined	Total
AddSub (52)	16	7	7	7	9	9
MulDivRem (29)	5	1	2	1	3	3
AndOrXor (162)	124	57	55	53	60	60
Select (51)	26	15	11	11	16	16
Shifts (17)	9	0	0	0	0	0
LoadStoreAlloca (9)	0	0	0	0	0	0
Total (320)	180	80	75	72	88	88

Overall, out of 180 equivalence verification conditions, we were able to prove 88. Our techniques were most successful for the AndOrXor family. This is not too surprising, since many verification conditions of this family require only Boolean reasoning and basic properties of ordering relations that are already included in the theory T_{IA} . For example, given bit-vector term a and bit-vector constants C_1 and C_2 , optimization AndOrXor:979 essentially rewrites $(a <_s^{BV} C_1 \wedge a <_s^{BV} C_2)$ to $a <_s^{BV} C_1$, provided that precondition $C_1 <_s^{BV} C_2$ holds. To prove its correctness, it suffices to apply the transitivity of $<_s^{BV}$ with Boolean reasoning. The same holds when lifting this equivalence to the integers, deducing the transitivity of $<_s^N$ from that of the builtin $<$ relation of T_{IA} .

None of the 9 benchmarks from the Shifts family were proven. These benchmarks are more complicated than others. They combine bit-wise and arithmetical operations and thus rely on their axiomatization. Solving these benchmarks is an interesting challenge for future work. Adding specialized axioms to partial is one promising approach.

Interestingly, for this case study, the results from the different axiomatization modes are very similar. This can again be explained by the fact that many optimizations rely on properties of the integers that are already included in T_{IA} , without requiring any particular property of functions pow2 , $\&^N$, $|^N$ and \oplus^N (as in the above example).

Note that we have also tried using our approach for proving the equivalence verification conditions for up to a bit-width of 64. However, all optimizations that were proven correct this way were already proven correct for arbitrary bit-widths, which suggests that this restriction did not make the benchmarks easier. Overall, from the 720 problems in this case study, CVC4 proved 42.6%, Vampire proved 36.2%, and Z3 proved 37.9%, while 32.5% of the problems were proved by all solvers.

4.3 BV Rewriting

SMT solvers for the theory of fixed-size bit-vectors heavily rely on rewriting to reduce the size of the input formula prior to solving the problem. Since these rewrite rules are usually implemented independently of the bit-width, verifying that they hold for any bit-width is crucial for the soundness of the solver. For this case study, we used a feature of the SyGuS solver in CVC4 that allows us to enumerate equivalent bit-vector terms/formulas (rewrite candidates) for a certain bit-width up to a certain term depth (nesting level of operators) [21]. We generated 1575 pairs of equivalent bit-vector terms of depth three and 431 equivalent pairs of formulas of depth two for bit-width 4 and translated them to integer problems with axiomatization modes full, partial, qf, and combined, resulting in $6300 + 1724 = 8024$ benchmarks in total. Since rewrites that have been proved can be used to further axiomatize the integer translation, we collected all proven rewrites after each run, added them as axioms to the initial problems and reran the experiments. This was repeated until we reached a fixpoint, i.e., no further rewrites were proved. With this approach, we were able to prove 409 out of the 435 formula equivalences (94%), reaching a fixpoint at the first iteration. For the equivalent terms, we initially proved 878 out of the 1575 equivalences, which increased to 935 (59%) after adding all axioms from the first run, reaching a fixpoint after two iterations. Overall, from the 8024 problems, CVC4 proved 64.2%, Vampire proved 66.5%, and Z3 proved 64.2%, while 63.8% of the problems were proved by all solvers.

5 Conclusion and Further Research

We have studied several translations from bit-vector formulas with parametric bit-width to the theories of integer arithmetic and uninterpreted functions. The translations differ in the way that the operator $2^{(-)}$ and bitwise logical operators are axiomatized, namely, fully (using induction) or partially (using some of their key properties). Our empirical results show that state-of-the-art SMT solvers are capable of solving the translated formulas for various benchmarks that originate from the verification of invertibility conditions, LLVM optimizations, and rewriting rules for fixed-size bit-vectors.

In future research, we plan to investigate a translation of our results to a proof assistant such as Coq, for which a bit-vector library was recently developed [9]. This will involve supporting proofs in the SMT solver for non-linear arithmetic and quantifiers. We believe that our promising experimental results with an integer encoding indicate that this is a viable approach for automating bit-width independent proofs. We also plan to explore satisfiable benchmarks, and to extend our approach for translating models.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 1–8 (2013)
2. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, UK (2010)
4. Bjørner, N.S., Pichora, M.C.: Deciding fixed and non-fixed size bit-vectors. In: Steffen, B. (ed.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 376–392. Springer, Berlin (1998). <https://doi.org/10.1007/BFb0054184>
5. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. *J. Autom. Reasoning* **51**(1), 109–128 (2013). <https://doi.org/10.1007/s10817-013-9278-5>
6. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Experimenting on solving nonlinear integer arithmetic with incremental linearization. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 383–398. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_23
7. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. *ACM Trans. Comput. Log.* **19**(3), 19:1–19:52 (2018)
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24. <http://dl.acm.org/citation.cfm?id=1792734.1792766>
9. Ekici, B., et al.: SMTCoq: a plug-in for integrating smt solvers into Coq. In: Majumdar, R., Kuncak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 126–133. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_7
10. Enderton, H., Enderton, H.B.: A Mathematical Introduction to logic. Elsevier, Amsterdam (2001)
11. Gupta, A., Fisher, A.L.: Parametric circuit representation using inductive boolean functions. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 15–28. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7_3
12. Gupta, A., Fisher, A.L.: Representation and symbolic manipulation of linearly inductive boolean functions. In: Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design, pp. 192–199, ICCAD 1993. IEEE Computer Society Press, Los Alamitos (1993). <http://dl.acm.org.stanford.idm.oclc.org/citation.cfm?id=259794.259827>
13. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
14. Kovásznai, G., Fröhlich, A., Biere, A.: Complexity of fixed-size bit-vector logics. *Theory Comput. Syst.* **59**(2), 323–376 (2016). <https://doi.org/10.1007/s00224-015-9653-1>
15. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Texts in Theoretical Computer Science. An EATCS Series, 2nd edn. Springer, Berlin (2016)

16. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004), 20–24 March 2004, San Jose, CA, USA, pp. 75–88. IEEE Computer Society (2004). <https://doi.org/10.1109/CGO.2004.1281665>
17. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 22–32, PLDI 2015. ACM, New York (2015). <https://doi.org/10.1145/2737924.2737965>
18. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
19. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving Quantified Bit-Vectors Using Invertibility Conditions. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 236–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_16
20. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
21. Nötzli, A., et al.: Syntax-guided rewrite rule enumeration for SMT solvers. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_20
22. Pichora, M.C.: Automated reasoning about hardware data types using bit-vectors of symbolic lengths. Ph.D. thesis, Toronto, ON, Canada (2003). aAINQ84686
23. Reger, G., Suda, M., Voronkov, A.: Unification with abstraction and theory instantiation in saturation-based reasoning. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 3–22. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_1
24. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 112–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_7
25. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_12
26. Reynolds, A., Tinelli, C., Jovanović, D., Barrett, C.: Designing theory solvers with extensions. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 22–40. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66167-4_2
27. Reynolds, A., Tinelli, C., de Moura, L.M.: Finding conflicting instances of quantified formulas in SMT. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 21–24 October 2014, pp. 195–202 (2014). <https://doi.org/10.1109/FMCAD.2014.6987613>
28. Solidity Language Developers: Solidity (2018). <https://solidity.readthedocs.io/en/v0.4.25/>
29. TC Development team: The Coq proof assistant reference manual version 8.9 (2019). <https://coq.inria.fr/distrib/current/refman/>
30. Tinelli, C., Zarba, C.G.: Combining decision procedures for sorted theories. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 641–653. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30227-8_53
31. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 696–710. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_46