Verifying Bit-vector Invertibility Conditions in Coq

Burak Ekici
University of Innsbruck
Innsbruck, Austria
burak.ekici@uibk.ac.at

Arjun Viswanathan
University of Iowa
Iowa City, USA
arjun-viswanathan@uiowa.edu

Yoni Zohar
Stanford University
Stanford, USA
yoniz@cs.stanford.edu

Clark Barrett
Stanford University
Stanford, USA
barrett@cs.stanford.edu

Cesare Tinelli
University of Iowa
Iowa City, USA
cesare-tinelli@uiowa.edu

This work is a part of an ongoing effort to prove the correctness of invertibility conditions for the theory of fixed-width bit-vectors, that are used to solve quantified bit-vector formulas in the Satisfiability Modulo Theories (SMT) solver CVC4. While many of these were proved in a completely automatic fashion for any bit-width, some were only proved for bit-widths up to 65, even though they are being used to solve formulas over arbitrary bit-widths. In this paper we describe our initial efforts in proving a subset of these invertibility conditions in the Coq proof assistant. We describe the Coq library that we use, as well as the extensions that we introduced to it.

1 Introduction

Reasoning logically about bit-vectors is useful for many applications in hardware and software verification. While Satisfiability Modulo Theories (SMT) solvers are able to reason about bit-vectors of fixed width, they currently require all widths to be expressed concretely (by a numeral) in their input formulas. For this reason, they cannot be used to prove properties of bit-vector operators that are parametric in the bit-width such as, for instance, the associativity of bit-vector concatenation. Proof assistants such as Coq [12], that have direct support for dependent-types are better suited for such tasks.

Bit-vector formulas that are parametric in the bit-width, arise in the verification of parametric Boolean functions and circuits (see, e.g., [7]). In our case, we are mainly interested in parametric lemmas that are relevant to internal techniques of SMT-solvers for the theory of fixed-width bit-vectors. Such techniques are developed a priori for every possible bit-width, even though they are applied on a particular bit-width. Meta-reasoning about the correctness of such solvers then requires bit-width independent reasoning.

An example of the latter kind, which is the focus of the current paper, is the notion of *invertibility conditions* [8] as a basis for a quantifier-instantiation technique to reason about the satisfiability of quantified bit-vector formulas. For a trivial case of an invertibility condition consider the equation $x+s=t$ where x, s and t are variables of the same bit-vector sort. In the terminology of [8], this equation is "invertible" for x, i.e., solvable for x, for any value of s and t. A general solution is represented by the term $t-s$. Since the solution is unconditional the invertibility condition for $x+s=t$ is simply the universally true formula $\top$. The formula stating this fact, referred to here as an *invertibility equivalence*, is $\top \Leftrightarrow \exists x.\, x+s=t$, a valid formula in the theory of fixed-sized bit-vectors for any bit-width n for x, s and t. In contrast, the equation $x \cdot s = t$ is not always invertible for x. A necessary and sufficient condition for invertibility is $(-s \mid s)\ \&\ t = t$ meaning that the invertibility equivalence $(-s \mid s)\ \&\ t = t \Leftrightarrow \exists x.\, x \cdot s = t$ is valid for any bit-width n for x, s and t [8]. Niemetz et al. [8] provide a total of 162 invertibility conditions for several bit-vector operators for both equations and inequations. However, they were able to verify,

Submitted to:
PxTP 2019

using SMT solvers, the corresponding invertibility equivalences only for concrete bit-widths up to 65, given the reasoning limitations of SMT solvers mentioned earlier. A recent paper by Niemetz et al. [9] addresses this challenge by translating these invertibility equivalences into quantified formulas over the combined theory of non-linear integer arithmetic and uninterpreted functions — a theory supported by a number of SMT solvers. While partially successful, this approach failed to verify over a quarter of the invertibility equivalences.

In this work, we approach the task of verifying the invertibility equivalences proposed in [8] by proving them interactively with the Coq proof assistant. We extend a rich Coq library for bit-vectors we developed in previous work [5] with additional operators and lemmas to facilitate the task of verifying invertibility equivalences for arbitrary bit-widths, and prove a representative subset of them. Our results offer evidence that proof assistants can support automated theorem provers in meta-verification tasks.

Our Coq library models the theory of fixed-width bit vectors adopted by the SMT-LIB 2 standard [1].[1] It represents bit-vectors as lists of Booleans. The bit-vector type is dependent on a positive integer that represents the length of the list. Underneath the dependent representation is a simply typed or *raw* bit-vector type with a size function which is used to explicitly state facts on the length of the list. A functor translates an instance of a raw bit-vector along with specific information about its size into a dependently-typed bit-vector. For this work, we extended the library with the arithmetic right shift operation and the unsigned weak less-than predicate and proved 13 invertibility equivalences. We initially proved these invertibility equivalences over raw-bit-vectors and then used these proofs when proving the invertibility equivalences over dependent bit-vectors, as explained in Section 4.

The remainder of this paper is organized as follows. After some technical preliminaries in Section 2, we provide an overview of invertibility conditions for the theory of fixed-width bit-vectors in Section 3, and discuss previous attempts to verify them. Then, in Section 4, we describe the bit-vector Coq library and our current extensions to it. In Section 5, we outline how we used the extended library to prove the correctness of a representative subset of invertibility equivalences. We conclude in Section 6 with directions for future work.

2 Preliminaries

We assume the usual terminology of many-sorted first-order logic with equality (see, e.g., [6] for more details). We denote equality by $=$, and use $x \neq y$ as an abbreviation for $\neg(x = y)$. The signature Σ_{BV} of the SMT-LIB 2 theory of fixed-width bit-vectors includes a unique sort for each positive integer n, which we denote by $\sigma_{[n]}$. For every positive integer n and a bit-vector of width n, the signature includes a constant of sort $\sigma_{[n]}$ in Σ_{BV} representing that bit-vector, which we denote as a binary string of length n. The function and predicate symbols of Σ_{BV} are as described the SMT-LIB 2 standard. Formulas of Σ_{BV} are built from variables (sorted by the sorts $\sigma_{[n]}$), bit-vector constants, and the function and predicate symbols of Σ_{BV}, along with the usual logical connectives and quantifiers. We write $\psi[x_1,\ldots,x_n]$ to represent a formula whose free variables are from $\{x_1,\ldots,x_n\}$.

The semantics of Σ_{BV}-formulas is given by interpretations that extend a single many-sorted first-order structure so that the domain of every sort $\sigma_{[n]}$ is the set of bit-vectors of bit-width n, and the function and predicate symbols are interpreted as specified by the SMT-LIB 2 standard. A Σ_{BV}-formula is said to be *valid* in the theory of fixed-width bit-vectors if it evaluates to true in every such interpretation.

In what follows, we denote by Σ_0 the sub-signature of Σ_{BV} containing only the predicate symbols $<_u$, $\leq_u$, $>_u$, and $\geq_u$ (corresponding to various unsigned comparisons between bit-vectors), as well as the

[1] The SMT-LIB 2 theory is defined at http://www.smt-lib.org/theories.shtml.

function symbols + (bit-vector addition), &, | (bit-wise conjunction and disjunction), and $\ll$, $\gg$ and $\gg_a$ (left shift, and logical and arithmetical right shift). We also denote by Σ_1 the extension of Σ_0 with the predicate symbols $<_s$, $\leq_s$, $>_s$, and $\geq_s$ (corresponding to signed comparisons between bit-vectors), as well as the function symbols $-$, $\cdot$, $\div$, mod (corresponding to subtraction, multiplication, division and remainder), $\sim$, $-$ (negation operators), and $\circ$ (concatenation). We use 0 to represent the bit-vectors composed of all 0-bits. Its numerical or bit-vector interpretation should be clear from context. Using bit-wise negation $\sim$, we can express the bitvectors composed of all 1-bits by ~ 0.

3 Invertibility Conditions And Their Verification

Many applications rely on bit-precise reasoning and thus can be modeled using the SMT-LIB 2 theory of fixed-width bit-vectors. For certain applications, such as verification of safety properties for programs, quantifier-free reasoning is not enough, and the combination of bit-precise reasoning with the ability to handle quantifiers is needed. Niemetz et al. present a technique to solve quantified bit-vector formulas, which is based on *invertibility conditions* [8]. An invertibility condition for a variable x in a Σ_{BV}-literal $\ell[x,s,t]$ is a formula $IC[s,t]$ such that $\forall s.\forall t.IC[s,t] \Leftrightarrow \exists x.\ell[x,s,t]$ is valid in the theory of fixed-width bit-vectors. For example, consider the bit-vector literal $x \,\&\, s = t$ where x, s and t are distinct variables of the same sort. The invertibility condition for x in that literal is $t \,\&\, s = t$.

Niemetz et al. [8] define invertibility conditions for a representative set of literals ℓ having a single occurrence of x, that involve the bit-vector operators of Σ_1. The soundness of the technique proposed in that work relies on the correctness of the invertibility conditions. Every literal $\ell[x,s,t]$ and its corresponding invertibility condition $IC[s,t]$ induce the *invertibility equivalence*

$$IC[s,t] \Leftrightarrow \exists x.\ell[x,s,t] \tag{1}$$

The correctness of invertibility equivalences should be verified for all possible sorts for the variables x,s,t for which the condition is well sorted. More concretely, for the case where x,s,t are all of sort $\sigma_{[n]}$, say, this means that one needs to prove, for *all* $n > 0$, the validity of

$$\forall s : \sigma_{[n]}.\forall t : \sigma_{[n]}.IC[s,t] \Leftrightarrow \exists x : \sigma_{[n]}.\ell[x,s,t] \ .$$

This was done in Niemetz et al. [8] using an SMT solver but only for concrete values of n from 1 to 65. A proof of Equation (1) that is parametric in the bit-width n cannot be done with SMT solvers since they currently only support the theory of *fixed-width* bit-vectors, where Equation (1) cannot even be expressed. To overcome this limitation, a later paper by Niemetz et al. [9] suggested a translation from bit-vector formulas with *parametric* bit-widths to the theory of (non-linear) integer arithmetic with uninterpreted functions. Thanks to this translation, the authors were able to verify, with the aid of SMT solvers for the theory of integer arithmetic with uninterpreted functions, the correctness of 110 out of 162 invertibility equivalences. None of the solvers used in that work were able to prove the remaining equivalences. For those, it then seems appropriate to use a proof-assistant, as this allows for more intervention by the user who can provide crucial intermediate steps. It goes without saying that even for the 110 invertibility equivalences that were proved, the level of confidence achieved by proving them in a proof-assistant such as Coq would be greater than a verification (without a verified formal proof) by an SMT solver.

In the rest of this paper we describe our initial efforts and future plans for proving the invertibility equivalences, starting with those that were not proved in [9].

4 The Coq Bit-vector Library

In this section, we describe the Coq library we use and the extensions we developed with the goal of formalizing and proving invertibility equivalences. The original library was developed for SMTCoq [5], a Coq plugin that enables Coq to dispatch proofs to external proof-producing solvers. It is used to represent SMT-LIB 2 bit-vectors in Coq. Coq's own library of bit-vectors [4] was an alternative, but it has only definitions and no lemmas. A more suitable substitute could have been the Bedrock Bit Vectors Library [2]. We chose the SMTCoq library mainly because it was explicitly developed to represent SMT-LIB 2 bit-vectors in Coq and came with a rich set of lemmas relevant to proving the invertibility equivalences.

The Coq library contains both a simply-typed and dependently-typed theory of bit-vectors implemented as module types. The former, which we also refer to as a theory of *raw bit-vectors*, formalizes bit-vectors as Boolean lists while the latter defines a bit-vector as a Coq record, with its size as the parameter, made of two fields: a Boolean list and a coherence condition to ensure that the parameterized size is indeed the length of the given list. The library also implements a functor module from the simply-typed module to the dependently-typed module establishing a correspondence between the two theories. This way, one can first prove a bit-vector property in the context of the simply-typed theory and then map it to its corresponding dependently-typed one via the functor module. Note that while it is possible to define bit-vectors natively as a dependently-typed theory in Coq and prove their properties there, it would be cumbersome and unduly complex to do dependent pattern matching or case analysis over bit-vector instances because of the complications brought by Coq's unification algorithm (which is inherently undecidable). One can try to handle such complications as illustrated by Sozeau [11]. However, we found the two-theory approach of Ekici et al. [5] more convenient in practice for our purposes.

The library adopts the little-endian notation for bit-vectors, thus following the internal representation of bit-vectors in SMT solvers such as CVC4. This makes arithmetic operations easier to perform since the least significant bit of a bit-vector is the head of the list representing it in the *raw* theory.

Out of the 11 bit-vector operators and 10 predicates contained in Σ_1, the library had support for 8 operators and 6 predicates. The supported predicates, however, can be used to express the other 4. The predicates that were not directly supported by the library were the weak inequalities $\leq_u$, $\geq_u$, $\leq_s$, $\geq_s$ and the operations that were not supported were $\gg_a$, $\div$, and mod. We extended the library with the operator $\gg_a$ and the predicate $\leq_u$, and redefined $\ll$ and $\gg$, as explained in Section 5.

We focused on invertibility conditions for literals of the form $x \diamond s \bowtie t$ and $s \diamond x \bowtie t$, where x, s and t are variables and $\diamond$ and $\bowtie$ are respectively function and predicate symbols in Σ_0. Σ_0 was chosen as a representative set because it seemed both expressive enough and feasible for proofs in Coq. Such literals, as well as their invertibility conditions, include only operators that are supported by the library (after its extension with $\gg_a$). Whenever an invertibility condition included an occurrence of a weak unsigned inequality, we represented that within Σ_0 using logical negation and the corresponding strict inequality. While the correctness of this representation is intuitively clear, we plan to prove it in Coq in future work. We did use $\leq_u$ explicitly, however, in the proof of the invertibility equivalence for the literal $x \gg s >_u t$.

To demonstrate the intuition and various aspects of the extension of the library, we briefly describe the addition of $\leq_u$. The Coq definitions concerning $\leq_u$ are presented in Figure 1.[2] Like most other operators, $\leq_u$ is defined in several *layers*. The function `bv_ule` at the highest layer, ensures that comparisons are between bit-vectors of the same size and then calls `ule_list`. Since bit-vectors in the library are

[2]Both the library and the proofs of invertibility equivalences can be found at https://github.com/ekiciburak/bitvector/tree/pxtp2019. It compiles with `coqc-8.9.0`.

```
Fixpoint ule_list_big_endian (x y : list bool) :=
  match x, y with
  | nil, nil => true
  | nil, _ => false
  | _, nil => false
  | xi :: nil, yi :: nil => orb (eqb xi yi)
                               (andb (negb xi) yi)
  | xi :: x', yi :: y' => orb (andb (Bool.eqb xi yi) (ule_list_big_endian x' y'))
                              (andb (negb xi) yi)
  end.

Definition ule_list (x y: list bool) :=
  (ule_list_big_endian (List.rev x) (List.rev y)).

Definition bv_ule (a b : bitvector) : bool :=
  if @size a =? @size b then
    ule_list a b
  else
    false.
```

Figure 1: Definitions of $\leq_u$ in Coq.

represented in little-endian notation, and since we want to start comparing bits starting from the most significant bit, which is more convenient to access from the head of the list, `ule_list` reverses the lists and calls `ule_list_big_endian`, which we consider to be at the lowest layer of the definition. `ule_list_big_endian` then does a lexicographical comparison of the two bit-vectors, starting from the most significant bits.

To see why the addition of $\leq_u$ to the library is useful, consider, for example, the following parametric lemma, stating that ~ 0 is the largest unsigned bit-vector of its type:

$$\forall x : \sigma_{[n]}. x \leq_u \sim 0 \tag{2}$$

When not using this explicit operator, we usually rewrite it as:

$$\forall x : \sigma_{[n]}. x <_u \sim 0 \vee x = \sim 0 \tag{3}$$

In such cases, since the definitions of $<_u$ and $=$ have a similar structure to the one in Figure 1, we strip down the layers of $<_u$ and $=$ separately, whereas using $\leq_u$, we only do this once. Depending on the specific proof at hand, using $\leq_u$ is sometimes more convenient for this reason.

5 Proving Invertibility Equivalences in Coq

In this section we provide specific details about proving invertibility equivalences in Coq. In addition to the bit-vector library described in Section 4, in several proofs of invertibility equivalences we benefited from CoqHammer [3], a plug-in that aims at extending the automation in Coq by combining machine learning and automated reasoning techniques in a similar fashion to what is done in Isabelle/HOL [10]. Note that one does not need to install CoqHammer in order to build the bit-vector library, since all the proof reconstruction tactics of CoqHammer are included in it.

```
1  Theorem bvashr_ult2_rtl : forall (n : N), forall (s t : bitvector n),
2  (exists (x : bitvector n), (bv_ult (bv_ashr_a s x) t = true)) ->
3  (((bv_ult s t = true) \/ (bv_slt s (zeros n)) = false) /\
4  (bv_eq t (zeros n)) = false).
5  Proof. intros n s t H.
6          destruct H as ((x, Hx), H).
7          destruct s as (s, Hs).
8          destruct t as (t, Ht).
9          unfold bv_ult, bv_slt, bv_ashr_a, bv_eq, bv in *. cbn in *.
10         specialize (InvCond.bvashr_ult2_rtl n s t Hs Ht); intro STIC.
11         rewrite Hs, Ht in STIC. apply STIC.
12         now exists x.
13 Qed.
```

Figure 2: A proof of one direction of the invertibility equivalence for $\gg_a$ and $<_u$ using dependent-types.

The natural representation of bit-vectors in Coq is the dependently-typed representation, and therefore the invertibility equivalences are formulated using this representation. As discussed in Section 4, however, proofs in this representation are composed of proofs over simply-typed bit-vectors, which are easier to reason about. Some conversions between the different representations are then needed to lift a proof over raw bit-vectors to one over dependently-typed bit-vectors.

For example, Figure 2 includes a proof of the following direction of the invertibility equivalence for $\gg_a$ and $<_u$:

$$\forall s : \sigma_{[n]}.\forall t : \sigma_{[n]}.(\exists x : \sigma_{[n]}.s \gg_a x <_u t) \Rightarrow ((s <_u t \vee \neg(s <_s 0)) \wedge t \neq 0) \tag{4}$$

In the proof, lines 6–9 transform the dependent bit-vectors from the goal and the hypotheses into simply-typed bit-vectors. Then, lines 10–12 invoke the corresponding lemma for simply-typed bit-vectors (called `InvCond.bvashr_ult2_rtl`) along with some simplifications.

Most of the effort in this project went into proving equivalences over raw bit-vectors. As an illustration, consider the following equivalence over $\ll$ and $>_u$:

$$\forall s : \sigma_{[n]}.\forall t : \sigma_{[n]}.(t <_u \sim 0 \ll s) \Leftrightarrow (\exists x : \sigma_{[n]}.x \ll s >_u t) \tag{5}$$

The left-to-right implication is easy to prove using ~ 0 itself as the witness of the existential proof goal and considering the symmetry between $>_u$ and $<_u$. The proof of the right-to-left implication relies on the following lemma:

$$\forall x : \sigma_{[n]}.\forall s : \sigma_{[n]}.(x \ll s) \leq_u (\sim 0 \ll s) \tag{6}$$

From the right side of the equivalence in Equation (5), we get some x for which $x \ll s >_u t$ holds. Flipping the inequality, we have that $t <_u x \ll s$; using this, and transitivity over $<_u$ and $\leq_u$, the aforementioned lemma (from Equation (6)) gives us the left side of the equivalence in Equation (5).

As mentioned in Section 4, we have redefined the shift operators $\ll$ and $\gg$ in the library. This was instrumental, for example, in the proof of Equation (6). Figure 3 includes both the original and new definitions of $\ll$. The definitions of $\gg$ are similar. Originally, $\ll$ was defined using the `shl_one_bit` and the `shl_n_bits` functions. `shl_one_bit` shifts the bit-vector left by one bit and is repeatedly called by `shl_n_bits` to complete the shift. The new definition `shl_n_bits_a` uses `mk_list_false` which constructs the necessary list of 0s and appends (++ in Coq) it to the beginning of the list (since we

```
1  Definition shl_one_bit  (a: list bool) : list bool :=
2    match a with
3          | [] => []
4          | _ => false :: removelast a
5    end.
6
7  Fixpoint shl_n_bits  (a: list bool) (n: nat): list bool :=
8    match n with
9          | O => a
10         | S n' => shl_n_bits (shl_one_bit a) n'
11   end.
12
13 Definition shl_n_bits_a  (a: list bool) (n: nat): list bool :=
14   if (n <? length a)%nat then
15         mk_list_false n ++ firstn (length a - n) a
16   else
17         mk_list_false (length a).
18
19 Theorem bv_shl_eq: forall (a b : bit-vector), bv_shl a b = bv_shl_a a b.
```

Figure 3: Various definitions of $\ll$

are using the little endian notation); the bits to be shifted from the original bit-vector are retrieved using the `firstn` function, which is defined in the Coq library for lists. The `nat` type used in Figure 3 is the Coq representation of Peano natural numbers that has `O` and `S` as its two constructors — as depicted in the pattern match in lines 9 and 10. The theorem at the bottom of Figure 3 allows us to switch between the two definitions when needed. Function `bv_shl` defines the left shift operation using `shl_n_bits` whereas `bv_shl_a` does it using `shl_n_bits_a`.

The new definition uses `firstn` and `++`, over which many necessary properties are already proven in the standard library. This benefits us in manual proofs, and in calls to CoqHammer, since the latter is able to use lemmas from the imported libraries to prove the goals that are given to it. Using this representation, proving Equation (6) reduces to proving the lemmas `bv_ule_1_firstn` and `bv_ule_pre_append`, shown in Figure 4. The proof of `bv_ule_pre_append` benefited from the property `app_comm_cons` from the standard list library of Coq, while `firstn_length_le` was useful in reducing the goal of `bv_ule_1_firstn` to Coq's equivalent of Equation (2). The statements of the properties mentioned from the standard library are also shown in Figure 4. `mk_list_true` creates a bit-vector that represents $\sim$0, of the length given to it as input, and `bv_ule` is the representation of $\leq_u$ in the bit-vector library. `bv_ule` has output type `bool` (and so we equate terms in which it occurs to `true`), while the functions from the standard library have output type `Prop`. We also have two definitions for $\gg_a$. Proof of their equivalence (as done for the other shift operators) is planned for future work.

Table 1 summarizes the results for proving invertibility equivalences that correspond to invertibility conditions for the signature Σ_0. In the table, ✓ means that we have verified the invertibility equivalence in Coq, and that it was not verified in [9], while ✓ means the opposite. ✓✓ means that the invertibility equivalence was verified using both approaches, and ✗ means that it was verified in neither. We successfully proved all invertibility equivalences over $=$ that are expressible in Σ_0, including 4 that were not proved in [9]. For the rest of the predicates, we focused only on the 5 invertibility equivalences that were not proved by [9], and were able to prove 4 of them. Overall, these results strictly improve the results

```
Lemma bv_ule_1_firstn : forall (n : nat) (x : bitvector),
        (n < length x)%nat ->
        bv_ule (firstn n x) (firstn n (mk_list_true (length x))) = true.

Lemma bv_ule_pre_append : forall (x y z : bitvector), bv_ule x y  = true ->
        bv_ule (z ++ x) (z ++ y) = true.

Theorem app_comm_cons : forall (x y:list A) (a:A), a :: (x ++ y) = (a :: x) ++ y.

Lemma firstn_length_le: forall l:list A, forall n:nat,
        n <= length l -> length (firstn n l) = n.
```

Figure 4: Examples for lemmas that were used in proofs of invertibility equivalences.

$\ell[x]$	$=$	$\neq$	$<_u$	$>_u$
$x \,\&\, s \bowtie t$	✓	✓	✓	✓
$x \mid s \bowtie t$	✓	✓	✓	✓
$x \ll s \bowtie t$	✓	✓	✓	✓
$s \ll x \bowtie t$	✓✓	✓	✓	✓
$x \gg s \bowtie t$	✓✓	✓	✓	×
$s \gg x \bowtie t$	✓✓	✓	✓	✓
$x \gg_a s \bowtie t$	✓	✓	✓	✓
$s \gg_a x \bowtie t$	✓✓	✓	✓	✓
$x + s \bowtie t$	✓✓	✓	✓	✓

Table 1: Results of proving invertibility equivalences for literals in Σ_0.

of [9], as we were able to prove 8 additional invertibility equivalences in Coq. Taking into account our work together with [9], only one invertibility equivalence for the restricted signature is not fully proved yet, for the literal $x \gg s >_u t$. Note that one direction of it was successfully proved both in Coq and in [9], namely the direction $IC[s,t] \Rightarrow \ell[x,s,t]$.

6 Conclusion and Future Work

We have described our work-in-progress on verifying bit-vector invertibility conditions in the Coq proof assistant, which required extending a bit-vector library in Coq. The most immediate direction for future work is proving more of the invertibility equivalences supported by the bit-vector library. In addition, we plan to extend the library so that it supports the full syntax in which invertibility conditions are expressed, namely Σ_1. We expect this to be useful also for verifying properties about bit-vectors in other applications. Finally, we plan to open source the library and make the connection between the various definitions in the different layers of the library more robust, so that general users can use any of them with an easy conversion.

References

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

[2] Tej Chajed, Haogang Chen, Adam Chlipala, Joonwon Choi, Andres Erbsen, Jason Gross, Samuel Gruetter, Frans Kaashoek, Alex Konradi, Gregory Malecha, Duckki Oe, Murali Vijayaraghavan, Nickolai Zeldovich & Daniel Ziegler: *Bedrock Bit Vectors Library*. Available at https://github.com/mit-plv/bbv.

[3] Lukasz Czajka & Cezary Kaliszyk (2018): *Hammer for Coq: Automation for Dependent Type Theory*. *J. Autom. Reasoning* 61(1-4), pp. 423–453.

[4] Jean Duprat: *Library Coq.Bool.Bvector*. Available at https://coq.inria.fr/library/Coq.Bool.Bvector.html.

[5] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds & Clark W. Barrett (2017): *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*. In: *CAV (2)*, *Lecture Notes in Computer Science* 10427, Springer, pp. 126–133.

[6] Herbert Enderton & Herbert B Enderton (2001): *A mathematical introduction to logic*. Elsevier.

[7] Aarti Gupta & Allan L. Fisher (1993): *Representation and Symbolic Manipulation of Linearly Inductive Boolean Functions*. In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '93, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 192–199. Available at http://dl.acm.org.stanford.idm.oclc.org/citation.cfm?id=259794.259827.

[8] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett & Cesare Tinelli (2018): *Solving Quantified Bit-Vectors Using Invertibility Conditions*. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pp. 236–255, doi:10.1007/978-3-319-96142-2_16. Available at https://doi.org/10.1007/978-3-319-96142-2_16.

[9] Aina Niemetz, Mathias Preiner, Andrew Reynolds Yoni Zohar, Clark Barrett & Cesare Tinelli: *Towards Bit-Width-Independent Proofs in SMT Solvers*. To appear in the proceedings of CADE 2019.

[10] Tobias Nipkow, Lawrence C Paulson & Markus Wenzel (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. 2283, Springer Science & Business Media.

[11] Matthieu Sozeau (2010): *Equations: A Dependent Pattern-Matching Compiler*. In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pp. 419–434, doi:10.1007/978-3-642-14052-5_29. Available at https://doi.org/10.1007/978-3-642-14052-5_29.

[12] The Coq development team (2019): *The Coq Proof Assistant Reference Manual Version 8.9*. Available at https://coq.inria.fr/distrib/current/refman/.