Solving Linear Arithmetic with SAT-based Model Checking

Yakir Vizel
Princeton University, USA

Alexander Nadel
Intel Development Center, Haifa, Israel

Sharad Malik
Princeton University, USA

Abstract—We present LIAMC, a novel decision procedure for (quantifier-free) linear arithmetic over both integers modulo \(2^N\) (LIA\(_N\)) and integers (LIA).

There is no need to explain our motivation to design a new efficient decision procedure for the widely used LIA logic. A LIA\(_N\) decision procedure can be extremely useful in the context of software (SW) verification. SW verification usually requires to reason about arithmetic constraints over finite integers. To that end, modern SW verification tools commonly use fixed-width bit-vector (BV) solvers. However, BV solvers’ efficiency drops dramatically as the width increases. To solve the performance problem, LIA solvers are applied, but they are imprecise as they cannot handle integer overflow. An efficient LIA\(_N\) solver would be the ideal solution in this context.

Our decision procedure LIAMC is based on a transformation of linear arithmetic into safety verification. We treat integers as unbounded streams of bits over time. More precisely, for each input integer, the least significant bit (LSB) corresponds to time 0 in the corresponding stream, and the \(k\)-th bit corresponds to the bit received at time \(k\). LIAMC then uses SAT-based model checking (SATMC) to solve the resulting problem. In order to achieve efficiency, LIAMC uses two forms of generalization. First, if it finds a formula to be unsatisfiable for width \(N\), it tries to generalize this result for all the widths. Second, if LIAMC finds a formula to be satisfiable for width \(N\), it tries to “extend” and thus generalize the assignment to a wider target width.

To evaluate LIAMC we used the QF_LIA subset of SMT-COMP’16, and ran two sets of experiments. First, we reinterpreted the QF_LIA over fixed-width bit-vectors of varying widths and compared LIAMC in LIA\(_N\) mode to both Boolector and Z3. LIAMC solved the most satisfiable instances out of the three even for the shortest width 32. Second, we compared LIAMC to CVC4 and Z3 on the original QF_LIA benchmarks. LIAMC was able to solve many instances that had not been solved by the other solvers.

I. INTRODUCTION

Nowadays, Satisfiability Modulo Theory (SMT) [1] solvers for the quantifier-free linear integer arithmetic (LIA) [2] logic are widely used, and have become highly efficient. Despite their efficiency, there is a growing demand for SMT solvers that can efficiently solve quantifier-free linear arithmetic over integers modulo \(2^N\) (LIA\(_N\)). This paper presents a novel decision procedure, LIAMC, suitable for solving LIA\(_N\) and arbitrary LIA instances. Our motivation for designing a decision procedure for LIA\(_N\) originates in software (SW) verification.

Formal verification of SW is one of the main driving forces in SMT research. SW verification usually involves reasoning about arithmetic constraints, and in particular, linear arithmetic constraints over integers modulo \(2^N\) for some \(N \in \mathbb{N}\). This is due to the fact that SW uses a finite representation for integers. More precisely, arithmetic operations over integers are interpreted over the ring \(\mathbb{Z}/2^N\mathbb{Z}\) (“machine arithmetic”) rather than over the ring \(\mathbb{Z}\). As a result, efficient bit-precise reasoning is highly desired.

In order to capture the semantics of linear arithmetic over \(\mathbb{Z}/2^N\mathbb{Z}\) (LIA\(_N\)), SMT solvers for the theory of fixed-width bit-vectors (BV solvers) are often used, since LIA\(_N\) is a proper subset of QF_BV. BV solvers, however, are not efficient when the bit-vectors are wide. Namely, when the value of \(2^N\) is large (e.g. \(N = 512\)), solving formulas in LIA\(_N\) becomes intractable for BV solvers. This inefficiency is mainly due to the way BV solvers are implemented: in most cases, the formula is reduced to a propositional formula using bit-blasting. Therefore, as \(N\) increases, so does the complexity of the resulting SAT formula. One way to overcome this inefficiency is by applying a LIA solver. Unlike BV solvers, LIA solvers reason about linear arithmetic over \(\mathbb{Z}\). While LIA solvers are more efficient than that of BV solvers for this task, they are less precise. This imprecision comes from the different semantics between LIA and LIA\(_N\). Namely, arithmetic operations over \(\mathbb{Z}\) cannot result in an “overflow” (i.e. wrap-around). In the context of SW verification, this may lead to unsound results. Hence, an efficient LIA\(_N\) solver, as presented in this paper, should be extremely useful for SW verification.

Our novel decision procedure for LIA\(_N\) and LIA, LIAMC, is based on a reduction of the input formula to a safety verification problem. Namely, a formula \(\varphi\) in either LIA\(_N\) or LIA is transformed to a transition system \(T\) such that the satisfiability of \(\varphi\) corresponds to whether \(T\) is SAFE or UNSAFE. The reduction treats integers as unbounded streams of bits over time. More precisely, for each input integer, the least significant bit (LSB) corresponds to time 0 in the corresponding stream, and the \(k\)-th bit corresponds to the bit received at time \(k\). The structure of \(T\) captures the constraints between the integer variables that appear in \(\varphi\). To determine if \(T\) is SAFE or UNSAFE, LIAMC uses SAT-based model checking (SATMC) [3].

One possible way to reason about \(T\) is by using Bounded Model Checking (BMC) [4], an efficient SATMC algorithm that can show \(T\) is UNSAFE. Considering our reduction, if BMC finds a counterexample of length \(N\) in \(T\) (\(T\) is UNSAFE), then \(\varphi\) is satisfiable over \(\mathbb{Z}/2^N\mathbb{Z}\). If no counterexample of length \(N\) exists in \(T\), then \(\varphi\) is unsatisfiable over \(\mathbb{Z}/2^N\mathbb{Z}\). This can be used as a decision procedure for LIA\(_N\). However, the performance of such an approach is usually not better then that of BV solvers [5]. BMC can either find a counterexample of length \(N\), or prove that counterexample of length \(N\) does
not exist. In that sense, in the context of LIAMC, it can only reason about LIA_N for a given N. In fact, this approach is somewhat “equivalent” to how modern eager BV solvers are implemented.

Unlike BMC, modern SATMC algorithms [6]–[8] use generalization in order to show that no counterexample, of any length, exists, and by that they can prove a transition system is SAFE. LIAMC takes advantage of this generalization mechanism. In case LIAMC finds φ to be unsatisfiable over \( \mathbb{Z}/2^N\mathbb{Z} \), SATMC’s generalization mechanism is applied to show φ is unsatisfiable over \( \mathbb{Z}/2^{N+1}\mathbb{Z} \) for every \( N < k \) and moreover, unsatisfiable over the integers. For the case a counterexample of length \( k \) is found, we have implemented an efficient procedure in LIAMC that tries to extend the counterexample to some target \( N \) (where \( N > k \)) and by that show φ is satisfiable over \( \mathbb{Z}/2^N\mathbb{Z} \). In addition, LIAMC can also extend a counterexample over \( \mathbb{Z}/2^N\mathbb{Z} \) to a counterexample over \( \mathbb{Z} \).

We evaluated our approach on QF_LIA subset of the SMT-COMP’16 benchmark. Since LIAMC can be used for both LIA and LIA_N, we used two sets of experiments. For our first set of experiments we translated QF_LIA benchmarks to QF_BV using fixed-width bit-vectors of sizes 32, 64, and 128. We then compared LIAMC to Boolestor [9], and Z3 [10], LIAMC solved the most satisfiable instances out of the three, even for a width as low as 32. For our second set of experiments we used the LIA solvers in CVC4 [11] and Z3, and compared LIAMC against them on QF_LIA_N. Here too, LIAMC was able to solve instances that were not solved by the other solvers.

II. PRELIMINARIES

In this section, we present notations and background that is required for the description of LIAMC.

A. Linear Integer Arithmetic

We consider First Order Logic modulo the theory of quantifier free Linear Arithmetic either over Integers (QF_LIA) or over Integers modulo a constant \( 2^N \) (QF_LIA_N). In what follows we denote QF_LIA and QF_LIA_N as LIA and LIA_N, respectively. The following grammar is used to define this theory:

\[
\begin{align*}
\varphi &::= \text{true} \mid \text{false} \mid p \mid \neg \varphi \mid \varphi \lor \varphi \mid \varphi \land \varphi \mid \text{term} \lor \text{term} \mid \text{term} \land \text{term} \mid \text{term} \\
\text{term} &::= c \mid x \mid \text{term} + \text{term} \mid \text{term} - \text{term} \mid \text{term} \times \text{term} \\
\text{ite}(\varphi, \text{term}, \text{term})
\end{align*}
\]

where \( \in \{=, <, \leq, >, \geq \} \), \( c \) and \( x \) are a constant symbol and a variable either over \( \mathbb{Z}/2^N\mathbb{Z} \) or \( \mathbb{Z} \), respectively.

Let \( \varphi \) be a formula in LIA_N (or LIA) over a set of variables \( \mathcal{V} \). \( \mathcal{V} := \mathcal{V_I} \cup \mathcal{V_B} \) where \( \mathcal{V_I} \) and \( \mathcal{V_B} \) are the sets of Integer and Boolean variables, respectively. Abusing notation, we write \( c \in \varphi \) for a constant integer \( c \) appearing in \( \varphi \). Let \( N \in \mathbb{N} \) be a natural number such that \( N \geq 2 \). We refer to the interpretation of \( \varphi \) over the ring \( \mathbb{Z}/2^N\mathbb{Z} \) as \( \varphi|_N \). For consistency, we use either \( \varphi \) or \( \varphi|_\infty \) as the interpretation of \( \varphi \) over \( \mathbb{Z} \).

Note that the semantics of linear arithmetic over \( \mathbb{Z} \) and over \( \mathbb{Z}/2^N\mathbb{Z} \) is different. This is mainly due to “overflow”. As an example, consider the following formula:

\[
\varphi := (z = x + y) \land (x > 0) \land (y > 0) \land (z < 0)
\]

While this formula is unsatisfiable over the ring \( \mathbb{Z} \), it is satisfiable over \( \mathbb{Z}/2^N\mathbb{Z} \). For example, for \( \mathbb{Z}/4\mathbb{Z} = \{-2, -1, 0, 1\} \), \( x = 1, y = 1 \) and \( z = -2 \) is a satisfying assignment.

B. Integers as Bit-Vectors

Integers can be represented using bit-vectors. In this work, we use the 2’s complement representation. Given an integer \( c \in \mathbb{Z} \), there exists \( N > 0 \) s.t. for every \( k \geq N \), there exists a bit-vector \( b = (b_{k-1}, \ldots, b_0) \) of size \( k \), and the following holds:

\[
c = -b_{k-1} \cdot 2^{k-1} + \sum_{i=0}^{k-2} b_i \cdot 2^i
\]

Note that a constant \( c \in \mathbb{Z}/2^N\mathbb{Z} \) for some \( N \geq 2 \) can be represented by a bit-vector of size \( N \). With abuse of notation, we define the function \( \omega : \mathbb{Z} \rightarrow \mathbb{N} \) such that:

\[
\omega(c) := \begin{cases} 
2 & \text{if } c \in \mathbb{Z}/4\mathbb{Z} \\
N & \text{if } c \in \mathbb{Z}/2^N\mathbb{Z} \land c \notin \mathbb{Z}/2^{N-1}\mathbb{Z} 
\end{cases}
\]

Note that \( \omega(c) \geq 2 \) for all \( c \in \mathbb{Z} \).

C. Safety Verification

A transition system \( T \) is a tuple \( (\mathcal{U}, \text{Init}, \text{Tr}, \text{Bad}) \), where \( \mathcal{U} \) is a set of Boolean variables, \( \text{Init} \) and \( \text{Bad} \) are formulas over \( \mathcal{U} \) denoting the set of initial states and bad states, respectively, and \( \text{Tr} \) is a formula over \( \mathcal{U}, \mathcal{U}' \) denoting the transition relation. A state \( s \in \mathcal{U} \) is said to be reachable in \( T \) if and only if there exists \( k \geq 0 \) and \( s_0, s_1, \ldots, s_k \) s.t. \( s_0 \in \text{Init} \) and \( (s_i, s_{i+1}) \in \text{Tr} \) for \( 0 \leq i < k \), and \( s = s_k \).

A transition system \( T \) is UNSAFE iff there exists a state \( s \in \text{Bad} \) s.t. \( s \) is reachable. The path from \( s_0 \in \text{Init} \) to \( s \in \text{Bad} \) is called a counterexample (CEX).

A transition system \( T \) is SAFE iff all reachable states in \( T \) do not satisfy \( \text{Bad} \). Equivalently, there exists a formula \( \text{Inv} \), called a safe inductive invariant, that satisfies:

\[
\text{Init}(\mathcal{U}) \Rightarrow \text{Inv}(\mathcal{U}) \tag{2}
\]

\[
\text{Inv}(\mathcal{U}) \land \text{Tr}(\mathcal{U}, \mathcal{U}') \Rightarrow \text{Inv}(\mathcal{U}') \tag{3}
\]

\[
\text{Inv}(\mathcal{U}) \Rightarrow \neg \text{Bad}(\mathcal{U}) \tag{4}
\]

A safety verification problem is to decide whether a transition system \( T \) is SAFE or UNSAFE.

Note that a transition system can be modeled by a sequential circuit with a single output. In this case, the Boolean variables \( \mathcal{U} \) represent registers and primary inputs, \( \text{Init} \) defines the initial values for the registers, and \( \text{Bad} \) defines the logic driving the output.
combinational implementation is considered more efficient, it cycles are required to complete the computation. While the number of bits it can handle (wider bit-vectors mean more takes several cycles. Moreover, there is no restriction on the bit of a given operand is available. As a result the computation k “flow” in, one by one, where at the k-th cycle, only the k-th bit of a given operand is available. As a result the computation takes several cycles. Moreover, there is no restriction on the number of bits it can handle (wider bit-vectors mean more cycles are required to complete the computation). While the combinational implementation is considered more efficient, it may not be the best representation for formal reasoning.

A. LIA to Transition System

From this point on, unless stated otherwise, \( \varphi \) is a formula in either LIA\(_N\) or LIA.

Given a formula \( \varphi \), L1ATOMC reduces \( \varphi \) to a transition system \( T \). The reduction is based on the representation of integers as bit-vectors. While an integer \( c \in \mathbb{Z}/2^N\mathbb{Z} \), for some \( N \geq 2 \), can be represented by a bit-vector of size \( k \) for \( k \geq N \), this is not the case when considering an arbitrary integer \( v \) over \( \mathbb{Z} \). As a result, a formula in LIA cannot be represented using fixed-width bit-vectors. To overcome this issue, and considering our intuitive example, we represent input variables in \( \varphi \) (either fixed-width bit-vectors or integers) as inputs to a sequential circuit, and thus, as unbounded bit-vectors. Intuitively, an unbounded bit-vector \( b \) is modeled by an unbounded stream of bits, starting from the LSB. More precisely, the bits of \( b \) are read over time, such that the \( k \)-th bit \( b_k \) is available at the \( k \)-th time cycle. Representing a constant integer \( c \in \mathbb{Z}/2^N\mathbb{Z} \) by a bit-vector of size \( k \), where \( k > N \), can be achieved by means of sign extension. Ideally, by duplicating the \( N \)-th bit for every \( k > N \).

Arithmetic constraints and relations in \( \varphi \) are modeled with sequential logic, and logical operators are treated using the corresponding logical gates.

The top level L1ATOMC procedure appears in Algorithm 1. L1ATOMC transforms a formula \( \varphi \) over variables \( \mathcal{V} \), to a transition system \( T \). \( T \) is represented by a sequential circuit \( C \). We discuss three different parts of L1ATOMC: initialization (lines 1-13), translation of constraints (lines 14-16), and the modeling of the property, i.e. Bad (line 19).

1) Initialization: The main part of initialization is to find the minimal width required to represent constants that appear in \( \varphi \). Recall that in this work, we use the 2’s complement representation. For example, if \(-3 \) (101 in binary) and 12 (01100 in binary) appear in \( \varphi \), then the minimal width is 5. More formally, \( k_{\min} = \max_{c \in \varphi} \omega(c) \) (see Equation 1). Now, assume that for a constant \( c \in \varphi \), there exists a wire \( w_c \in C \) representing it. The value of \( w_c \) at a given cycle is determined by the 2’s complement representation of \( c \). For example, for \(-3 \), \( w_c = 1 \) at cycle 0 and at cycle 2, \( w_c = 0 \) at cycle 1. To achieve this, we create a counter in \( C \) (line 3), which counts cycles up to \( k_{\min} \). We denote by \( w_{\min} \) the wire in \( C \) that becomes \( \top \) once the counter reaches \( k_{\min} - 1 \) and is \( \bot \) otherwise (i.e. from 0 to \( k_{\min} - 1 \)). For the example above, the counter counts from 0 to 4. Using this counter we can set \( w_c \) to the right value at the right cycle. After hitting the maximum value of the counter, \( w_c \) is sign-extended. Going back to our example, for every cycle \( k > 4 \), the value of \( w_c \) equals the value it was assigned to at the 4-th cycle.

2) Translating Linear Arithmetic Constraints: The function TRANSLATE operates on a Directed Acyclic Graph (DAG) mirroring the structure of \( \varphi \). Leaf nodes in \( G \) represent either a variable (in \( \mathcal{V} \)) or a constant in \( \varphi \), while internal nodes represent the different operators. Starting from the root, TRANSLATE recursively traverses \( G \), and for each node in \( G \), the proper logic is added to \( C \). TRANSLATE appears in Algorithm 2.

Before describing the transformation in more detail, we highlight the handling of the sign bit. The input variables of \( \varphi \) are represented as streams of bits over time, namely, at each cycle, a new bit is added. As a result, at every cycle, the most recent bit is treated as the sign bit. Consequently, as the computation progresses, the sign bit is updated.

We now describe the transformation in more detail. W.l.o.g. every node \( g \in G \) has at most two operands, \( a \) and \( b \). In addition, for simplicity, we assume that \( g := \text{ite}(c, a, b) \) is modeled by adding a new variable \( u \) s.t. \( g := u \) and \((c \Rightarrow u = a) \vee (\neg c \Rightarrow u = b) \) is added as a conjunct
of sort Integer, bit-vectors, or Boolean. The rules below
to the formula \( \varphi \) represented by
operands are
out, respectively.

\[ a \land \neg b \land \neg c \land \neg d \land \neg e \land \neg f \]

Our implementation handles the \textit{ite} operator directly.

1) \textbf{LIAToMC}(\varphi)

\begin{algorithm}
\begin{algorithmic}
  \State \textbf{Input:} A LIA formula \( \varphi \) over variables \( \forall \)
  \State \textbf{Output:} A safety verification problem \((\text{Init}, \text{Tr}, \text{Bad})\)
  \State \( C \leftarrow \text{InitCircuit}() \)
  \State \( k_{\text{min}} \leftarrow \text{FindMinWidth}(\varphi) \)
  \State \( C.\text{CreateCounter}(k_{\text{min}}) \)
  \State \( G \leftarrow \text{DAG}(\varphi) \)
  \State \( g_{\text{root}} \leftarrow \text{G.Root}() \)
  \State \( \text{Translate}(C, G, g_{\text{root}}) \)
  \State \( \text{Init} \leftarrow C.\text{Init}() \)
  \State \( \text{Tr} \leftarrow C.\text{Tr}() \)
  \State \( \text{Bad} \leftarrow w_{\text{min}} \land C.\text{Output}() \)
  \State \( T = (\text{Init}, \text{Tr}, \text{Bad}) \)
  \State \text{return } T \)
\end{algorithmic}
\end{algorithm}

2) \textbf{Translate}(C, G, g)

\begin{algorithm}
\begin{algorithmic}
  \State \textbf{Input:} A circuit \( C \), DAG \( G \) and a node \( g \)
  \For{\text{h} \in \text{g.Operands}()}
    \If{\text{h} is undefined in C}
      \State \text{Translate}(C, G, h)
    \EndIf
  \EndFor
  \State \( C.\text{CreateLogic}(g) \)
\end{algorithmic}
\end{algorithm}

Let us assume that for unary and binary operators the operands are \( a \) or \( a \) and \( b \), respectively, where \( a \) and \( b \) can be of sort Integer, bit-vectors, or Boolean. The rules below describe the transformation.

- \( g \) is a leaf of sort integer/bit-vector: create an input terminal \( v_g \) in \( C \), \( w_g := v_g \).
- \( g \) is a leaf of a constant type (i.e. \( c \in \mathbb{Z} \)): use the counter to add logic that defines the right values for \( w_g \) over time.
- \( g \) is a leaf of sort Boolean: create an uninitialized latch \( v_g \) such that \( v'_g := v_g \) and \( w_g := v_g \).
- Boolean operations are implemented using their equivalent logical gates.
- \( g := a + b \): add a sequential adder (see Figure 2). A latch \( v_+ \) and a full-adder \( f(v_a, v_b, s, v_+, c_{\text{out}}) \) are added. \( v_+ \) is defined as follows: \( \text{init}(v_+) := \perp \land v'_+ := f.c_{\text{out}} \). \( w_g := f.s \) (note that \( f.c_{\text{in}} := v_+ \)).
- \( g := a - b \): subtraction uses the identity: \( x - y \equiv x + y + 1 \).
- \( g := c \cdot a \): multiplication by constant uses the "Shift and Add" identity. Namely, \( c \cdot a \equiv \sum_{i=0}^{k} c_i \cdot 2^i \cdot a \).

The root node of \( G \) represents the output of the circuit \( C \).

We describe equality and inequality in more detail.

\textbf{a) Equality} \( g := a = b \): The equality operator amounts to bitwise comparison, namely, \( a_i = b_i \) for every \( i \geq 0 \). The sequential implementation of it uses a latch \( v_\text{eq} \) s.t. \( w_g := v_\text{eq} \land (v_a = v_b) \). \( \text{init}(v_\text{eq}) := \top \land v'_\text{eq} := w_g \). The latch "remembers" the comparison of earlier bits. Note that if at any point in time, the bits are unequal, the value of \( v_\text{eq} \) can never be \( \top \) from that point on.

\textbf{b) Inequality} \( g := a < b \): This case is more complex since the sign bit changes at each cycle. Therefore, the sequential circuit representing it is built of two parts. The first implements an unsigned comparison, and the second takes care of the sign bit. For the unsigned comparison a latch \( v_< \) is added s.t. \( \text{init}(v_<) := \perp \land v'_< := (\neg v_a \land v_b) \lor (v_a \land \neg v_b) \land v<_\text{c} \). The sign is handled by \( w_g := \text{MUX}(v_<, v_a \lor \neg v_b, v_a \land \neg v_b) \).

The other comparison operators \( \leq, \geq \) and \( \geq \) can naturally be adjusted based on the above reasoning. We therefore refrain from describing them in detail.

Related transformations can be found in [5], [12].

3) \textbf{Modeling the Property (Bad):} Recall that when modeling a transition system with a sequential circuit, the output represents \textit{Bad}. The above reduction creates a sequential circuit \( C \) with an output \( o \). A \( k \)-cycle execution of \( C \) represents the interpretation of \( \varphi \) over \( \mathbb{Z}/2^k\mathbb{Z} \). Therefore, if \( o \) is evaluated to \( \top \) in \( k \) cycles, then \( \varphi \) is satisfiable over \( \mathbb{Z}/2^k\mathbb{Z} \).

Note that a \( k \)-cycle computation of the circuit is not necessarily well defined for all \( k \geq 1 \). The reason for this is the fact that \( \varphi \) includes constant values. Recall that \( k_{\text{min}} \) represents the minimum bit-vector width required to represent the constants in \( \varphi \), and that \( w_{\text{min}} \) indicates when \( k_{\text{min}} \) cycles of \( C \) has been completed. We can therefore use \( w_{\text{min}} \) as a "guard" when defining \textit{Bad}. The "guard" disables the output until \( k_{\text{min}} \)-th cycle.

To complete the reduction from LIA to a transition system, we create a safety verification problem \( T = (\text{Init}, \text{Tr}, \text{Bad}) \) where \( \text{Init} = C.\text{Init}() \), \( \text{Tr} = C.\text{Tr}() \) and \( \text{Bad} := w_{\text{min}} \land C.\text{Output}() \).

B. \textbf{Naïve Decision Procedure}

Before describing \textit{LIA-MC}, let us first provide an intuition. A well known SAT-based verification technique is Bounded Model Checking (BMC) [4]. Given a transition system \( T \), BMC searches for an execution that starts from the initial states (i.e. \( \text{Init} \)) and reaches the bad states (i.e. \( \text{Bad} \)) s.t. it satisfies the transition relation. This path is called a \textit{counterexample}. To find such a counterexample of length \( N \), BMC generates the following \textit{N-unrolling} formula:

\[ \mu(T, N) := \text{Init}(U^0) \land \left( \bigwedge_{i=0}^{N-1} \text{Tr}(U^i, U^{i+1}) \right) \land \text{Bad}(U^N) \]

This formula is then passed to a SAT solver. If it is satisfiable, a counterexample of length \( N \) exists. When clear from the context, we omit \( T \) and write \( \mu(N) \).

Consider again our example in Figure 2. The combinational adder that appears in Figure 1 is a result of unrolling the sequential circuit 3 times.

Recall that the reduction of LIA to a transition system treats integers as streams of bits. Let \( \varphi \) be a LIA formula and let \( T = (\text{Init}, \text{Tr}, \text{Bad}) \) be the corresponding transition system.

\textbf{Proposition 1} For \( N \geq k_{\text{min}} \), \( \varphi|_N \) and \( \mu(T, N) \) are equa-satisfiable.
Algorithm 3: LIAMC ($\varphi, N$)

Input: A LIA formula $\varphi$ over variables $V$, a constant $N \in \mathbb{N} \cup \{\infty\}$
Output: sat, unsat or unknown.
1 $T \leftarrow \text{LiAToMC}(\varphi)$
2 $\text{MC} \leftarrow \text{InitMC}(T, N)$
3 repeat
4 \hspace{1em} (result, $k$) $\leftarrow$ MC.Solve()
5 \hspace{1em} if result = SAFE then
6 \hspace{2em} return unsat
7 \hspace{1em} else if result = UNSAFE then
8 \hspace{2em} if $k = N$ then
9 \hspace{3em} return sat
10 \hspace{2em} $\pi \leftarrow \text{MC.GetCex}()$
11 \hspace{2em} if Extendable($\varphi, \pi, N$) then
12 \hspace{3em} return sat
13 \hspace{2em} else
14 \hspace{3em} return $\text{MC.BlockCex}(\pi)$
15 until $\infty$
16 return unknown

IV. DECISION PROCEDURE FOR LIA$_N$ AND LIA

In this section we describe LIAMC, a decision procedure for LIA$_N$ and LIA. In the previous section we show how a formula $\varphi$ can be reduced to a transition system, and how BMC can be used as a decision procedure. Yet, such a decision procedure is not more efficient than using BV solvers.

In order to achieve efficiency, LIAMC relies on the ability of state-of-the-art SATMC algorithms to generalize a bounded proof of correctness into a safe inductive invariant. This generalization proves the absence of a counterexample for any $N$. Note that this gives another intuitive justification for why the reduction from LIA$_N$ and LIA treats both integers and fixed-width bit-vectors as unbounded streams of bits. In case the SATMC algorithm finds an inductive invariant, there exists $k$ such that $\varphi|_N$ is unsatisfiable for every $N \geq k$.

For the case a counterexample of length $k$ exists in $T$, we have implemented a procedure that uses the “structure” for $T$ such that it can, iteratively, and incrementally extend the counterexample to a target length $N$ (for LIA$_N$) or extend the counterexample for the integers. The key insight here is to treat a counterexample of length $k$ as a partial assignment.

LIAMC appears in Algorithm 3. It can operate in two modes, which are determined by the value of $N$. If $N = \infty$, then $\varphi$ is interpreted over $\mathbb{Z}$ (LIA mode), otherwise it is interpreted over $\mathbb{Z}/2^N\mathbb{Z}$ (LIA$_N$ mode). The initialization (lines 1-2) of LIAMC starts by transforming $\varphi$ to a transition system $T = (\text{Init}, \text{Tr}, \text{Bad})$ and setting up an instance of a model checker $MC$. Note that $MC$ receives $N$, the maximum time frame it needs to consider. The main loop (lines 3-15) uses a model checker to prove either $T$ is SAFE or UNSAFE.

We assume that $\text{MC.Solve}()$ (line 4) returns a pair (result, $k$), where result is either SAFE or UNSAFE. In case result = UNSAFE then $k$ is the length of the counterexample. Otherwise, if result = SAFE, $k$ is the depth at which an inductive invariant is found, or if no invariant is found $k = N$ (indicating no counterexample up to $N$).

We now describe LIAMC in more detail. We start by describing the case $\varphi|_N$ is satisfiable, and then the case it is unsatisfiable.

A. Satisfiability: $T$ is UNSAFE (line 7)

Let us assume a counterexample of length $k$ exists. In this case, $\varphi|_k$ is satisfiable. Since the satisfiability of $\varphi|_k$ does not entail the satisfiability of $\varphi|_N$, LIAMC checks if the returned counterexample can be “extended” into a counterexample of $\varphi|_N$. In what follows we detail this procedure for both LIA$_N$ and LIA.

1) Extending a Counterexample: As noted above, a counterexample of length $k$ implies that $\varphi|_k$ is satisfiable. Due to the different semantics of LIA$_N$ and LIA$_k$ ($N > k$), the satisfying assignment for $\varphi|_k$ is not necessarily a satisfying assignment for $\varphi|_N$. The naive solution to the above problem, is to check whether the given assignment is also an assignment for $\varphi|_N$ (either when $N = \infty$ or $N < \infty$). This solution amounts to a sign-extension of the satisfying assignment. While it is simple, in most cases it does not work. As an example, consider again the following formula:

$$\varphi := (z = x + y) \land (x > 0) \land (y > 0) \land (z < 0)$$

As noted before, $x = 1, y = 1$ and $z = -2$ is a satisfying assignment for $\varphi|_2$, but it is not a satisfying assignment for $\varphi|_N$ for all $N > 2$.

Given a counterexample $\pi$ of length $k$, let us assume we would like to extend it for $\varphi|_{k+1}$. Intuitively, $\pi$ assigns values for $k$ bits out of $k + 1$. Therefore, if there exists a satisfying assignment $\pi^* \land \mu(k + 1)$ then $\pi^*$ is an extension of $\pi$ for $\varphi|_{k+1}$, since it satisfies $\mu(k + 1)$.

Using the above intuition, we can iteratively, and incrementally, extend a counterexample up to a desired depth $N$ such that $N > k$. This gives us an efficient procedure to determine satisfiability for LIA$_N$ ($N < \infty$).

We now need to handle the case of LIA ($N = \infty$). For LIA, we use the same intuition as above. Namely, a counterexample $\pi$ of length $k$ gives a valuation to the lower $k$ bits. However, we need to adjust this intuition for integers in $\mathbb{Z}$. Let us assume an integer $v \in \text{Init}(\varphi)$ is evaluated to $c_v \in \mathbb{Z}/2^k\mathbb{Z}$ in $\pi$. In order to extend it, we can add the following constraint: ($v = v^* \cdot 2^k + |c_v|) \lor (v = -(v^* \cdot 2^k + |c_v|))$ where $v^*$ is a fresh integer variable.

Given a counterexample $\pi$ of length $k$, let us define:

$$\Delta(\pi) := \bigwedge_{v \in V_I} ((v = v^* \cdot 2^k + |c_v|) \lor (v = -(v^* \cdot 2^k + |c_v|)))$$

The function $\Delta(\pi)$ captures the value $\pi$ assigns to the lower $k$ bits of an integer in $\varphi$.

Lemma 1: If $\varphi \land \Delta(\pi)$ is satisfiable, then $\varphi$ is satisfiable.
Algorithm 4: Extendable(ϕ, T, π, N)

| Input: A LIA formula ϕ and its corresponding safety verification problem T = (Init, Tr, Bad), a counterexample π of length k, and a constant N > k, s.t. N ∈ N ∪ {∞} |
| Output: (false, L) or (true, π#).

1. if N = ∞ then
2. | (result, π#) ← LIA.IsSAT(ϕ ∧ Δ(π))
3. | if result = sat then
4. | return (true, π#)
5. return (false, L)
6. else
7. | m ← k
8. | i ← k + 1
9. while i ≤ N do
10. | (result, π#) ← IsSAT(μ(T, i) ∧ π)
11. | if result = sat then
12. | m ← i
13. | i ← i + 1
14. if m = N then return (true, π#)
15. return (false, L)

In order to determine satisfiability of ϕ ∧ Δ(π), we use a LIA solver. In case ϕ ∧ Δ(π) is satisfiable, π can be extended to a satisfying assignment for ϕ. We would like to emphasize that using a LIA solver when N = ∞ (LIA mode) is intended to rule out counterexamples that may appear due to overflow. Note that it may be possible to model T s.t. overflow is not possible.

The procedure for extending counterexamples appears in Algorithm 4. In the LIA mode, we try and iteratively extend a counterexample of length k to N (lines 9-13). As mentioned above, extending a counterexample in LIA mode requires a call to a LIA solver (line 2). In both cases, a counterexample that cannot be extended is blocked.

Theorem 1 If LIAMC returns sat, then ϕ|N is satisfiable.

Proof Sketch: Let us assume a counterexample π of length k < N exists. If N < ∞, and π can be extended, then a counterexample of length N exists and thus ϕ|N is satisfiable. For N = ∞, satisfiability of ϕ follows from Lemma 1.

B. Unsatisfiability: T is SAFE (line 3)

Due to the definition of Bad, the property being verified (i.e., ¬Bad) is of the form wmin ⇒ 0. Moreover, since the reduction from ϕ to T uses unbounded bit-vectors, if a safe inductive invariant is found by the model checker, ϕ|N is unsatisfiable for all N ≥ k.

Lemma 2 Let ϕ be a formula in LIA. If there exists k s.t. for all N > k ϕ|N is unsatisfiable then ϕ is unsatisfiable.

The proof for Lemma 2 relies on the following: if ϕ is satisfiable, there exists N ∈ N s.t. ϕ|N is satisfiable.

The above lemma gives us a way to determine the unsatisfiability of ϕ: if T is SAFE and (N = ∞), LIAMC only returns SAFE if an inductive invariant is found. In that case, using Lemma 2, LIAMC concludes ϕ is unsatisfiable.

In the case N < ∞ (LIA mode), if T is SAFE up to bound N, LIAMC can terminate concluding ϕ|N is unsatisfiable even when an inductive invariant is not found. This is due to the fact that if no counterexample exists at depth N, ϕ|N is unsatisfiable.

We do like to emphasize that while there is no requirement to find a safe inductive invariant in case N < ∞, if such an invariant is found at bound smaller than N, it implies the unsatisfiability of ϕ|N.

Theorem 2 If LIAMC returns unsat, then ϕ|N is unsatisfiable.

Proof Sketch: We consider two cases. First, if no counterexample is found during the execution of LIAMC, and the model checker returns SAFE, then for N < ∞ the proof is immediate, and for N = ∞ we use Lemma 2.

The second case occurs when LIAMC blocks a counterexample π. It remains to be shown that π cannot be part of a real counterexample. Let us assume π is of length k (and k < N).

For the case of N < ∞, this is immediate - if π cannot be extended up to N it cannot be part of a real counterexample and therefore can safely be blocked.

For N = ∞, a similar logic applies. Let us assume that ϕ ∧ Δ(π) is unsatisfiable. Since every positive integer can be expressed by a sum of powers of 2, Δ(π) fixes only the first k elements of that sum. The rest of the elements in this summation are unrestricted, and therefore, if there exists an assignment for this valuation, the LIA solver finds it. However, if such an assignment does not exist, we can safely block this valuation for the first k elements of the sum.

V. Experiments

We implemented a prototype of LIAMC using a generic SMT-LIB2 parser and ABC [13]. We use the SMT-LIB2 parser to transform a LIA or LIA mode formula to a transition system represented by an And-Inverter Graph (AIG) in ABC. For the SATMC procedure we use ABC’s dprove command.

For evaluation, we use the QF_LIA benchmarks from SMT-COMP’16. Since LIAMC targets both LIA and LIA mode, we used two sets of experiments. First, we reinterpreted the LIA benchmark over ℤ/2NZ for N ∈ [32, 64, 128] and evaluated LIAMC against Boolector [9] and the Bit-Vector solver in Z3 [10]. Second, we compared the performance of LIAMC against the LIA solver in CVC4 [11] and Z3.

We set a 900 seconds time limit for all benchmarks. All experiments were conducted on a machine running Ubuntu.

2Available at http://www.cs.technion.ac.il/~yyiziel/liamc.html
3Source code is available from https://cs-static.fbk.eu/people/griggio/misc/smtlib2parser.html
4Only nec-smt subset was excluded due to its size, as even without this subset our benchmark includes more than 2700 test cases.
5Benchmarks are available from http://smtcomp.sourceforge.net/2016/benchmarks.shtml
TABLE I: Number of solved instances for LIA$_N$. Total stands for the total number of test cases in that benchmark. The difference is due to the fact that not all LIA test cases can be represented in LIA$_N$ for certain values of $N$.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>Status</th>
<th>LIAMC 32bit</th>
<th>Boolector</th>
<th>Z3 32bit</th>
<th>Virtual Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIA$_{12}$ (32bit)</td>
<td>2647</td>
<td>SAT</td>
<td>1475</td>
<td>1257</td>
<td>1371</td>
<td>1539</td>
</tr>
<tr>
<td>LIA$_{24}$ (64bit)</td>
<td>2784</td>
<td>SAT</td>
<td>1630</td>
<td>1107</td>
<td>1340</td>
<td>1100</td>
</tr>
<tr>
<td>LIA$_{128}$ (128bit)</td>
<td>2742</td>
<td>SAT</td>
<td>1585</td>
<td>1257</td>
<td>1340</td>
<td>1100</td>
</tr>
</tbody>
</table>

For different experiments of LIA and 32GB of RAM. 16.04.2 LTS, with Intel Xeon E3-1240V2 running at 3.4GHz and 32GB of RAM.

Table I shows the number of solved instances for the different experiments of LIA$_N$. As can be seen from the table, LIAMC has a big advantage specifically on satisfiable instances, for all values of $N$. LIAMC constructs a satisfying assignment, incrementally, starting from the LSB. We believe this is the main reason for the performance advantage of LIAMC over the other methods. Figures 3-5 further emphasize the performance advantage of LIAMC on satisfiable instances. Moreover, we can see the performance advantage of LIAMC grows as the width of bit-vectors grows.

It is important to note that the approaches are complementary as many test cases are solved by LIAMC and not by Boolector, and vice-versa. Overall, LIAMC solves 205 test cases not solved neither by Boolector nor Z3 for $N = 32$. For $N = 64$ and for $N = 128$, LIAMC solves 288 and 331 test cases that are not solvable by the other solvers. When compared to Boolector, for $N = 32$, LIAMC solves 370 test cases not solved by Boolector, and Boolector solves 331 test cases not solved by LIAMC. For $N = 64$ and $N = 128$, LIAMC solves 427 and 496 test cases not solved by Boolector, while Boolector solves 482 and 501 test cases not solved by LIAMC. In the case of Z3, for $N = 32, 64, 128$, LIAMC solves 324, 329 and 397 cases not solved by Z3, while Z3 solves 265, 337, 349 cases not solved by LIAMC.

When considering unsatisfiable instances, LIAMC finds an inductive invariant in 577 cases. It is important to note that for large values of $N$ (e.g. $N \geq 512$) this fact translates to a clear advantage of LIAMC over the other solvers.

Table II presents solved instances in LIA mode. It compares LIAMC to CVC4 and Z3. The table shows that both CVC4 and Z3 perform better than LIAMC. Analyzing the results shows that LIAMC can solve 88 instances not solved by CVC4, and 126 instances not solved by Z3.

We would like to note that the current results of LIAMC (in both modes) can be greatly improved. While the dprove command in ABC is capable, we are sure that LIAMC can benefit from a portfolio-based model checker, as well as from SATMC algorithms that target the kind of transition systems LIAMC generates. To evaluate this idea, we have chosen a random subset of unsolved instances and used the SATMC algorithm AVY as part of LIAMC. Many of these unsolved instances (UNSAT) were solvable by AVY. Moreover, an efficient BMC engine can probably solve many of the UNSAT cases LIAMC did not solve. This is due to the conceptual similarity between using BMC and an eager BV solver (as mentioned before). We intend to explore these avenues in our future work.

We did not add these solved instances to the results presented in this paper since we had not run AVY on the entire benchmark set.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LIAMC</th>
<th>CVC4</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>1289</td>
<td>1657</td>
<td>1581</td>
</tr>
<tr>
<td>UNSAT</td>
<td>577</td>
<td>1106</td>
<td>1103</td>
</tr>
<tr>
<td>Uniquely Solved</td>
<td>18</td>
<td>18</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 3: $\mathbb{Z}/2^{32}\mathbb{Z}$: Trend for satisfiable instances (32 bit).

Fig. 4: $\mathbb{Z}/2^{64}\mathbb{Z}$: Trend for satisfiable instances (64 bit).

Fig. 5: $\mathbb{Z}/2^{128}\mathbb{Z}$: Trend for satisfiable instances (128 bit).
VI. RELATED WORK

A translation from linear arithmetic to finite automata had been proposed more than 20 years ago [14] and studied further in a number of works [15]–[17]. Our work belongs to that line of research as a finite automata can be thought of as a sequential circuit. The added value of our work is that we present an efficient decision procedure, achieved by leveraging a symbolic representation of the automata (i.e. the sequential circuit) and advancements from modern SATMC research, enhanced by novel generalization techniques. Our results challenge the pessimistic forecast in [16]: “there is little hope that these techniques will consistently outperform more traditional approaches when these can be applied”.

Several related methods for synthesizing unbounded bit-vector arithmetic were proposed in [12], [18], but in these works the context is synthesis and no efficient decision procedure was detailed.

A closely related line of work is [5], [19], where a reduction from a fragment of BV (restricted to bitwise operators, addition, subtraction, shift by one, indexing and comparators) to propositional model checking has been introduced (as a by-product of studying the complexity of bit-vector logic). The proposed method has been implemented and shown to outperform traditional SMT solvers on crafted BV benchmarks, restricted to the aforementioned BV fragment. Unlike the transformation applied by LIAMC, the modeling suggested in [19] encodes the width of the bit-vectors into the model checking problem, making SATMC algorithms inefficient. As a result, BDD-based model checking algorithms were found to be the most efficient experimentally [19]. LIAMC shows how SATMC can be applied efficiently even for the subset supported by [19] by applying generalization techniques. Generalization is possible for LIAMC since the width of bit-vectors is not encoded in the transition system. In addition, our approach also handles multiplication by a constant, which makes it applicable to arbitrary formulas in LIA and LIA.

LIAMC constructs a satisfying assignment incrementally by iteratively extending a satisfying assignment from a simple theory to a more complex one (i.e. from \( \mathbb{Z}/2^k\mathbb{Z} \) to \( \mathbb{Z}/2^N\mathbb{Z} \) where \( N > k \)). A somewhat similar concept is applied by [20] in the context of floating-point arithmetic (FPA). In [20], the formula is solved w.r.t. a simpler “proxy” theory. In case that a satisfying assignment is found, it is then tried to be adjusted to FPA semantics.

VII. CONCLUSION

In this paper we introduced LIAMC, a novel decision procedure for \( \text{LIA}_N \) and LIA. LIAMC is based on a transformation of linear arithmetic constraints to a transition system. While this approach, in general, has been suggested and explored in the past in different contexts, to our knowledge LIAMC is the first efficient implementation. There are three key insights that make LIAMC efficient and different from previous approaches: 1) We treat both integers and fixed-width bit-vectors as unbounded streams of bits, which allows us to apply SATMC, and 2) We use generalization to efficiently reason about wide bit-vectors and integers.

Our experiments show that LIAMC can solve many instances that cannot be solved by other top-tier SMT solvers, for both \( \text{LIA}_N \) and LIA. Moreover, in the case of \( \text{LIA}_N \), LIAMC is the best performer solving the most instances. We therefore believe that this approach has a promising future.

REFERENCES