Certified Decision Procedures for Width-Independent Bitvector Predicates

SIDDHARTH BHAT*, University of Cambridge, United Kingdom
LÉO STEFANESCO*, University of Cambridge, United Kingdom
CHRIS HUGHES, Independent Researcher, United Kingdom
TOBIAS GROSSER, University of Cambridge, United Kingdom

Bitvectors are foundational for automated reasoning. A few interactive theorem provers (ITP), such as Lean, have strong support for deciding *fixed-width* bitvector predicates by means of bitblasting. However, even these ITPs provide little automation for *width-independent* bitvector predicates. To fill this gap, we contribute novel, mechanized decision procedures for width-independent bitvector predicates in Lean. Classical algorithms to decide fragments of width-independent bitvector theory can be viewed from the lens of model checking, where the formula corresponds to an automaton and the correctness of the formula is a safety property. However, we cannot currently use this lens in mechanized proofs, as there are no executable, fast, and formally verified model checking algorithms that can be used interactively *from within* ITPs. To fill this gap, we mechanize key algorithms in the model checking literature: k-induction, automata reachability, automata emptiness checking, and automata minimization. Using these mechanized algorithms, we contribute scalable, mechanized, decision procedures for width-independent bitvector predicates. Furthermore, for controlled fragments of mixtures of arithmetic and bitwise operations which occur in the deobfuscation literature, we mechanize a recent fast algorithm (MBA-Blast), which outperforms the more general procedures on this fragment. Finally, we evaluate our decision procedures on benchmarks from classical compiler problems such as Hacker's Delight and the LLVM peephole optimizer, as well as on equivalence checking problems for program obfuscation. Our tools solve 100% of Hacker's Delight, two of our tools solve 100% of the deobfuscation dataset, and up to 27% of peephole rewrites extracted from LLVM's peephole rewriting test suite. Our new decision procedures provide a push-button experience for width-independent bitvector reasoning in interactive theorem provers, and, more broadly, pave the way for foundational algorithms for fast, formally verified model checking.

CCS Concepts: • **Software and its engineering** → **Formal methods**; • **Theory of computation** → *Logic and verification.*

Additional Key Words and Phrases: bitvector, decision procedure, model checking, automata, Lean

ACM Reference Format:
Siddharth Bhat, Léo Stefanesco, Chris Hughes, and Tobias Grosser. 2025. Certified Decision Procedures for Width-Independent Bitvector Predicates. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 370 (October 2025), 23 pages. https://doi.org/10.1145/3763148

1 Introduction

Effective reasoning about bitvectors is an essential tool for program verification, as, intuitively, the pointer-free fragment of programming languages can be expressed as bitvectors, corresponding to

*Both authors contributed equally to the paper

Authors' Contact Information: Siddharth Bhat, Department of Computer Science and Technology, University of Cambridge, Cambridge, United Kingdom; Léo Stefanesco, Department of Computer Science and Technology, University of Cambridge, Cambridge, United Kingdom; Chris Hughes, Independent Researcher, United Kingdom; Tobias Grosser, Department of Computer Science and Technology, University of Cambridge, Cambridge, United Kingdom.

ACM 2475-1421/2025/10-ART370
https://doi.org/10.1145/3763148

the memory representation of objects such as integers, enums, and structures. More specifically, bitvectors model integer operations in compiler intermediate representations (e.g., LLVM-IR [46]) and machine code [1, 40], combinational logic in hardware designs [10], and are a central component when modeling floating point arithmetic [8, 9].

In program verification, the bitvectors usually have concrete widths, corresponding to the size of the objects being verified, and, as such, the problem of deciding formulas containing fixed-width bitvectors has been thoroughly studied. For example, Bitwuzla [35] is a recent tool specialized for this task, and, in the context of proof assistants, Lean [17] provides a powerful `bv_decide` tactic backed by a SAT solver. Fixed-width solvers, however, face scaling issues as the width of the bitvectors increases, and have difficulties coping with bitwidths above 512 that are common in domains such as hardware design and cryptography [30, 34].

Width-independent proof automation was first studied by Pichora et al [37], where the width of the bitvectors is a universally quantified variable. That is, the decision problem asks to decide whether the formula holds for *all* bitwidths. In the context of interactive theorem provers (ITP), a further application of width-independent bitvector automation is simply to help the user of the ITP automatically solve certain goals. This plays a similar role to the `omega` or `lia` tactics, which solve goals in the linear fragment of arithmetic in ITPs such as Lean or Rocq [44], and are essential tools for practitioners. One particular instance is actually helping to prove the correctness of fixed-width bitvector solvers based on bitblasting, as modern fixed-width bitblasters rely on width-independent rewriting and normalization procedures to simplify expressions before bitblasting [36].

Bitvectors of size w can be seen in two ways: with an arithmetic lens, they are the ring of integers modulo 2^w; with a bitwise lens, they are a sequence of w Booleans. The two fragments of the theory of width-independent bitvectors corresponding to these two aspects can be solved using existing solvers: purely arithmetic problems can be handled by solvers for the theory of commutative rings [20], and purely bitwise problems can be reduced to a Boolean satisfiability problem using a bitwise extensionality principle. In this paper, we focus on larger fragments which combine these two aspects. In particular, we consider the *linear fragment of theory of bitvector*, which contains bitwise operations, the linear fragment of arithmetic (addition and subtraction), as well as equality and both signed and unsigned comparison relations. We implement two decision procedures for this fragment, and formally prove that both decision procedures are correct using the Lean interactive theorem prover. Furthermore, we embed them as tactics, so that they can be used interactively *inside* of Lean to solve goals. This means in particular that the specification of the procedure need not be trusted, as it is directly connected to each statement where they are used. Both of these decision procedures are based on ideas from model checking, and perform a reduction to automata, followed by a safety property [41] analysis (automata emptiness checking, k-induction).

Using automata theory. The first decision procedure uses *proof by reflection* and the theory of *non-determinisitic finite automata* (NFA): using metaprogramming, the Lean goal G (that is, the Lean abstract syntax tree) is transformed into a *linear bitvector formula* F, a datatype which corresponds to the fragment our decision procedures can solve. For example, the identity $-x = \neg x + 1$, which mixes bitwise and arithmetic operations, is transformed into the following formula F

```
  .eq (.neg (.var 0)) (.add (.not (.var 0)) (.ofNat 1)).
```

Then, an NFA $\mathcal{A}_F$ = `nfaOfFormula`(F) is computed from the formula F using a Lean function `nfaOfFormula`. The language recognized by this NFA is (the encoding of) the set of bitvectors that satisfy the formula F. As such, to decide whether the goal G is true, it suffices to check that $\mathcal{A}_F$ is *universal*, that it, that it recognizes all the words on its alphabet, which is a decidable problem. Thus, to check whether G holds, Lean's kernel only needs to check that a Boolean-valued function reduces to `true`. For more efficiency, we use Lean's native compiler on this function. While this reduction to automata is an instance of a well known theory called *automatic structures* [6], this paper presents, to the best of our knowledge, the first time it is used in the context of a proof assistant.

Using k-induction. In our second approach, the automaton corresponding to the formula F is not explicitly constructed, instead its transitions are described using *logical circuits*. The formula is true if all reachable states of the automaton are accepting. We use a standard technique in model checking called *k-induction* to establish an inductive invariant that ensures that the states of the automaton always output true. The algorithm tries to find an appropriate horizon k, starting from $k = 1$, such that the safety property can be established by proving an inductive invariant that reasons about transitions with k steps. First, we prove a precondition expressing that states reachable from the initial state in k steps always output true. If we fail to prove this, then we have a concrete counterexample, which we report to the user. Next, having established the precondition, we try to establish an inductive invariant, which shows that for any state s, there exists a horizon $l \leq k$ such that all states reachable from s in l steps output true. If we fail to establish the inductive invariant, we increment the horizon k and try again. For more efficiency, we can use an external SAT solver to prove the tentative inductive invariants.

Mixed Boolean Arithmetic. In addition, we consider a smaller fragment, called *mixed boolean arithmetic* [47] (MBA) in which expressions are stratified: arithmetic operations cannot appear inside bitwise operations. This fragment is of particular interest because it is used for program obfuscation, and traditional SMT solvers seem to scale poorly on this fragment. However, a new algorithmic idea [27, 39] of reducing the problem to a statement over bounded integers has made this problem tractable. We formalize and implement this idea, yielding a more efficient decision procedure when the problem is contained in this smaller fragment. Furthermore, we extend this algorithm to the unsigned inequality fragment (under mild assumptions), which is used in the context of peephole optimizations in compilers.

Contributions. Toward building a push-button experience for width-independent bitvector reasoning in Lean, we use the above algorithmic knowledge to mechanize fundamental algorithms, which power tactics for width-independent reasoning in Lean. Concretely, we contribute the following:

- The mechanization of a fast finite automata library in a dependently typed ITP. Our library powers a novel tactic for deciding width-independent bitvector predicates.
- The first formally verified, proof producing implementation of the k-induction algorithm in a proof assistant. The implementation is used to provide a novel tactic to decide width-independent bitvector predicates, and also supports much faster solving using Lean's in-built SAT solving capabilities.
- The first mechanization of the MBA-Blast algorithm for deciding width-independent mixed-boolean-arithmetic equalities, along with a new tactic powered by the algorithm, along with a novel extension of the MBA-blast algorithm to the unsigned inequality fragment.
- An evaluation of our tactics on datasets from program obfuscation [28] and peephole rewrites in compilers extracted from the LLVM compiler test suite. We show that two of our tactics can solve *all* the problems from the 64-bit MBA-Blast dataset, and all solve a large fraction of the problems from LLVM's peephole rewrite test suite, which has been extracted into Lean.

2 Notation & Background

Notation. $\mathbb{N}$ are the natural numbers. We assume an infinite set of variables $\mathbb{V}$. $\mathbb{B}$ is the finite field of Booleans. We define bitstreams as infinite sequences of booleans ($\mathsf{Bitstream} \coloneqq \mathbb{N} \to \mathbb{B}$).

$$\begin{aligned}
\mathsf{E} \coloneqq\ & \mathsf{ofNat}\ (v : \mathbb{N}) \\
\mid\ & \mathsf{Var}\ (x : \mathbb{V}) \\
\mid\ & \mathsf{E}_1 \boxplus \mathsf{E}_2 \quad (\boxplus \in \{+, -, \|, \&, \oplus\}) \\
\mid\ & -\mathsf{E} \quad \mid \sim \mathsf{E} \\
\mid\ & \mathsf{E} \ll k
\end{aligned}
\qquad
\begin{aligned}
\mathsf{F} \coloneqq\ & \mathsf{E}_1 \otimes \mathsf{E}_2 \quad (\otimes \in \{<_u, \leq_u, \leq_s, <_s, =, \neq\}) \\
\mid\ & \mathsf{width}\ (r : \mathsf{WidthRel})\ (k \in \mathbb{N}) \\
\mid\ & \mathsf{F}_1 \land \mathsf{F}_2 \quad \mid \mathsf{F}_1 \lor \mathsf{F}_2 \\
\mid\ & \mathsf{F}_1 \Rightarrow \mathsf{F}_2 \\
\mid\ & \neg \mathsf{F}
\end{aligned}$$

$$\mathsf{WidthRel} \coloneqq\ < \mid \leq \mid > \mid \geq \mid = \mid \neq$$

Fig. 1. Syntax of bitvector expressions and formulas

Bitvectors. A bitvector x of width $w \in \mathbb{N}$ is a sequence of w booleans, and the corresponding sequence of bits is written $\langle x_{w-1}, \ldots, x_0 \rangle$. For example, $x = \langle 1, 0, 1 \rangle$ is a bitvector of length 3. We write $\mathsf{BitVec}\ w$ for the set of bitvectors of width w. Every bitvector x can be interpreted as a natural number, by interpreting the sequence of bits as a binary number $\mathsf{nat}(x) \coloneqq \sum_{i=0}^{w-1} x_i * 2^i$ (In this case, $\mathsf{nat}(x) = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$). We can also interpret a bitvector as a two's complement integer, $\mathsf{int}(x) \equiv \sum_{i=0}^{w-2} x_i * 2^i - x_{w-1} * 2^{w-1}$ (In this case, $\mathsf{int}(x) = 1 \cdot 2^1 + 0 \cdot 2^0 - 1 \cdot 2^2 = -3$). Bitvectors are both a boolean algebra (a sequence of bits) and a ring (seen as integers modulo 2^w), and the two algebraic structures mix in subtle ways when reasoning about bitvectors (e.g., $-x = \neg x + 1$).

Signed and unsigned comparison. Because bitvectors can either be seen as representing signed or unsigned integers, they are equipped with two relevant orders depending on how they are interpreted, and which our decision procedures need to handle:

$$\begin{aligned}
\textit{Unsigned comparison:} \quad & x <_u y \iff \mathsf{nat}(x) < \mathsf{nat}(y) \\
\textit{Signed comparison:} \quad & x <_s y \iff \mathsf{int}(x) < \mathsf{int}(y)
\end{aligned}$$

3 Reducing Width-Independent Bitvector Formulas to Model Checking

We formally define the fragment of bitvector theory we decide, and then describe the reduction to automata, followed by the model checking algorithm and its mechanization.

3.1 Syntax and Semantics of formulas

Syntax. The syntax of bitvectors and formulas is given in Figure 1. All the bitvectors have the same symbolic bitwidth which we denote by w in the paper. The base expressions are constants ($\mathsf{ofNat}\ v$) and variables ($\mathsf{Var}\ i$). The constant expression ($\mathsf{ofNat}\ v$) denotes a *width-independent* constant: For example, ($\mathsf{ofNat}\ 3$) denotes the empty bitvector $\langle\rangle$ at width 0, $\langle 1 \rangle$ at width 1, $\langle 11 \rangle$ at width 2, $\langle 011 \rangle$ at width 3, and so on. These atoms can be combined using the usual bitwise operations (bitwise or $\|$, and $\&$, xor $\oplus$, negation $\sim$ as well as left shift $\ll$ by a constant) and arithmetic addition, subtraction and negation.

The formulas are built from relations for equality of bitvectors, as well as signed and unsigned inequalities, which can be combined using the usual Boolean operators. Moreover, we support predicates which are assertions about the widths of the bitvectors. For instance, $\mathsf{width}\ (\leq)\ 4$ states that the width w is less than or equal to 4. This can, for example, be useful to prove results that only hold for non-trivial bitwidth ($w > 0$). Since the atomic predicates (equality, signed and unsigned comparison) are closed under negation, predicates can be normalized into negation normal form. We write $\mathsf{FV}(\mathsf{F})$ the set of free variables of the formula F.

Semantics. The semantics of expressions and formulas are parameterized by a bitwidth w and *environment* ρ mapping the free variables of E to bitvectors of width w. The semantics of an expression E is written $[\![\mathsf{E}]\!]^w_\rho$ and is defined in the standard way:

$$\begin{aligned}
[\![\mathsf{ofNat\ v}]\!]^w_\rho &\coloneqq \langle v_{w-1}, \ldots, v_0\rangle \text{ if } n \text{ can be written in binary } v_N \ldots v_0 \text{ with } N \geq w-1 \\
[\![\mathsf{Var\ x}]\!]^w_\rho &\coloneqq \rho(x) \\
[\![\mathsf{E}_1 \boxplus \mathsf{E}_2]\!]^w_\rho &\coloneqq [\![\mathsf{E}_1]\!]^w_\rho \boxplus [\![\mathsf{E}_2]\!]^w_\rho
\end{aligned}$$

Similarly, formulas are interpreted as predicates over the bitwidth and the environment in the obvious way. We can now formally state the decision problem: given a formula F, determine whether

$$\forall w \in \mathbb{N},\ \forall \rho \in \mathsf{FV}(\mathsf{F}) \to \mathsf{BitVec}\ w,\ \ [\![\mathsf{F}]\!]^w_\rho \text{ holds.}$$

3.2 Reduction to Automata

The first order theory of bitvectors described in the previous section can be decided by reducing it to the emptiness problem of a regular language; this was first discovered by Büchi [14] for Presburger arithmetic over (unbounded) natural numbers. Given a formula P with n-free variables $\vec{x} \equiv (x_1, \ldots, x_n)$, its solutions of width w are the subset $\mathsf{Sol}_w(\mathsf{P}) = \{\vec{bv} \in (\mathsf{BitVec}\ w)^n \mid [\![\mathsf{P}]\!]^w_{\vec{bv}}\}$, and its set of solutions is the set

$$\mathsf{Sol}(\mathsf{P}) = \bigcup_{w \in \mathbb{N}} \mathsf{Sol}_w(\mathsf{P}) \subseteq \bigcup_{w \in \mathbb{N}} (\mathsf{BitVec}\ w)^n$$

The decision problem amounts to deciding whether $\mathsf{Sol}(\mathsf{P})$ is equal to the whole set of possible solutions $\bigcup_{w \in \mathbb{N}} (\mathsf{BitVec}\ w)^n$.

This set of solutions is not a *language* itself, as it not composed of words over a finite alphabet, but is isomorphic to one: a bitvector bv of width w can be encoded as a word of length w over the alphabet $\{0, 1\}$ using a big endian convention, and a tuple $\vec{bv}$ of n bitvectors of width w can be encoded as a word over the alphabet $\{0, 1\}^n$, where the kth letter corresponds to the kth bit of each bitvector in $\vec{bv}$. In summary, the solutions (over all widths) of a formula P can simply be seen as a language over the alphabet $\{0, 1\}^n$. This representation also has the advantage of being much simpler than the one above, as the width of the bitvectors is now implicit.

It now remains to see that the language $\mathsf{Sol}(\mathsf{P})$ is regular, and is therefore recognizable by a finite automaton. As the logical connectives in Figure 1 correspond to union and intersection of the solutions, they preserve the regularity of the language of solutions, and it suffices to recognize the atomic predicates. This can be further split into two problems, that of recognizing terms and relations.

Terms. We recognize terms T, by providing a finite automaton $\mathcal{A}_\mathsf{T}$ over the alphabet $\{0, 1\}^{n+1}$ where n is the number of free variables in T. The first n bits of the letters correspond to the values of the free variables, and the last bit corresponds to the value of T. In other words, $\mathcal{A}_\mathsf{T}$ is a transducer. For example, Figure 2 implements the XOR operation and addition: For XOR, there is single state, and the transitions $[x_1, x_2, y]$ express $y = x_1 \oplus x_2$ for the 4 possible values of the inputs x_1 and x_2. For addition, the states of the NFA correspond to the carry, and each transition labeled with $[x_1, x_2, y]$ determines the output bit y and the new carry state based on the two input bits x_1 and x_2 and the current carry state. We provide an automaton for each of the constructors of expressions E in Figure 1.

Of course, this technique is limited by the expressivity of NFAs. For example, multiplication cannot be represented by a finite automaton because the ith output bit cannot be determined from the ith input bit and a finite state. Another, perhaps more surprising, non-example are *right* bitshifts

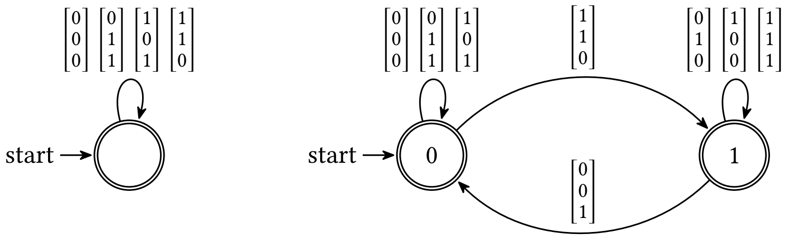


Fig. 2. Automata implementing XOR (left) and addition of bitvectors (right)

by a constant k: here the issue is that the ith bit of the output depends on the $i + k$th bit of the input, so the automaton needs a delay of k bits before starting to output bits. We believe this is not a fundamental limitation of NFAs and, if we considered a more complex encoding of bitvectors into strings with a special letter for delay, we could handle right shifts.

Relations. We recognize each basic relation $\otimes \in \{<_u, \leq_u, \leq_s, <_s, =, \neq\}$, by providing an automaton $\mathcal{A}_\otimes$ over the alphabet $\{0,1\}^2$ which recognizes the relation $\otimes$, in the sense that $bv_1 \otimes bv_2$ iff the automaton $\mathcal{A}_\otimes$ recognizes the encoding of the pair (bv_1, bv_2). For instance, the automaton for unsigned comparison $\leq_u$ has three states, corresponding to the following cases:

(1) the state = corresponds to the case where, so far, the bits of the two bitvectors are equal;
(2) the state < corresponds to the case where, based on the bits that have been read, the first operand is smaller;
(3) the state > is dual to <.

The initial state is =, and the accepting states are = and < since we wish to recognize $\leq_u$. The automaton for signed comparison $\leq_s$ is similar, but has two additional states $<_{\text{end}}$ and $>_{\text{end}}$ without outgoing transitions, so that they can only be reached by the last letter, and allows to treat the sign bit differently from all the others. The accepting states are = and $<_{\text{end}}$.

Using standard techniques from automata theory, these automata can be combined into an automaton recognizing any formula. This technique is an instance of a more general framework called *automatic structures* [6].

4 Automata-Based Decision Procedure

The first model checking method presented here uses standard automata theory to directly decide if the automaton $\mathcal{A}_{\mathsf{P}}$ is *universal*, that is, whether it recognizes all the words over its alphabet. At a high level, we first develop an efficient automata library in Lean, and use it to define

- a function `nfaOfFormula` that takes a formula P as input and produces an automaton which recognizes its solutions,
- a function `isUniversal` which returns a Boolean indicating whether an automaton is universal.

By composing them, we obtain a function

```
def formulaIsUniversal (F : Formula) : Bool :=
  (nfaOfFormula F).isUniversal
```

which satisfies the following correctness theorem:

```
theorem decision_procedure_is_correct (F : Formula) :
    formulaIsUniversal F = true → ∀ (w : Nat) (env : Nat → BitVec w), F.sat env
```

This allows us to do *proofs by reflection*: to prove a Lean formula ϕ, a tactic computes a syntactic formula F and an environment ρ such that $\phi \iff [\![\mathsf{F}]\!]_\rho$. It then suffices to check that the Lean

Boolean term `formulaIsUniversal F` is equal to true. To be more efficient, we ask Lean to compile `formulaIsUniversal` to native code, and to invoke this native code during type checking.

4.1 A Certified and Efficient NFA Library

As part of this decision procedure, we implement a formally verified NFA library in Lean which needs to be efficient in order to decide large problems.

Data representation. One important feature for efficiency is that the states of the automaton do not have an identity, they are instead natural numbers between 0 and `stateMax`. Thus, a *computational* NFA over the alphabet `A` is defined as follows in Lean:

```
structure RawCNFA (A : Type) where
  stateMax : Nat
  initials : Std.HashSet Nat
  finals   : Std.HashSet Nat
  trans    : Std.HashMap (Nat × A) (Std.HashSet Nat)
```

To be able to reason about such automata, we define a well-formedness predicate `m.WF` which states that all the integers in the `RawCNFA` are less than `m.stateMax`. A `CNFA` is then a (dependent) pair of a `RawCNFA` and a proof that it is well-formed. We found it easier to work with a simply typed raw data-structure with an external well-formedness predicate than with an intrinsically typed version that would use dependent pairs `{s : Nat // s < stateMax}` as states instead of simple integers, as adding a new state to an automaton would increment `stateMax` and thus invalidate this type. All high-level operations are exposed in terms of `CNFA`.

Operations over automata. There are a handful of widely useful operations over NFAs, namely the product construction (to define disjunction and conjunction of formulas) and the powerset construction (to determinize, and thus to negate). In the usual textbook description of these operations, the states of the resulting automaton are (respectively) pairs of states of the input automatons, and subsets of the input operation. This works well when the goal is to reason about these constructions, but it is computationally inefficient, as unreachable states are constructed.

A more efficient method is to use a *worklist* algorithm, where states to be processed are put on a queue and processed until it becomes empty, as described for example by Esparza and Blondin [18]. However, termination of worklist algorithms is non-trivial to establish formally, as one needs to show that the (finite) set of states of the resulting automaton that have not yet been explored is decreasing. To amortize this effort, we define a generic worklist algorithm for NFAs which dramatically reduces the required proof effort for the individual operations.

Generic construction algorithm. The idea is that, although our automata states do not have identities, *during the construction*, we need to keep track of the mapping between states in the new automaton and state(s) in the input automata. Consider, for instance, the product construction: suppose we want to compute the product automaton of `m1 : CNFA A` and `m2 : CNFA A`. During the construction of the product, given a state s_{new} of the product automaton, we need to know what the successor states of s_{new} should be, since we are building the automaton incrementally starting from its initial states and following transitions. For this, we need to know that it corresponds to some pair (s_1, s_2) of the input automata `m1` and `m2` in order to be able to determine its successor states and potentially add new states to the worklist: states that correspond to pairs (s_1', s_2') reachable from s_1 and s_2 using the same letter. Generalizing, we posit a type of states `S` together with operations:

```
step  : S → Array (A x S)
inits : Array S
final : S → Bool
```

We think of the elements of `S` as the states of the automaton under construction: for example pairs of states (integers) for the product or finite sets of states of the input automaton for the powerset construction. The worklist algorithm keeps track of the *worklist*, the set of elements of `S` that need to be processed and a *mapping* from `S` to integers corresponding to the states of the new automaton. The initial state of the queue is given by `inits`, and the outgoing transitions of a state s are given by `step s`, and the new states are suitably added to the worklist. Finally, the `final` function determines which states are final. Termination of this algorithm is proved once and for all by establishing that the set of states in `S` that are not in the mapping or are in the worklist is strictly decreasing. With this generic algorithm at hand, defining these constructions essentially boils down to writing down their textbook definitions.

Specification by Bisimulation. We choose to never directly reason about the language accepted by a `CNFA`, instead we use Mathlib's [32] `NFA` type, which corresponds to the textbook definition of an automaton as a tuple $\langle Q, A, I, F, \delta\rangle$ of a set Q of states, the subsets I and F of initial and final states, and a transition relation δ. We then completely specify a `CNFA` using an abstract NFA to which it is *bisimilar* [26]. This is an extension of the notion of bisimulation and of bisimulation from labeled transition systems (LTS) to NFAs: two NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ over the same alphabet are *bisimilar* if there exists a relation R between their sets of states such that:

(1) forgetting initial and final states, R is a simulation between the LTSs (Q_1, δ_1) and (Q_2, δ_2), that is, if $R(q_1, q_2)$ and $q_1' \in \delta_1(q_1, a)$, then there exists $q_2' \in \delta_2(q_2, a)$ such that $R(q_1', q_2')$;
(2) for any initial state $q_1 \in I_1$ of $\mathcal{A}_1$, there exists an initial state q_2 of $\mathcal{A}_2$ such that $R(q_1, q_2)$, and vice-versa;
(3) given $R(q_1, q_2)$, q_1 is final in $\mathcal{A}_1$ iff q_2 is final in $\mathcal{A}_2$.

The choice of NFA as specification does not matter, since two bisimilar automata necessarily recognize the same language: The advantage of this strategy is that, once the bisimulation has been established, we can reason about the properties of mathematical NFAs, a substantially easier task.

One example of this style of specification is the generic worklist algorithm: with only very weak assumptions on `S` and its associated operations (to ensure termination), we establish that the `CNFA` it produces is in bisimulation with the NFA

$$\mathcal{A}_W \;\coloneqq\; \left\langle \mathtt{S}, \mathtt{A}, \mathtt{inits}, \{q \mid \mathtt{final}\ q = \mathtt{true}\}, \mathtt{step} \right\rangle$$

It then suffices to provide a bisimulation between the NFA above and, say, the product automaton, which is straightforward. For instance, when constructing the product automaton, we assume that we have `m1` and `m2` which are (respectively) bisimilar to two abstract NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, with relations R_1 and R_2 between their states. When running the worklist algorithm, we have chosen $S = \mathbb{N} \times \mathbb{N}$ (corresponding to states of the two input automata). It is then straightforward to prove that the relation R defined by

$$R((s_1, s_2), (q_1, q_2)) \iff R_1(s_1, q_1) \;\wedge\; R_2(s_2, q_2)$$

is a bisimulation between $\mathcal{A}_W$ and $\mathcal{A}_1 \times \mathcal{A}_2$, by proving that the parameters `inits`, `step` and `final` passed to the worklist algorithm correspond to the standard product construction.

Using bisimilarity, we define the language (formalized as a set of lists of letters) recognized by a CNFA as follows:

```
def CNFA.recognizes (m : CNFA A) (L : Language A) :=
  ∃ (M : NFA A), m.Sim M ∧ M.accepts = L
```

and prove that this relation is functional (in the sense that, if an automaton recognizes L_1 and L_2, then $L_1 = L_2$).

Minimization. To minimize an automaton $\mathcal{A}$, we use Brzozowski's algorithm as it is simple to implement and verify, it consists of reversing (reversing all the transitions, and swapping initial and final states) and determinizing the NFA twice, yielding the minimal deterministic automaton recognizing the same language as $\mathcal{A}$.

One difficulty is that simulation and reversal are not compatible, in the sense that it is not true that, if $\mathcal{A}_1$ is bisimilar to $\mathcal{A}_2$, then $\mathcal{A}_1^R$ is bisimilar to $\mathcal{A}_2^R$. Indeed, the notion of (forward) simulation is fundamentally oriented, and does not say anything about backwards simulation. There is a trick though: Since we only care about the language recognized by $\mathcal{A}_2$, we can replace $\mathcal{A}_2$ with another automaton which is easier to work with, as long as it is bisimilar to $\mathcal{A}_1$, namely $\mathcal{A}_1$ itself! The proof that the language of $\mathcal{A}^R$ is the reverse of the language of $\mathcal{A}$ becomes easy, and thus the proof that Brzozowski's algorithm is correct.

4.2 Mechanization of the Reduction to Automata

With this NFA library in hand, it is relatively straightforward to implement the encoding of bitvector formulas into automata:

Correspondence between solutions and language. We formalize the isomorphism between the set of solutions of the formula, which is a set of vectors of n bitvectors of arbitrary bitwidths, and the language over the alphabet `BitVec n`, by writing an encoding and decoding function, and proving they are inverses of each other. The former is defined as a dependent pair of a number w and a vector of n bitvectors of width w:

```
structure BitVecs (n : Nat) where
 w : Nat
 bvs : Vector (BitVec w) n
```

We then give a specification to each construction on NFAs over the alphabet BitVec n in terms of the set of `BitVecs n` that it accepts.

Composition of relations and operations. To simplify the encoding, we use an over-approximation of the set $\mathsf{FV}(\mathsf{F})$ of free variables: we only keep track of $n = \max \mathsf{FV}(\mathsf{F})$. Each formula F is then interpreted as an NFA over the alphabet BitVec n. As such, when defining the automaton for, say, $\mathsf{F}_1 \wedge \mathsf{F}_2$, the product construction cannot be used directly, as the two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ have (in general) different alphabets, when the two formulas have different free variables.

The solution is to *lift* both automata to a common alphabet: given an automaton $\mathcal{A}$ over the alphabet A^n, and a function $f : \{0, \ldots, n-1\} \to \{0, \ldots, m-1\}$, the lifting of $\mathcal{A}$ along f is an automaton over the alphabet A^m with the same states, and whose transitions are defined as $q \xrightarrow{\vec{a}} q'$ whenever $q \xrightarrow{f^*(\vec{a})} q'$ in $\mathcal{A}$. The ith component of the vector $f^*(\vec{a})$ is defined to be equal to $a_{f(i)}$.

For expression, the situation is similar, if more complex: an expression with n free variables is interpreted as an automaton over the alphabet BitVec $(n+1)$, where the last component corresponds to one bit of the *result* of the expression. Therefore, to interpret $\mathsf{E}_1 + \mathsf{E}_2$, we need to build an automaton over BitVec $(k+3)$, with $k = \max(n_1, n_2)$. The bit k corresponds to the result of E_1, the bit $k+1$ to that of E_2 and the last bit, at position $k+2$, is the result of the addition of the bits k and $k+1$. We then remove the bits k and $k+1$ using *projection*, an operation dual to *lifting*.

With these tools at hand, it is straightforward to interpret any formula as an NFA. We choose to apply a minimization procedure at each step of the process in order to keep their size manageable.

Automata for the base cases. The standard constructions on NFAs handle the Boolean structure of formulas. To handle the operations on terms and the base logical predicates over bitvectors, we

$$[\![-]\!] : \mathsf{Term} \times \mathsf{Env} \to (\mathbb{N} \to \mathbb{B}) \quad \mathsf{Env} \equiv \mathbb{V} \to (\mathbb{N} \to \mathbb{B})$$

$$[\![\mathsf{Var}(x)]\!]_\rho (i) \equiv \begin{cases} 0 & i = 0 \\ \rho(x)(i) & \text{otherwise} \end{cases} \quad [\![\mathsf{ofNat}(i)]\!]_\rho (i) \equiv \begin{cases} 0 & i = 0 \\ \lfloor v/2^i \rfloor \mod 2 & \text{otherwise} \end{cases}$$

$$[\![s||t]\!]_\rho (i) \equiv [\![s]\!]_\rho (i) || [\![t]\!]_\rho (i) \quad [\![\neg s]\!]_\rho (i) \equiv \begin{cases} 0 & i = 0 \\ 1 - [\![s]\!]_\rho (i) & \text{otherwise} \end{cases}$$

$$[\![P]\!]_\rho (i) = 1 \iff \text{Predicate } P \text{ holds at bitwidth } i \text{ for environment } \rho$$

$$[\![s = t]\!]_\rho (i) \equiv \forall j \leq i, [\![s]\!]_\rho (j) = [\![t]\!]_\rho (j) \quad [\![P \wedge Q]\!]_\rho (i) \equiv [\![P]\!]_\rho (i) \wedge [\![Q]\!]_\rho (i)$$

Fig. 3. Bitstream semantics of the bitvector theory.

construct explicit NFAs which recognize them, this (somewhat tedious) work is done using the following procedure:

(1) write a program constructing `m : CNFA`, it is usually straightforward;
(2) describe the same automaton as an abstract `NFA` $\mathcal{A}$ with a concrete set Q of states;
(3) prove there is a simulation between `m` and $\mathcal{A}$, by defining a bijection from Q to $\{0, ..., N - 1\}$ where N is the (concrete) size of `m`;
(4) prove $\mathcal{A}$ recognizes the right language, by providing, for each $q \in Q$ the language it recognizes and checking that each transition respects this assignment.

Since these constructions are bounded with a concrete size (except for the width predicates), *ad hoc* proof automation is usable and the proof effort is moderate.

5 Model Checking by k-Induction

The k-induction based decision procedure uses a similar construction to the previously described automata encodings, but instead, chooses to view the automata as *transducers*, whose outputs are infinite bitstreams.

5.1 Reduction to Transducers

The problem of deciding a bitvector formula with a universally quantified bitwidth is reduced to checking that the same formula interpreted in the domain of infinite bitstreams always outputs 1. The interpretation of the terms E and the formulas F (defined in Figure 1) in terms of bitstreams is presented in Figure 3. The interpretation of terms is the straightforward extension of bitwise and arithmetic operations to bitstreams. Formulas are interpreted as bitstreams whose kth bit indicates whether the predicates holds for bits at positions $\leq k$. As such, the interpretation of a formula is always a non-increasing sequence.

Thus, deciding whether a formula F universally holds boils down to deciding whether its bitstream interpretation satisfies $\forall w\ \rho, [\![\mathsf{F}]\!]_\rho (w) = 1$. That is to say, whether the formula F holds for all bitwidths and environments. The difference from the automata-based approach is that rather than labeling states as accepting or rejecting, it instead builds *transducers*, which produce an output bit for each state.

The transducers are abstractly a tuple $(S, s_0 : S, \iota : \mathbb{N}, \pi : S \times \mathsf{BitVec}\ \iota \to \mathbb{B}, \delta : S \times \mathsf{BitVec}\ \iota \to S)$, where ι is the arity of the transducer (e.g., constants have arity 0, negation has arity 1, addition has arity 2), π is the output function, δ is the state transition function. However, since the set S

of states is *finite*, we can always replace it with BitVec σ, where σ is large enough to encode all states of the transducer. The encoding of transducers is now a tuple $(\sigma : \mathbb{N}, s_0 : \text{BitVec } \sigma, \iota : \mathbb{N}, \pi : \text{BitVec } \sigma + \iota \to \text{BitVec } 1, \delta : \text{BitVec } \sigma + \iota \to \text{BitVec } \sigma)$. Next, since the *function spaces* of π and δ are themselves finite, we can encode π and δ as boolean circuits, which will be exploited by the k-induction algorithm to build a SAT formula. The final definition is the following:

```
structure FSM (ι  : Type) where
  σ  : Type  -- state space size
  s0 : ι → Bool -- initial state
  π  : CircuitFamily (σ ⊕ ι) Unit -- output function: BitVec[σ+ι] → BitVec[1]
  δ  : CircuitFamily (σ ⊕ ι) σ -- transition function: BitVec[σ+ι] → BitVec[σ]
```

Given the finite state transducer that represents a formula F, we use model checking on the finite state transducer $\mathcal{F}_F$ to establish that the output function π on all states reachable from s_0 for all sequences of inputs always produces true. This in turn establishes that $[\![F]\!]_\rho(w) = \text{true}$ for all widths, which proves that $\forall w, F(w)$ is true. This hinges on the fact that $\forall w, F(w)$ is a safety property for the FSM $\mathcal{F}_F$, and thus, any counter-example is a finite trace, which can be found by symbolically exploring the state space of $\mathcal{F}_F$. The efficiency of this algorithm depends on the encoding of $\mathcal{F}_F$ and the algorithm used to establish the safety invariant that the output function π always emits true.

5.2 Establishing Safety via k-Induction

Reachable by path p:

$$(s : S) \xrightarrow{p}{}^{*} (u : S) \equiv \begin{cases} s = u & \text{if } p = \langle\rangle \\ \delta(s, p_0) = t \wedge t \xrightarrow{q}{}^{*} u & \text{if } p = \langle p_0; q\rangle \end{cases}$$

Reachable by path p, all intermediate states output true:

$$(s : S) \xrightarrow{p}{}^{*}_{\text{true}} (u : S) \equiv \begin{matrix} s = u & \text{if } p = \langle\rangle \\ \pi(s, p_0) = \text{true} \wedge \delta(s, p_0) = t \wedge t \xrightarrow{q}{}^{*}_{\text{true}} u & \text{if } p = \langle p_0; q\rangle \end{matrix}$$

States reachable in k steps from s_0 are safe:

$$\text{InitPrecond}_k \equiv \forall (t : S)\ (i : \mathbb{B})\ (p : \text{BitVec } k), s_0 \xrightarrow{p}{}^{*} t \implies \pi(t, i) = \text{true}$$

Safe reachability in k steps can be extended to $k + 1$ steps:

$$\text{Ind}_k \equiv \bigvee_{j=1}^{k} \forall (s\ t : S)\ (i : \mathbb{B})\ (p : \text{BitVec } k), s \xrightarrow{p}{}^{*}_{\text{true}} t \implies \pi(t, i) = \text{true}$$

Safety = Preconditions + Inductive Invariant:

$$\text{Safe}_k \equiv \text{InitPrecond}_k \wedge \text{Ind}_k$$

Fig. 4. We establish the safety property Safe_k to show that an FSM always outputs true. InitPrecond_k establishes that all states reachable in k steps from s_0 are safe, and Ind_k allows extending the safety frontier from k to $k + 1$. Together, these prove the FSM is safe. The procedure either finds a counter-example via InitPrecond_k for large k, or establishes Ind_k, as the set of states satisfying Ind_k is strictly increasing. See that the final safety property, Safe_k, consists of universally quantified predicates, and can thus be proven by showing UNSAT of the negated (existentially quantified) problem.

We define the safety property Safe_k as the conjunction of two properties (Figure 4): We notate the property of a state s being reachable from s_0 by a sequence of inputs p : BitVec k (for some natural number k) as $s_0 \xrightarrow{p}{}^{*} s$, and the fact that s transitions to t with input bit i producing output bit b as $s \xrightarrow{i}_b t$. InitPrecond_k establishes that all states reachable in k steps from s_0 are safe and Ind_k allows us to extend the safety frontier from k steps to $k + 1$ steps. Algorithmically, we either find a counter-example via $\mathsf{InitPrecond}_k$ for a large enough k, or we establish $\mathsf{InitPrecond}_k \wedge \mathsf{Ind}_k$, which establishes that the FSM is safe for all states.

Soundness. Assume that we establish $\mathsf{InitPrecond}_k \wedge \mathsf{Ind}_k$. So, all states reachable in k steps from s_0 are safe. Also, any state reachable in k steps from s_0 can be extended to $k + 1$ steps (by the induction hypothesis). Repeatedly applying this, we see that all states reachable from s_0 are safe, since the set of states is finite. This establishes that the FSM is safe, assuming the algorithm terminates.

Completeness. The set of states satisfying Ind_k is strictly increasing, since the set of states satisfying Ind_{k+1} is at least as large as the set of states satisfying Ind_k, as $\mathsf{Ind}_k \implies \mathsf{Ind}_{k+1}$. However, naive k-induction performs induction on the length of paths in the state transition graph of the finite state machine, which is unbounded. To ensure completeness, we add *simple path constraints*, which express that we check k-induction only on simple paths. This is standard, and prevents the algorithm from getting stuck due to the presence of an unreachable cycle of good states, that have an out transition into a bad state.

Note that having correctness and termination proves that k-induction is *complete* for the bitvector fragment we handle, which guarantees to the user that our automation is robust.

5.3 Mechanization in Lean

We mechanize the k-decision procedure in Lean, which uses Lean's inbuilt support for SAT solving to establish the safety property by producing a SAT formula that is then proven to be unsatisfiable. Our proof of correctness of the algorithm is to show that $\mathsf{InitPrecond}_k \wedge \mathsf{Ind}_k$ implies that the FSM produces true for all inputs by the k-induction argument. The safety of the SFM implies that the corresponding formula is true for all bitwidths. The proof structure is straightforward, establishes that the circuit we build checks two properties: (1) Safety of states reachable in $\leq k$ steps from the initial state (2) Safety on a simple path upto length k implies safety on the $k + 1$th step. These two properties together imply safety for all reachable states.

An interesting point is to compare this to the automata decision procedure — While both of these fundamentally rely on exploiting a finite description of the bitwise behavior of the circuits, the algorithm they employ is quite different. The automata decision procedure uses the well-known theory of finite automata to decide formulas, while the circuit based approach instead chooses to take a SAT inspired view of the same.

6 Tactic Implementation

We implement and verify our algorithms in the Lean proof assistant. We illustrate the overall flow of the algorithm on a Lean-level, universally quantified statement relating addition and bitwise-or (`|||`) (Figure 5). The Lean statement is first converted to a statement in terms of negation normal form. Then, it is rewritten into a reflected formula F whose denotation is the bitvector statement, using `Predicate.denote`.

```
def Predicate.denote (p : Predicate) (w : Nat) (vars : List (BitVec w)) : Prop :=
  match p with
  | .eq t₁ t₂ => t₁.denote w vars = t₂.denote w vars
  | ... (other cases elided)
```

The reflected predicate language is then transformed into the infinite stream semantics (Figure 3), using `Predicate.eval`.

```
def Predicate.eval (p : Predicate) (vars : List BitStream) : BitStream :=
  match p with
  | eq t₁ t₂ => Predicate.evalEq (t₁.eval vars) (t₂.eval vars)
```

1. Original Goal State

```
w : ℕ    a b : BitVec w    h : a &&& b = 0#w
⊢ a + b = a ||| b
```

2. Negation Normal Form

```
w : ℕ    a b : BitVec w
⊢ a &&& b ≠ 0#w ∨ a + b = a ||| b
```

3. Reflection into Term Language

```
w : ℕ    a b : BitVec w
⊢ (lor
    (neq ((var 0).and (var 1)) zero)
    (eq ((var 0).add (var 1)) (or (var 0) (var 1))))
   .denote w [a, b]
```

4. Bitstream Denotation

```
w : ℕ    a b : BitVec w
⊢ (lor
    (neq ((var 0).and (var 1)) zero)
    (eq ((var 0).add (var 1)) (or (var 0) (var 1))))
   .eval w [a, b] = BitStream.allOnes
```

5. FSM Denotation

```
⊢ ∀ (w : ℕ) (a b : BitVec w),
<fsm>.eval w [a, b] = BitStream.allOnes
```

6. Deciding FSM Denotation by k-induction

```
w : ℕ    a b : BitVec w
⊢ proven by <decide>
```

Fig. 5. The structure of the k induction proof by reflection. See that we go from the predicate, to the bitstream semantics, to the FSM transducer semantics, which is then proven by the k induction algorithm.

```
  | ...
```

We prove a theorem which shows that the stream semantics agrees with the bitvector semantics.

```
theorem Predicate.eval_eq_denote (w : Nat) (p : Predicate) (vars : List (BitVec w)) :
    p.denote w vars  ⟺  -- denotation into bitvectors.
    (p.eval vars w = false) -- bitstream semantics.
```

Next, the stream semantics of the predicate is made concrete by realizing the bitstream as the output of a finite state machine. First, we show that every finite state machine denotes an infinite bitstream:

```
/-- `eval p` morally gives the function
  `BitStream → ... → BitStream` represented by FSM `p` -/
def FiniteStateMachine.eval (x : arity → BitStream) : BitStream
```

Next, we define that a FSM implements a predicate p iff the stream of outputs from the FSM agrees with the denotation of the predicate p:

```
/-- An `FSMPredicateSolution `t` is an FSM with a witness
    that the FSM evaluates to the same value as `t` does -/
structure FSMPredicateSolution (p : Predicate) extends FSM (Fin p.arity) where
    good : p.eval = toFSM.eval
```

We then show that every predicate p has a corresponding FSM that implements it:

```
def predicateToFSM (p : Predicate) → FSMPredicateSolution p
  | .eq t₁ t₂ =>
    let t₁ := termEvalEqFSM t₁ -- Convert term syntax to FSM
    let t₂ := termEvalEqFSM t₂ -- Convert term syntax to FSM
    {
     -- Build equality FSM from term FSMs
     toFSM := fsmEq t₁.toFSM t₂.toFSM
     good := eval_fsmEq_eq_evalFin_Predicate_eq
    }
  | ... (Other predicates, elided)
```

With this machinery, we show that each predicate has an FSM associated to it. This FSM is built using constructs from our library, to build FSMs for terms, which are combined into FSMs for predicates. As a concrete example, consider the equality predicate: we build an FSM, which first outputs a true, since all bitvectors are equal at width 0, and then, for each width, checks if the bits are equal. However, this does not suffice, since given two bitvectors $x \equiv \langle 100\ldots\rangle$ and $y \equiv \langle 000\ldots\rangle$. such an FSM would output $\langle \text{true}, \text{false}, \text{true}, \text{true}, \ldots\rangle$. However, we need the FSM to output ⟨true, false, false, false, … ⟩ after the first bit, since the bitvectors being disequal at index 1 implies they are disequal at all widths. Thus, we *scan* the output of the FSM with boolean and to accumulate the first difference. In the mechanization, this is:

```
def fsmEq (a : FSM (Fin k)) (b : FSM (Fin l)) : FSM (Fin (k ⊔ l)) :=
  composeUnary FSM.scanAnd (  -- scan with && to accumulate 1st difference.
    (composeUnary (FSM.ls true)  -- At width 0, all BVs are equal.
     composeBinary FSM.eq a b  -- For other widths, check if bit is equal
    ))
```

Our verification library has support for composing unary and binary functions over FSMs, which we use as building blocks to build the composite FSMs for terms and predicates. The core building block, which allows a composition of n-ary FSMs is the compose function. Given an FSM p of arity n, a family of n FSMs q_i of arities m_i, and given yet another arity m such that $m_i \leq m$ for all i, we can compose p with the q_i yielding a single FSM of arity m, such that each FSM q_i computes the ith bit that is fed to the FSM p.

```
def compose (n : Type) (p : FSM n) -- FSM p of arity n
  (ms : n → Type) (q : ∀ (i : n), FSM (ms i)) -- family of FSMs q_i
  (m : Type) (f : ∀ (i : n), ms i → m) : -- m_i ≤ m
  FSM m -- output FSM
```

This building block is used to create the functions `composeUnary` and `composeBinary`, which take a unary or binary function and apply it to the FSMs. As seen above, we use `composeUnary` and `composeBinary` to build the FSM for the equality predicate, and more generally, to build the FSM for the predicate P as a composition of smaller FSMs. The library has common FSM constructors for (a) lifting bitwise operations on booleans, (b) concatenating bits to an existing FSM, (c) scanning the output of an FSM with a binary function, and (d) common arithmetic circuits built from these primitives.

Finally, we show that the FSM for the equality predicate is correct, by proving that the denotation of the FSM equals the bitstream denotation of the predicate:

```
/-- Evaluation FSM.eq is the same as evaluating Predicate.eq.evalFin. -/
theorem eval_fsmEq_eq_evalFin_Predicate_eq (t₁ t₂ : Term) :
   (fsmEq (termEvalEqFSM t₁).toFSM (termEvalEqFSM t₂).toFSM).eval =
   (Predicate.eq t₁ t₂).evalFin
```

See that this establishes a proof of correctness, which connects the semantics of the finite state machine as a transducer to the semantics of the term/predicate's denotation as a transducer.

7 Secrets of Robust Implementation: Preprocessing & Normalization of Inputs

The tactic front-end performs heavy preprocessing before performing reflection. This preprocessing is necessary to ensure that the input formula is in a normal form that our small reflective fragment can consume. We also use the preprocessing to extend the set of decidable predicates that can be handled by the decision procedure, to a more expressive, but logically equivalent fragment. We use Lean's simplifier, `simp` to drive our preprocessing. We register lemmas that simplify the input formula into a normal form.

Normalize literals. The constant bitvector ⟨10⟩ can be represented in many ways in Lean: (a) by the numeral `3`, which is implicitly cast into a bitvector, (b) by the integer `-1 : Int`, which is cast into a bitvector, (c) by the bitvector `BitVec.ofNat (w := 2) 3`, which is the explicit bitvector representation. and (d) by the bitvector expression `BitVec.ofInt (w := 2) (-1)`, which is the explicit integer representation. We coerce all of these into `BitVec.ofNat` as a normal form, since it has the most complete support for reasoning and constant folding currently in the proof assistant.

Normalize arithmetic. To help with computing normal forms, we perform the usual reassociations and commutations. First, we move constants to the left of expressions for better constant folding. This permits the following set of rewrites $1 + (x + 2) \longrightarrow 1 + (2 + x) \longrightarrow (1 + 2) + x \longrightarrow 3 + x$. Next, we unfold multiplication into a shift-and-add form, so that $x \times 2^l \cdot k$ is rewritten as $(x \ll l) * k$, and $x \times (2k + 1)$ is rewritten to $x * 2k + x$, and we rewrite $x \ll l \ll m \longrightarrow x \ll (l + m)$. These rewrites, taken together, break a multiplication up into repeated shifts-and-adds. For example, the rewrites normalize $x * 13 \longrightarrow x * 12 + x \longrightarrow (x \ll 2) * 3 + x \longrightarrow (x \ll 2) * 2 + (x \ll 2) + x \longrightarrow ((x \ll 2) \ll 1) + ((x \ll 2) + x) \longrightarrow x \ll 3 + x \ll 2 + x$ This allows the core decision procedure to only reason about shifts and adds.

Normalize predicates. Given a goal state of the form $H_1, H_2 \cdots \vdash P \longrightarrow Q$, we transform the goal into $H_1 \longrightarrow H_2 \longrightarrow \cdots \longrightarrow P \longrightarrow Q$ by using Lean's `revert` tactic to revert all hypotheses. Next, we convert the goal into $\vdash \neg H_1 \vee \neg H_2 \vee \cdots \vee \neg P \vee Q$. This is now in a form that can be converted to negation normal form. Next, we normalize the predicate into negation normal form, which pushes negation into the leaves, using the rewrites $\neg(P \wedge Q) \longrightarrow \neg P \vee \neg Q$, $\neg(P \vee Q) \longrightarrow \neg P \wedge \neg Q$, and $\neg\neg P \longrightarrow P$, and $\neg(P = Q) \longrightarrow P \neq Q$ (and similar negations for the other atomic predicates), which we are able to do since our atomic predicates are closed under negation. Next, $P = P_1 \wedge P_2$, one can trivially distribute the quantification over the conjunction, since $\forall x, P_1(x) \wedge P_2(x)$ is logically equivalent to $\forall x, P_1(x) \wedge \forall y, P_2(y)$. This allows us to solve smaller subproblems.

Specialized preprocessing for k-induction. The k-induction solver has a smaller core language, and it reduces unsigned and signed inequalities into properties that are checked on the msb of the inputs and outputs. See that $a <_u b$ if and only if upon computing $a - b$, the high borrow bit is 1. Therefore, one can build the stream of borrows from the subtraction circuit, and define $[\![a <_u b]\!]\,[i] \equiv \mathsf{subBorrow}(a, b)[i]$, where subBorrow builds the borrow circuit. Similarly, note that $a <_s b$ if and only if $(a - b).\mathsf{toInt} < 0$. At any finite bitwidth w, we have that $(a - b)|_w.\mathsf{toInt} < 0$ iff $(a - b)|_w.\mathsf{msb} = \mathsf{true}$. So we define $[\![a <_s b]\!] \equiv \neg(a - b)$, since the bit at index w of $\neg(a - b)$ is false iff the msbof $a - b$ is true, which implies that $(a - b).\mathsf{toInt}$ is negative.

Width Constraints. Furthermore, we can impose constraints on the width of the bitvector, by building bitstreams that start at 1 and become 0 after k steps, or vice versa. This encoding provides constraints of the form $w = k$, $w \geq k$, $w \leq k$, thereby embedding the finite with fragment into the infinite width theory. The idea is that the predicate $w = k$ corresponds to a bitstream $\langle 0 \ldots 010 \ldots \rangle$, which has a 1 at the index k, and 0 everywhere else. Thus, the predicate will be true at bitwidth k, and 0 everywhere else. Similarly, we encode $w \leq k$ as a bitstream that is 1 at all indices less than k, and 0 afterward, and $w \geq k$ as a bitstream that is 1 at all indices greater than k. These constraints are important to rule out vacuous cases which occur at width 0, and to prove special cases for small bitwidth. For example, the statement $\forall (w : \mathbb{N})\ (x\ y : \mathsf{BitVec}\ w), w = 1 \longrightarrow x + y = x \oplus y$ is only true at width 1.

Symbolic normalization. We run `bv_normalize`, a custom bitvector normalizer which rewrites bitvector expressions into simpler bitvector expressions, which produced smaller problems that are easier to then reflect into our core language.

8 A Specialized Solver for Mixed-Boolean-Arithmetic

We consider a restricted fragment of the larger bitvector theory, called as mixed-boolean-arithmetic [28] for which an efficient reduction to a decidable integer problem exists. This works on the fragment of the full theory where we allow linear combinations of *boolean* terms, where boolean terms are terms comprising of purely bitwise operations on atoms (Figure 6). This is an interesting fragment, which subsumes both ring properties and bitwise properties, and allows for mild mixtures of these properties.

$$\mathsf{A} \equiv \mathsf{Const} \mid \mathsf{Var} \quad \mathsf{B} \equiv \mathsf{A} \mid \mathsf{B} \boxplus \mathsf{B} \mid \sim \mathsf{B}, (\boxplus \in \{\|, \&\}) \quad \mathsf{T} \equiv \mathsf{B} \mid \mathsf{T} \otimes \mathsf{T}, (\otimes \in \{+, -\}) \quad \mathsf{P} \equiv (\mathsf{T} = 0).$$

Fig. 6. Syntax of the theory decided by the MBA solver.

The key idea of such a solver is to first prove MBA identities at the one-bit level, and to show that if they hold at the one-bit level, then they hold at all bitwidths. We sketch the idea with an example. Consider the equation $\forall\, w : \mathbb{N}\ (x\ y : \mathsf{BitVec}\ w), x + y - 2(x\&y) - (x \oplus y) =_{\mathsf{BitVec}\ w} 0$. First, note that to show this equation, it suffices to show the equation interpreted *over the integers*, with $\forall (w : \mathbb{N})\ (x\ y : \mathsf{BitVec}\ w), [x]_{\mathbb{Z}} + [y]_{\mathbb{Z}} - 2([x]_{\mathbb{Z}} \& [y]_{\mathbb{Z}}) - ([x]_{\mathbb{Z}} \oplus [y]_{\mathbb{Z}}) =_{\mathbb{Z}} 0$, where $[\,\cdot\,]_{\mathbb{Z}} : \mathsf{BitVec}\ w \longrightarrow \mathbb{Z}$ is the interpretation of the bitvector as a 2s complement *unsigned* integer. Intuitively, this is true because an integer is an infinite width bitvector, and the equation holds for all bitwidths if it holds for the infinite width bitvector. To prove the example, we induct the width w. For $w = 1$, we prove the equation by exhaustive enumeration:

x	y	$[x]_{\mathbb{Z}} + [y]_{\mathbb{Z}} - 2[x]_{\mathbb{Z}} \& [y]_{\mathbb{Z}} - [x]_{\mathbb{Z}} \oplus [y]_{\mathbb{Z}}$
0	0	$0 + 0 - 2 \cdot (0) - 0 = 0$
0	1	$0 + 1 - 2 \cdot (0) - 1 = 0$
1	0	$1 + 0 - 2 \cdot (0) - 1 = 0$
1	1	$1 + 1 - 2 \cdot (1) - 0 = 0$

Which shows that the equation holds for $w = 1$. Next, assume it holds for $w = n$, and for $w = n+1$, use the key property that $[\,\cdot\,]_{\mathbb{Z}}$ *distributes* over bit concatenation and bit operations:

$$[\,x : xs\,]_{\mathbb{Z}} \& [\,y : ys\,]_{\mathbb{Z}} = [\,x \& y : xs \& ys\,]_{\mathbb{Z}},$$

and similarly for all other bitwise operations $\boxplus$. Also, we compute $[\,x : xs\,]_{\mathbb{Z}}$ as $2^n \cdot [\,x\,]_{\mathbb{Z}} + [\,xs\,]_{\mathbb{Z}}$. This allows us to perform the following sequence of rewrites on the goal of the induction, to show that at width $n + 1$, the equation is zero:

$$\begin{aligned}
&[\,x : xs\,]_{\mathbb{Z}} + [\,y : ys\,]_{\mathbb{Z}} - 2[\,x : xs\,]_{\mathbb{Z}} \& [\,y : ys\,]_{\mathbb{Z}} - [\,x : xs\,]_{\mathbb{Z}} \oplus [\,y : ys\,]_{\mathbb{Z}} \\
&= [\,x : xs\,]_{\mathbb{Z}} + [\,y : ys\,]_{\mathbb{Z}} - 2[\,x \& y : xs \& ys\,]_{\mathbb{Z}} - [\,x \oplus y : xs \oplus ys\,]_{\mathbb{Z}} \text{ (distribute)} \\
&= (2^n \cdot [\,x\,]_{\mathbb{Z}} + [\,xs\,]_{\mathbb{Z}}) + (2^n \cdot [\,y\,]_{\mathbb{Z}} + [\,ys\,]_{\mathbb{Z}}) - \\
&\qquad 2(2^n \cdot [\,x \& y\,]_{\mathbb{Z}} + [\,xs \& ys\,]_{\mathbb{Z}}) - (2^n \cdot [\,x \oplus y\,]_{\mathbb{Z}} + [\,xs \oplus ys\,]_{\mathbb{Z}}) \text{ (evaluate)} \\
&= 2^n \cdot ([\,x\,]_{\mathbb{Z}} + [\,y\,]_{\mathbb{Z}} - 2[\,x \& y\,]_{\mathbb{Z}} - [\,x \oplus y\,]_{\mathbb{Z}}) + \\
&\quad ([\,xs\,]_{\mathbb{Z}} + [\,ys\,]_{\mathbb{Z}} - 2[\,xs \& ys\,]_{\mathbb{Z}} - [\,xs \oplus ys\,]_{\mathbb{Z}}) \text{ (regroup)} \\
&= 2^n \cdot 0 \text{ (by theorem for width 1)} + 0 \text{ (by IH)} = 0,
\end{aligned}$$

which proves the identity for all bitwidths.

8.1 Mechanization in Lean

In our mechanization, we use the above procedure, appropriately generalized for all boolean operations $\boxplus$, to check that the identity holds at width 1 by exhaustive enumeration, as the table does. We use proof by reflection, and we reflect the equation into a term language:

```
inductive Factor
| var (n : Nat) -- Variable index
| and (x y : Factor) -- Bitwise &
| ...
| not (x : Factor) -- Bitwise !

structure Term where
  c : Int
  f : Factor

def Eqn := List Term
```

These define a factor, which is a bitwise expression or a variable, terms, which are factors scaled by a constant integer, and equations, which are terms which are added up to create the final expression. We first run a preprocessing pass, which normalizes inputs to be of the form $\sum_i c_i f_i$, where c_i is a constant integer and f_i is the fragment of `Factor`. Example rewrites include translating $-(x + y) \longrightarrow (-x) + (-y)$, and $-x \longrightarrow \langle 0 \rangle - x$. Once we have reflected our expressions, we use the key theorem:

```
theorem Eqn.forall_width_reflect_zero_of_width_one_denote_zero
    (e : Eqn) (w : Nat) (env : List (BitVec w))
    (h : (∀ env1 : EnvFin 1 e.numVars, Eqn.denoteFin e env1 = 0)) :
    Eqn.reflect eqn env = BitVec.ofInt w 0
```

which states that to establish, for all widths, that the equation `eqn` is equal to zero, it suffices to establish this for all a one-bit environments (as stated by hypothesis `h`). The proof of this theorem follows along the lines of the example, with the reasoning generalized to regroup any linear combination. This establishes the identity for all bitwidths. Furthermore, the algorithm, while subtle, is algorithmically simple to implement, taking only 800 lines of code with no external dependencies other than the Lean language's core library.

8.2 Algorithmic Extension to (Un)Signed Inequalities

We contribute a novel extension of the core MBA philosophy of extending 1-bit identities to all bitwidths, and show that this can be used to extend the equational theory to the inequational theory. The key tool will be to exploit the assumption that signed and unsigned overflow do not occur. Compiler IRs such as LLVM express this assumption by being able to define signed (and unsigned) overflow as *undefined behaviour*. This is necessary, for example, to express the C language semantics, where integer overflow is undefined behavior, and in LLVM IR, this is expressed using the `nsw` and `nuw` flags (no (un)signed wrap) on arithmetic operations. If we add the assumption that overflow does not occur, then it is indeed the case that $[\![x + y]\!]_Z = [\![x]\!]_Z + [\![y]\!]_Z$, which enables us to prove bounds on the integer values of the bitvectors, and to have these bounds transfer to the bitvectors themselves.

For example, consider the expression $(x \| y) \geq_u x$. We can show that this is true by showing that $(x \| y) - x \geq 0$ is true for $w = 1$ by exhaustive enumeration, and by a similar regrouping argument as for equalities, show that it holds for all bitwidths.

9 Evaluation

To show that our MBA decision procedure is complete for its fragment, and to show that our width-independent decision procedures are able to prove real-world identities from compiler development, we evaluate our decision procedures quantitatively, by measuring the number of problems solved, as well as with survival plots [12], to characterize performance trade-offs of the different solvers.

Solver	Hacker's Del (31)	Alive (104)	MBA-Blast (2500)	`InstCombine` (7978)
Width Indep.				
`k-ind (cadical)`	31	54	1546	**2498**
`automata`	31	54	2500	2138
Fixed Width				
`bv_decide` ($w = 64$)	31	54	2500	7904

Fig. 7. Both k-induction and automata based solvers are competitive, and solve similar fragments. We provide a fixed-width comparison (which **does not solve** the infinite width fragment) as a coarse measure of the complexity of the problems.

Hacker's Delight (31 problems). For a stock of bit-twiddling algorithms, we take the first two chapters of Hacker's Delight [45], we take the problems that have been translated into Lean (by `lean-mlir` [2]).

Alive (104 problems). We take the test suite of Alive [29], which is a translation validator for LLVM IR. We use problems on addition, subtraction, and bitwise operations which have been translated into Lean (by `lean-mlir` [2]), and evaluate our algorithms on this dataset.

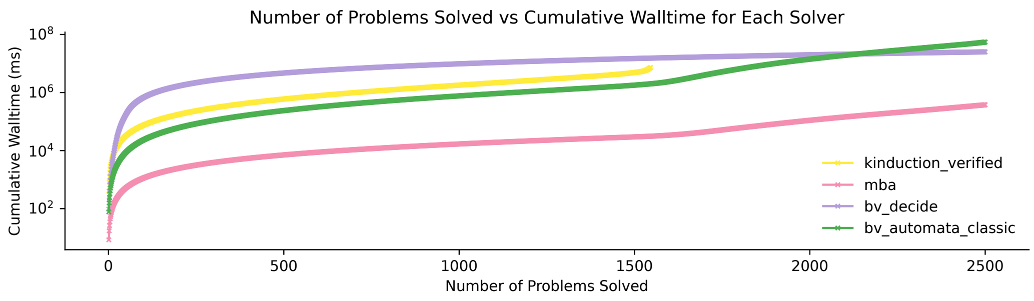
InstCombine (7978 problems). We use `InstCombine` a dataset of 7978 bitvector lemmas, which are generated from a dataset of 4556 peephole rewrites from LLVM's peephole optimizer, by reducing the validity of rewrites on LLVM values to equalities of bitvectors. Note that these problems are for fixed bitwidths. Thus, we can run `bv_decide` on these problems. To run our decision procedures on this dataset, we treat the fixed width problem as an infinite width problem, and run our decision procedures on them. This can introduce spurious counterexamples, such as the optimization $x + y = x \oplus y$, which untrue for bit-width greater than 1. Regardless, we are interested in the performance of our decision procedures to prove real-world rewrites, and thus this is a valuable dataset.

9.1 Choosing the Right Solver for the Right Problem

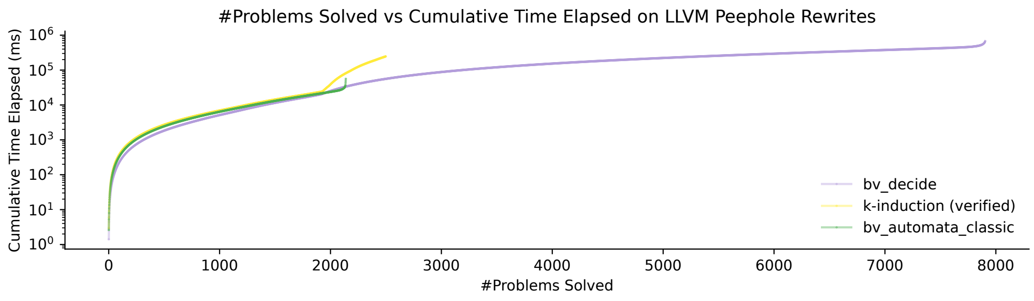
In order to understand the trade-offs of the different solvers, we plot the number of problems solved against cumulative wall-clock time, with problems sorted in ascending order of wall-clock time. Such a survival plot [12] allows one to quickly read off the total number of problem solved, as well as the behavior of the different solvers given a fixed amount of time.

MBA (2500 problems). Out of the 2500 problems (Figure 8a), the k-induction solver solves 1546 problems out of 2500, while the other solvers solve 2500 problems. The k-induction solver builds circuits that are quadratic in size, due to the need to build a circuit that forbids loops, ensuring that all paths are simple paths. This is not too surprising, as the MBA dataset was designed to be a fragment used for obfuscation, which is hard for SAT solvers to solve. The NFA-based solver, which takes 15h13m total time, is in the same order of magnitude as the bitblaster (where the bitblaster solves for fixed $w = 64$, which we provide as a comparison, but *does not* solve the infinite width problem), where the bitblaster takes 6h59m total time. The MBA solver is the fastest, with a geomean time of 54.2ms, which is an order of magnitude faster incomparison to both k-induction (2.7s) and the NFA-based solver (4.26s).

The solvers introduced in this paper, and especially the automata-based one which executes entirely inside of Lean, would run faster if native compilation was enabled (by a factor of at least two for the automata-base one, from experiments with earlier versions of Mathlib and Lean). However, native compilation does not work with the recent versions of Mathlib and Lean we tested with, and so our decision procedures are run in interpreted mode in these benchmarks.



(a) Survival plot on the MBA-blast dataset. See that the specialized MBA solver is an order of magnitude faster than both k-induction and the NFA-based solver, this accurately captures the trade-offs available in this space, where specialized solvers are fastest, and that consuming certificates from fast, external, unverified solvers effectively does scales to moderate problem sizes.



(b) Survival plot on the InstCombine (LLVM Peephole Rewrites) dataset. We treat fixed-width problems as infinite width problems to be solved by our k-induction and automata solvers. Since the problem sizes are small, the lean-based automata solver is faster than the k-induction solver which repeatedly invokes CaDiCaL for a proof.

Fig. 8. Survival plots for the MBA-blast and InstCombine datasets.

InstCombine (7978 problems). In the 7978 problems (Figure 8b), `bv_decide` is effective, since the problem is always fixed to a particular bitwidth, and solves 7904 problems, a large majority. k-induction solves 2498 problems (the largest amongst the symbolic width solvers), and the automata solver solves 2138 problems. Interestingly, the automata solver (with preprocessing) is faster than the k-induction solver (with preprocessing) for the LLVM dataset.[1] The geomean time for the automata solver is 11ms, while the geomean time for the k-induction solver is 24ms. In contrast, bit-blasting is slower, with a geomean time of 45ms. Overall, this shows us that the fragments that our arbitrary-width decision procedures tackle occur in a non-trivial fraction of the real-world problems from the LLVM ecosystem, and that our algorithmically complete procedures for this fragment is a useful addition to the Lean ecosystem.

Root causes of LLVM peephole failures. We catalog the failures on the LLVM peephole optimizer dataset, by logging all errors emitted from the preprocessor for reflection, and manually inspect the failures, using regular expressions to categorize them. We find the following categories: (1)

[1]We conjecture that this is because the problems are small and the overhead of invoking a SAT solver is larger than the time it takes to solve the problem.

Rewrites which ought to hold, but are classified as untrue (447 problems). These occur since we translate fixed-width problems into infinite width-problems, and such example use constants such as 7, which should be generalized to e.g. $2^w - 1$; a promising path would be to use tools such as Hydra [34] to automatically do the generalization. (2) Rewrites that involve quantification over multiple widths $w_1, \ldots, w_k$, rather than a fixed width (1925 problems). These rewrites then perform truncation / sign extension of the bitvectors involved. (3) Theorems whose goal state lie outside our fragment (463 problems). This includes two kinds of theorems: (3a) those with operations that are truly not expressible in our fragment, such as signed division, and (3b) those with operations that need further reduction to be expressible in the fragment, such as division-by-constant. (4) Theorems that involve boolean equalities that lie outside our fragment (565). For example, a statement such as `x : BitVec w ⊢ (255#w <`$_u$` x || 255#w <`$_s$` x) = (255#w <`$_u$` x)` uses the boolean fragment with boolean or, instead of being expressed as a proposition. This requires further preprocessing, using the newly developed `bool_to_prop` tactic in Lean, which we plan to integrate into our decision procedures. (5) Unknown boolean disequality (81 problems), which establish disequality between operations such as `msb` which are not supported by our preprocessor. As many of these issues are not fundamental problems with our approach, we believe our decision procedures will soon be able to prove substantially more test cases.

10 Related Work

Bitwidth Independent Boolean Circuits. Bitwidth independent circuits were introduced by Pichora [37]. Algorithms for deciding this fragment based on unification [5], families of boolean circuits to establish base cases and inductive cases [21, 22] have been explored in the literature.

Automata-Based Presburger Arithmetic Solvers. The idea that a Presburger formula can be represented by a finite-state automaton is classical [13]. Moreover, it can be seen that Presburger arithmetic is expressible into WS1S (weak monadic second order logic with one successor), which can also be decided via automata. Schuele and Schneider [42] use similar techniques in an uncertified context to decide Presburger arithmetic with bitwise operations: a formula is translated to an alternating automaton, whose emptiness is checked using k-induction and a SAT solver. While they mostly focus on unbounded natural numbers with bitwise operations rather than bitvectors (and do not support left shifts, for example), their high-level approach is similar to ours.

Model Checking Finite Transition Systems. Bounded Model Checking allows properties to be proved for upto n states by unrolling the model. While the method is incomplete, the power of SAT solving meant that BMC gained popularity in the 1990s, when SAT solvers started outperforming BDDs [4]. *k-Induction* [43] establishes that if the past k states are good, then the $k+1$th state is also good, for all states. *Interpolation based methods* instead try to prove the safety property by finding a stronger property P such that it (i) implies the safety property, and (ii) if it holds for the previous k states, then it also holds for the next state. Such properties are discovered by computing Craig interpolants [16]. Such interpolants can be extracted from resolution proofs in a SAT solver [33]. *IC3* [11] and related algorithms all produce a sequence of lemmas, which are inductive relative to previous lemmas and stepwise assumptions. At convergence, a subset of the generated lemmas imply the safety property to be proven. Model checkers such as Pono [31], AVR [19] can use k-induction, IC3, and interpolation to establish inductive invariants.

Mechanized Model Checking Algorithms & Finite State Machine Libraries. The cava LTL model checker [20] is formally verified in Isabelle, and mechanizes executable Büchi automata. This contrasts to our work, where we mechanize *finite* automata, and consume finite traces, instead of infinite traces. More importantly, our decision procedures, in addition to being verified, can be used internally to the proof assistant in the form of tactics and generate proofs without additional assumptions. Pous [38] developed a tactic in Coq to decide statements in the theory of Kleene Algebras with Tests (KAT) by translating from to an automaton and checking for language inclusion. In contrast to this work, the translation from KAT, which is a variant of regular expressions, to automata is more direct, and, as the statements being decided were very small, the automata library is not designed for performance: for example, the transitions are represented as closures which grow with the numnber of operations. The ACL2 proof assistant internally builds finite state machines from its Symbolic Vector Expression (SVEX) [24], which is associated to each signal in a Verilog module. These finite state machines are typically used for symbolic simulation and (bounded) model checking. In contrast, we use our finite automata for decision procedures for bitvector arithmetic, and we are thus interested in the *languages* that our automata consume. Previous work has abstractly mechanized k-induction [25], but does not extract executable tactics from their mechanization, as their mechanization is entirely abstract with no implementation.

Finite Automata Libraries. Mata [15] is a high-performance automata library implemented in C++, optimized for fast manipulation and analysis of finite automata. It supports efficient automata-theoretic operations, Mona [23] is a C-based tool focused on deterministic finite automata, and is tailored for logics like WS1S. Mona, like our approach, compiles logical specifications into minimal DFAs. LASH [7] combines automata with arithmetic reasoning, leveraging automata-based representations to solve Presburger arithmetic formulas. In contrast to our work, none of the above libraries are mechanized, and none are directly integrated into a proof assistant.

Mixed Boolean Arithmetic Solvers. The line of work that builds upon the MBA-Blast algorithm [28], which essentially analyzes 1-bit patterns and extrapolates them to n-bits to prove MBA identities. This was improved upon by SiMBA [39] that can deobfuscate all linear MBAs. These algorithms are specialized, and prove identities in the fragment where we have a linear combination of terms, where each term is bitwise operations applied to the bitvector atoms. To our knowledge, our extension of this algorithm to solve the full horn-clause fragment of the theory is novel, and our work is the first mechanization of these algorithms. Our modified presentation of the algorithm in terms of linear algebra is also novel, to the best of our knowledge.

11 Conclusion

We fill the gap in ITP proof automation for width-independent bitvector reasoning through verified decision procedures: a k-induction-based model checker, an automata-theoretic solver, and an MBA-Blast extension handling mixed arithmetic-bitwise predicates. These tools solve 100% of Hacker's Delight benchmarks and 27% of LLVM peephole rewrites, with the MBA solver achieving 54.2ms geomean runtime on 2500 problems. By grounding width-independent reasoning in mechanized proofs, and providing sound and complete decision procedures for fragments of bitvector reasoning, we establish Lean as a platform for trustworthy, scalable, width-independent automation across cryptography, hardware, and verified compilers.

Data Availability Statement

The data that support the findings of this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.16269885[3].

Acknowledgments

We thank Alex Keizer and Atticus Kuhn for the initial version of the tactic interface, We thank Sebastien Michelland & Henrik Böving for discussions of SAT solver interaction with metaprogramming, and the Lean FRO for offering powerful software foundations with and around Lean.

We acknowledge support from the Ethereum Foundation, Horizon Europe (Grant 101070375 - CONVOLVE), and Amazon via an Amazon Research Award.

References

[1] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E Gray, Robert M Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, et al. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–31.

[2] Siddharth Bhat, Alex Keizer, Chris Hughes, Andrés Goens, and Tobias Grosser. 2024. Verifying Peephole Rewriting In SSA Compiler IRs. *arXiv preprint arXiv:2407.03685* (2024).

[3] Siddharth Bhat and Léo Stefanesco. 2025. *Certified Decision Procedures for Width- Independent Bitvector Predicates in Interactive Theorem Provers.* https://doi.org/10.5281/zenodo.16269885

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *TACAS*. Springer, 193–207.

[5] Nikolaj S Bjørner and Mark C Pichora. 1998. Deciding fixed and non-fixed size bit-vectors. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 376–392.

[6] Achim Blumensath and Erich Grädel. 2000. Automatic Structures. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 51–62. https://doi.org/10.1109/LICS.2000.855755

[7] Bernard Boigelot and Louis Latour. 2004. Counting the solutions of Presburger equations without enumerating them. *Theoretical Computer Science* 313, 1 (2004), 17–29.

[8] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. 2013. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *2013 IEEE 21st Symposium on Computer Arithmetic*. 107–115. https://doi.org/10.1109/ARITH.2013.30

[9] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*. IEEE, 243–252.

[10] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 243–257.

[11] Aaron R Bradley. 2011. SAT-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 70–87.

[12] Martin Brain, James H Davenport, and Alberto Griggio. 2017. Benchmarking Solvers, SAT-style.. In *SC^2 @ ISSAC*.

[13] J Richakd Büchi. 1990. Weak second-order arithmetic and finite automata. In *The Collected Works of J. Richard Büchi*. Springer, 398–424.

[14] J. Richard Büchi. 1960. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly* 6, 1-6 (1960), 66–92. https://doi.org/10.1002/malq.19600060105

[15] David Chocholatỳ, Tomáš Fiedor, Vojtěch Havlena, Lukáš Holík, Martin Hruška, Ondřej Lengál, and Juraj Síč. 2024. Mata: A fast and simple finite automata library. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 130–151.

[16] William Craig. 1957. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic* 22, 3 (1957), 250–268.

[17] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *International Conference on Automated Deduction*. Springer, 625–635.

[18] J. Esparza and M. Blondin. 2023. *Automata Theory: An Algorithmic Approach*. MIT Press. https://books.google.de/books?id=SP2nEAAAQBAJ

[19] Aman Goel and Karem Sakallah. 2020. AVR: abstractly verifying reachability. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 413–422.

[20] Benjamin Grégoire and Assia Mahboubi. 2005. Proving equalities in a commutative ring done right in Coq. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 98–113.

[21] Aarti Gupta and Allan L Fisher. 1993. Parametric circuit representation using inductive boolean functions. In *Computer Aided Verification: 5th International Conference, CAV'93 Elounda, Greece, June 28–July 1, 1993 Proceedings 5*. Springer, 15–28.

[22] Aarti Gupta and Allan L Fisher. 1993. Representation and symbolic manipulation of linearly inductive boolean functions. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE, 192–199.

[23] Jesper G Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. 1995. Mona: Monadic second-order logic in practice. In *TACAS*. Springer, 89–110.

[24] Warren A Hunt Jr, Matt Kaufmann, J Strother Moore, and Anna Slobodova. 2017. Industrial hardware and software verification with ACL2. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 20150399.

[25] Daisuke Ishii and Saito Fujii. 2020. Formalizing the soundness of the encoding methods of SAT-based model checking. In *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 105–112.

[26] Dexter Kozen. 1997. *Automata and computability.* Springer.

[27] Gengwang Li, Min Yu, Dongliang Fang, Gang Li, Xiang Meng, Jiangguo Jiang, and Weiqing Huang. 2024. X-MBA: Towards Heterogeneous Mixed Boolean-Arithmetic Deobfuscation. In *MILCOM 2024-2024 IEEE Military Communications Conference (MILCOM)*. IEEE, 1082–1087.

[28] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. 2021. {MBA-Blast}: Unveiling and Simplifying Mixed {Boolean-Arithmetic} Obfuscation. In *30th USENIX Security Symposium (USENIX Security 21)*. 1701–1718.

[29] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 22–32.

[30] Nuno P Lopes and John Regehr. 2018. Future Directions for Optimizing Compilers. *arXiv preprint arXiv:1809.02161* (2018).

[31] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark Barrett. 2021. Pono: a flexible and extensible SMT-based model checker. In *International Conference on Computer Aided Verification*. Springer, 461–474.

[32] The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. https://doi.org/10.1145/3372885.3373824

[33] Kenneth L McMillan. 2003. Interpolation and SAT-based model checking. In *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*. Springer, 1–13.

[34] Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing peephole optimizations with program synthesis. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 725–753.

[35] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 3–17.

[36] Aina Niemetz, Mathias Preiner, and Yoni Zohar. 2024. Scalable bit-blasting with abstractions. In *International Conference on Computer Aided Verification*. Springer, 178–200.

[37] Mark Christopher Pichora. 2003. *Automated reasoning about hardware data types using bit-vectors of symbolic lengths.* University of Toronto.

[38] Damien Pous. 2013. Kleene Algebra with Tests and Coq Tools for while Programs. In *ITP (Lecture Notes in Computer Science, Vol. 7998)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer, 180–196. https://doi.org/10.1007/978-3-642-39634-2_15

[39] Benjamin Reichenwallner and Peter Meerwald-Stadler. 2022. Efficient deobfuscation of linear mixed boolean-arithmetic expressions. In *Proceedings of the 2022 ACM Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks*. 19–28.

[40] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 161–168.

[41] Klaus Schneider, Jimmy Shabolt, and John G Taylor. 2004. *Verification of reactive systems: formal methods and algorithms.* Springer.

[42] Tobias Schuele and Klaus Schneider. 2007. Verification of Data Paths Using Unbounded Integers: Automata Strike Back. In *Hardware and Software, Verification and Testing*, Eyal Bin, Avi Ziv, and Shmuel Ur (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 65–80.

[43] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking safety properties using induction and a SAT-solver. In *International conference on formal methods in computer-aided design*. Springer, 127–144.

[44] The Coq Development Team. 2022. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.1003420

[45] Henry S Warren. 2013. *Hacker's delight.* Pearson Education.

[46] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 427–440.

[47] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. 2007. Information hiding in software with mixed boolean-arithmetic transforms. In *International Workshop on Information Security Applications*. Springer, 61–75.

Received 2025-03-25; accepted 2025-08-12