Solving LIA* Using Approximations

Maxwell Levatich and Nikolaj Bjørner and Ruzica Piskac and Sharon Shoham

Yale maxwell.levatich@yale.edu,ruzica.piskac@yale.edu
Microsoft Research nbjorner@microsoft.com
Tel Aviv University sharon.shoham@gmail.com

Abstract. Linear arithmetic with stars, LIA*, is an extension of Presburger arithmetic that allows forming indefinite summations over values that satisfy a formula. It has found uses in decision procedures for multi-sets and for vector addition systems. LIA* formulas can be translated back into Presburger arithmetic, but with non-trivial space overhead. In this paper we develop a decision procedure for LIA* that checks satisfiability of LIA* formulas. By refining on-demand under and over-approximations of LIA* formulas, it can avoid the space overhead that is integral to previous approaches. We have implemented our procedure in a prototype and report on encouraging results that suggest that LIA* formulas can be checked for satisfiability without computing a prohibitively large equivalent Presburger formula.

1 Introduction

Decision procedures for Presburger arithmetic, also known as linear integer arithmetic, LIA, are fundamental to many uses of SMT solvers. LIA is a first-order theory of integers that includes addition and subtraction, but does not include multiplication between variables. Reasoning about linear integer arithmetic is widely used in verification. Furthermore, there are several decidable theories for which the satisfiability problem reduces to reasoning in LIA [13,16]. Yet, LIA is a mild subset of the highly undecidable Peano arithmetic.

In this paper, we pursue an extension of LIA called LIA*. LIA* extends LIA by admitting predicates of the form $\boldsymbol{x} \in \{\boldsymbol{y} \mid F\}^\star$, where F is a LIA (or in the nested case, a LIA*) formula. The set of $\boldsymbol{x}$ that satisfy the formula are sums of values that satisfy F, thus $\boldsymbol{x} = \sum_{i=0}^{n} \boldsymbol{v}_i$, for some $n \geq 0$ and such that $F(\boldsymbol{v}_i)$ for each $\boldsymbol{v}_i$. We describe an efficient algorithm, also empirically tested in practice, for reasoning about LIA*. To our knowledge it is the first available approach for solving LIA* without requiring eagerly computing a semilinear set representation explicitly or using a large template as suggested in [17]. Our algorithm maintains under- and over-approximations of a star formula in the form of LIA formulas. The approximations are refined iteratively until they converge to the actual solution: the under-approximation may determine satisfiability, while the over-approximation may determine unsatisfiability. Technically, the under-approximation is weakened by extending an underapproximate semilinear representation of the formula, while the over-approximation is strengthened via

LIA* formulas: $\varphi ::= F_1 \wedge \boldsymbol{x_1} \in \{\boldsymbol{x_2} \mid F_2\}^\star$
such that $dim(\boldsymbol{x_1}) = dim(\boldsymbol{x_2})$ and $\mathit{free\text{-}vars}(F_2) \subseteq \boldsymbol{x_2}$
LIA formulas:
$F ::= A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F_1 \mid \exists x.\ F \mid \forall x.\ F$
$A ::= T_1 \leq T_2 \mid T_1 = T_2$
$T ::= x \mid C \mid T_1 + T_2 \mid C \cdot T_1 \mid \mathsf{ite}(F, T_1, T_2)$
terminals: x - integer variable; C - integer constant

Fig. 1: Presburger Arithmetic and an extension with the Star Operator.

interpolation exploiting a characterization of the star operator as a solution to a set of Constraint Horn Clauses (CHCs). In the limit, the algorithm creates a semilinear set representation of a LIA formula, but only if it is unable to determine satisfiability using an approximation. The algorithm we present considers the class of formulas studied in [17]. They involve only a single star formula in a conjunction. Handling these formulas suffices for an evaluation based on multiset formulas, as well as formulas from the more specialized theory of Boolean Algebra over Presburger Arithmetic, BAPA [12].

We have also investigated how to handle *full* LIA* allowing an arbitrary nesting of star operators with negations, other Boolean connectives and quantifiers. Full LIA* extends the $\exists$LIA* fragment from [9], which does not admit alternating negations and universal quantifiers with stars. The generalization, which we do not describe in this paper, can be accomplished using a scheme that also works with under- and over-approximations of each subformula. We plan to describe this generalization in future work. While the lower bound complexity of $\exists$LIA* is known [9], we do not know the lower bound complexity of full LIA*.

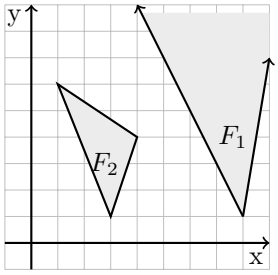
2 Linear Integer Arithmetic with the Star Operator

In this section we introduce LIA* formally. The definition of the LIA* logic relies on the crucial new operator, the star operator, defined over a set of integer vectors S, as follows:

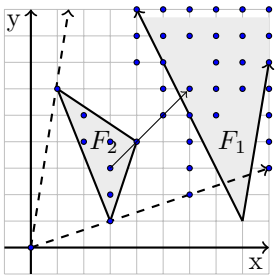
$$S^\star \quad \triangleq \quad \left\{ \sum_{i=1}^{n} \boldsymbol{s}_i \quad \mid \ \forall i. 1 \leq i \leq n.\ \boldsymbol{s}_i \in S \right\} \tag{1}$$

In other words, the set $S^\star$ is a set of all linear combinations of vectors from S. Implicitly, $\mathbf{0} \in S^\star$, for every set S. Figure 1 contains the definition of the LIA* logic. A LIA* formula is a conjunction of a LIA formula F_1 and a star formula $\boldsymbol{x_1} \in \{\boldsymbol{x_2} \mid F_2\}^\star$ that states that the vector $\boldsymbol{x_1}$ is a linear combination of solution vectors $\boldsymbol{x_2}$ of the LIA formula F_2. General LIA* formulas allow arbitrary Boolean combinations as well as nesting of the star operator.

Through the rest of the paper we often use $\varphi^\star(\boldsymbol{x_1})$, or simply $\varphi^\star$, as a shorthand for $\boldsymbol{x_1} \in \{\boldsymbol{x_2} \mid \varphi\}^\star$.



(a) Integer solutions of formulas F_1 and F_2 lie within the shaded areas. Note that the solution set for F_1 is unbounded.



(b) The vector $(6, 6)$ is a solution for $F_1(x, y) \wedge F_2^*(x, y)$

Fig. 2: An illustration of a LIA* formula $F_1(x, y) \wedge F_2^*(x, y)$, such that $F_1(x, y) \Leftrightarrow y + 2x \geq 17 \wedge 6x - y \leq 47$ and $F_2(x, y) \Leftrightarrow 5x + 2y \geq 17 \wedge 3x - y \leq 8 \wedge 2x + 3y \leq 20$.

Example 1. Consider a simple LIA* example given in Fig. 2. The solid lines indicate borders within which lie integer solutions of each formula. As it is clear from Fig. 2a, formula $F_1(x, y) \wedge F_2(x, y)$ is unsatisfiable. However, the LIA* formula $F_1(x, y) \wedge F_2^*(x, y)$ is satisfiable. The dashed lines in Fig. 2b outline the borders within which lie integer vectors satisfying $F_2^*(x, y)$ – they are indicated by the points. Consider, for example, the vector $(6, 6)$: it satisfies $F_1(6, 6)$, while at the same time $(6, 6) = 2 * (3, 3)$ and $F_2(3, 3)$ holds.

Checking satisfiability of a LIA* formula is decidable [16]. Furthermore, when restricting the underlying LIA formulas to be quantifier free, it is an NP complete problem [17]. The key insight is that (i) the set of solutions of every LIA formula is a semilinear set, as proved in [8], and (ii) the representation of the solutions as a semilinear set allows to eliminate the star operator (cf. Theorem 2).

Definition 1. *A linear set $LS(\boldsymbol{a}, B)$ is defined by an integer vector $\boldsymbol{a}$ and a finite set of integer vectors $B = \{\boldsymbol{b_1}, \ldots, \boldsymbol{b_n}\}$, all of the same dimension, as follows:*

$$LS(\boldsymbol{a}, B) \quad \triangleq \quad \left\{ \boldsymbol{a} + \sum_{i=1}^{n} \lambda_i \boldsymbol{b_i} \quad | \quad \bigwedge_{i=1}^{n} \lambda_i \geq 0 \right\} \tag{2}$$

The vector $\boldsymbol{a}$ is called the shift *vector, and the vectors $\boldsymbol{b_1}, \ldots, \boldsymbol{b_n} \in B$ are called the* offset *vectors.*

A semilinear set $SLS(ls_1, \ldots, ls_n)$ is a finite union of linear sets $ls_1, \ldots, ls_n$, i.e., $SLS(ls_1, \ldots, ls_n) = \bigcup_{i=1}^{n} ls_i$.

A linear set $LS(\boldsymbol{a}, B)$ can be seen as Minkowski sum $\{\boldsymbol{a}\} + B^*$. In the sequel, we often view B as a matrix and use $\boldsymbol{\lambda} B$ as a shorthand for $\sum_{i=1}^{n} \lambda_i \boldsymbol{b_i}$.

Theorem 1 (Theorem 1.3 in [8]). *Let f be a* LIA *formula. Then the set of vectors that satisfy f forms a semilinear set. Furthermore, any semilinear set*

$U = SLS(LS(\boldsymbol{a_1}, B_1), \ldots, LS(\boldsymbol{a_k}, B_k))$ *can be characterized by a* LIA *formula, defined as follows:*

$$\textsc{Lia}(U)(\boldsymbol{x}) \quad \triangleq \quad \bigvee_{i=1}^{k} \exists \boldsymbol{\lambda} \geq 0 \;.\; \boldsymbol{x} = \boldsymbol{a}_i + \boldsymbol{\lambda} B_i \tag{3}$$

Theorem 2 (Lemmas 2 and 3 in [17]). *Let f be a* LIA *formula and let $U = SLS(LS(\boldsymbol{a_1}, B_1), \ldots, LS(\boldsymbol{a_k}, B_k))$ be the semilinear set of vectors that satisfy f. Then $f^\star(\boldsymbol{x}) \equiv \textsc{StarLia}(U)(\boldsymbol{x})$, where $\textsc{StarLia}(U)$ is a* LIA *formula that characterizes $U^\star$ and is defined as follows:*

$$\begin{aligned}\textsc{StarLia}(U)(\boldsymbol{x}) \quad \triangleq \quad & \exists \mu_1 \geq 0, \ldots, \mu_k \geq 0, \boldsymbol{\lambda}_1 \geq 0, \ldots \boldsymbol{\lambda}_k \geq 0 \;. \\ & \boldsymbol{x} = \sum_{i=1}^{k} \mu_i \boldsymbol{a}_i + \boldsymbol{\lambda}_i B_i \wedge \bigwedge_{i=1}^{k} (\mu_i = 0 \rightarrow \boldsymbol{\lambda}_i = 0)\end{aligned} \tag{4}$$

Given a LIA* formula $F_1 \wedge \boldsymbol{x} \in \{\boldsymbol{y} \mid F_2\}^\star$, where F_1 and F_2 are LIA formulas, Theorem 1 ensures that there is a semilinear set describing the set of solutions of F_2. Theorem 2 shows how to use that semilinear set to eliminate the star operator. The resulting LIA formula is equivalent to $\boldsymbol{x} \in \{\boldsymbol{y} \mid F_2\}^\star$, thereby reducing satisfiability checking to LIA.

3 Reasoning about Multisets as a LIA* Problem

Multisets can be seen as a generalization of sets: they are mathematical objects where an element can appear multiple times in a collection. For example, if a set contains an element, adding that same element to the set does not change the set. However, in the same scenario, adding an element to a *multiset* results in a different multiset. Formally, a multiset can be defined as a function from some unbounded set of elements $\mathbb{E}$ to the set of natural numbers $\mathbb{N}$. Formulas involving multisets with cardinality constraints naturally arise in verification when a container data structure is abstracted in a way that it only tracks the elements appearing in the data structure. While there are several decision procedures for multisets [15–17, 20], they were essentially impractical, until now.

Multisets And Presburger Arithmetic (MAPA) formulas allow an arbitrary Boolean combination of atomic formulas that compare multisets for equality ($m_1 = m_2$) or inclusion ($m_1 \subseteq m_2$), and quantifier-free LIA formulas, where arithmetic terms are extended with a cardinality operator for multisets; The syntax is given in Figure 3. The cardinality operator returns the number of elements in the multiset; the same elements are counted as many times as they appear. To count the number of distinct elements in a multiset m, we can use the expression $|\mathsf{set}(m)|$. The $\mathsf{set}(\cdot)$ function converts a multiset into a set. As an illustration, two different multisets $\{a, a, a, b, b\}$ and $\{a, a, b, b, b\}$ as sets are the same: $\mathsf{set}(\{a, a, a, b, b\}) = \mathsf{set}(\{a, a, b, b, b\}) = \{a, b\}$. Using the $\mathsf{set}(\cdot)$ function,

top-level formulas:

$$F ::= A \mid F \wedge F \mid F \vee F \mid \neg F$$
$$A ::= M{=}M \mid M \subseteq M \mid F_{\mathrm{LIA}}$$

quantifier-free linear arithmetic formulas:

$$F_{\mathrm{LIA}} ::= A_{\mathrm{LIA}} \mid F_{\mathrm{LIA}} \wedge F_{\mathrm{LIA}} \mid F_{\mathrm{LIA}} \vee F_{\mathrm{LIA}} \mid \neg F_{\mathrm{LIA}}$$
$$A_{\mathrm{LIA}} ::= t \leq t \mid t{=}t$$

linear arithmetic terms:

$$t ::= x \mid \ \mid M \mid \ \mid C \mid t + t \mid C \cdot t \mid \mathsf{ite}(F_{\mathrm{LIA}}, t, t)$$

multiset expressions:

$$M ::= m \mid \emptyset \mid M \cap M \mid M \cup M \mid M \uplus M \mid M \setminus M \mid M \setminus\!\setminus M \mid \mathsf{set}(M)$$

terminals:

m - multiset variables; x - integer variable; C - integer constant

Fig. 3: MAPA: Quantifier-Free Multiset Constraints with Cardinality Operator

we can easily express standard BAPA benchmarks as MAPA benchmarks. All standard set expressions are also defined on multisets. In addition the disjoint union, $\uplus$, operator produces a multiset where the multiplicity of elements are added. Figure 3 provides a grammar for quantifier-free MAPA.

The semantics of MAPA is provided in Figure 4, which describes how every MAPA formula can be reduced to an equisatisfiable LIA* formula in linear time. The reduction follows a sequence of rewriting steps corresponding to the definitions of multiset operators. A justification for this translation is provided in [16].

Example 2. Consider the following constraint: if an element is removed from a multiset, its size will decrease by one. In MAPA, this property can be expressed as $s \subseteq L \wedge |s| = 1 \Rightarrow |L \setminus s| = |L| - 1$. To prove its validity, we apply the algorithm given in Fig. 4 to check the satisfiability of the formula $s \subseteq L \wedge |s| = 1 \wedge |L \setminus s| \neq |L| - 1$. The first step flattens the formula and we introduce new variables for all non-trivial expressions:

$$x_1 \neq x_2 - 1 \wedge x_3 = 1 \wedge |m| = x_1 \wedge |L| = x_2 \wedge |s| = x_3 \wedge m = L \setminus s \wedge s \subseteq L$$

The resulting formula has three parts: a part that is a pure LIA, a part which defines cardinality constraints, and a part that is only about multisets without cardinality constraints. Every MAPA formula can be reduced to this form.

The next step is to translate the resulting formula into a LIA* formula. For every multiset variable M we introduce an integer variable $\widetilde{M}$. After some basic simplifications the above formula becomes:

$$x_1 \neq x_2 - 1 \wedge x_3 = 1 \wedge (x_1, x_2, x_3) \in \{(\widetilde{m}, \widetilde{L}, \widetilde{s}) \mid \widetilde{m} = \widetilde{L} - \widetilde{s} \wedge \widetilde{s} \leq \widetilde{m}\}^\star$$

For brevity, we suppress the sign constraints $\widetilde{m} \geq 0, \widetilde{L} \geq 0$ and $\widetilde{s} \geq 0$.

INPUT: a multiset formula in the syntax of Figure 3
OUTPUT: an equisatisfiable LIA* formula

1. Occurrences of multiset equalities $M_1 = M_2$ that are not top-level are rewritten to $|M_1| = |M_2| \wedge |M_1 \setminus M_2| = 0 \wedge |M_2 \setminus M_1| = 0$, and similar with $M_1 \subseteq M_2$.
2. Flatten all expressions e where e is one of the expressions $\emptyset$, $M_1 \cup M_2$, $M_1 \cup M_2$, $M_1 \uplus M_2$, $M_1 \setminus M_2$, $M_1 \setminus\!\setminus M_2$, $\mathsf{set}(M_1)$, $|M_1|$, and where the occurrence of e is not already in a top-level conjunct $x = e$ or $e = x$ for some variable x:
   $C[e] \rightsquigarrow (x_f = e \wedge C[x_f])$, where x_f is a fresh variable.
3. Furthermore, for multi-set variable M_i introduce a top-level conjunction $x_i = |M_i|$ if it doesn't already exist for fresh x_i.
4. Create a LIA* formula. The step eliminates all multisets M_i using a corresponding fresh integer variable $\widetilde{M_i}$. Let $x_1 = |M_1|, \ldots, x_n = |M_n|$ be the cardinality equalities, then the integer variables are $\widetilde{M_1}, \ldots, \widetilde{M_n}$. All the rewrite steps are applying the following schema:

$$F \wedge F_{mul} \rightsquigarrow F \wedge (x_1, \ldots, x_n) \in \{(\widetilde{M_1}, \ldots, \widetilde{M_n}) \mid F_{\mathrm{LIA}} \wedge \bigwedge_i \widetilde{M_i} \geq 0\}^*$$

The schema is applied to the following pairs of multiset and LIA formula:

$$\begin{array}{ll}
F_{mul} : M_0 = \emptyset & \rightsquigarrow F_{\mathrm{LIA}} : \widetilde{M_0} = 0 \\
F_{mul} : M_0 = M_1 \cap M_2 & \rightsquigarrow F_{\mathrm{LIA}} : \widetilde{M_0} = \mathsf{ite}(\widetilde{M_1} \leq \widetilde{M_2}, \widetilde{M_1}, \widetilde{M_2}) \\
F_{mul} : M_0 = M_1 \cup M_2 & \rightsquigarrow F_{\mathrm{LIA}} : \widetilde{M_0} = \mathsf{ite}(\widetilde{M_1} \leq \widetilde{M_2}, \widetilde{M_2}, \widetilde{M_1}) \\
F_{mul} : M_0 = M_1 \uplus M_2 & \rightsquigarrow F_{\mathrm{LIA}} : \widetilde{M_0} = \widetilde{M_1} + \widetilde{M_2} \\
F_{mul} : M_0 = M_1 \setminus M_2 & \rightsquigarrow F_{\mathrm{LIA}} : \widetilde{M_0} = \mathsf{ite}(\widetilde{M_1} \leq \widetilde{M_2}, 0, \widetilde{M_1} - \widetilde{M_2}) \\
F_{mul} : M_0 = M_1 \setminus\!\setminus M_2 & \rightsquigarrow F_{\mathrm{LIA}} : \widetilde{M_0} = \mathsf{ite}(\widetilde{M_2} = 0, \widetilde{M_1}, 0) \\
F_{mul} : M_0 = \mathsf{set}(M_1) & \rightsquigarrow F_{\mathrm{LIA}} : \widetilde{M_0} = \mathsf{ite}(1 \leq \widetilde{M_1}, 1, 0) \\
F_{mul} : M_1 \subseteq M_2 & \rightsquigarrow F_{\mathrm{LIA}} : \widetilde{M_1} \leq \widetilde{M_2} \\
F_{mul} : M_1 = M_2 & \rightsquigarrow F_{\mathrm{LIA}} : \widetilde{M_1} = \widetilde{M_2} \\
F_{mul} : x_i = |M_i| & \rightsquigarrow \mathsf{true}
\end{array}$$

Fig. 4: Algorithm for converting MAPA formulas to LIA* formulas.

The final step is the elimination of the star operator. A semilinear set describing all the solutions of the formula $\widetilde{m} = \widetilde{L} - \widetilde{s} \wedge \widetilde{s} \leq \widetilde{m}$ is a linear set $LS((0,0,0), \{(1,1,0), (0,1,1)\})$. Having the zero vector as the shift vector, simplified the process of eliminating the star operator:

$$(x_1, x_2, x_3) \in \{(\widetilde{m}, \widetilde{L}, \widetilde{s}) \mid \widetilde{m} = \widetilde{L} - \widetilde{s} \wedge \widetilde{s} \leq \widetilde{m}\}^\star \Leftrightarrow \\ \exists \lambda_1, \lambda_2 . (x_1, x_2, x_3) = \lambda_1(1,1,0) + \lambda_2(0,1,1)$$

The final formula $x_1 \neq x_2 - 1 \wedge x_3 = 1 \wedge (x_1, x_2, x_3) = \lambda_1(1,1,0) + \lambda_2(0,1,1)$ is unsatisfiable, proving that the originally given formula was valid.

4 Checking Satisfiability of LIA* Formulas by Approximating from Above and Below

In this section, we explain our algorithm for checking satisfiability of LIA* formulas.

We fix a LIA* formula $g \wedge \boldsymbol{x} \in \{\boldsymbol{y} \mid f\}^\star$. Observe that the set of solutions of $f^\star$ is the least fixpoint of the following set of equations (Constrained Horn Clauses):

$$\begin{aligned} \boldsymbol{x} = \mathbf{0} &\longrightarrow f^\star(\boldsymbol{x}) \\ f^\star(\boldsymbol{y}) \wedge f(\boldsymbol{z}) \wedge \boldsymbol{x} = \boldsymbol{y} + \boldsymbol{z} &\longrightarrow f^\star(\boldsymbol{x}) \end{aligned} \tag{5}$$

However, to determine unsatisfiability of $g \wedge f^\star$, it suffices to find an over-approximation o of $f^\star$ such that $g \wedge o$ is UNSAT, while satisfiability may be determined based on satisfiability of an under-approximation u. As such, rather than computing a LIA formula that captures $f^\star$, the algorithm approximates this set and uses the approximations for checking satisfiability of $g \wedge f^\star$. To do so, the algorithm maintains:

- A LIA formula u that underapproximates $f^\star$, i.e., $u \rightarrow f^\star$.
- A LIA formula o that overapproximates $f^\star$, i.e., $f^\star \rightarrow o$.

Algorithm 1 displays the steps for checking satisfiability of $g \wedge f^\star$ as a set of inference rules. The algorithm manipulates three types of states: *initial states* of the form $\langle g, \varphi \rangle$, *internal states* of the form $\langle g, u, \varphi, o \rangle$ and *terminal states* $[u, o]$, where $g, u, o, f \in \text{LIA}$ and $\varphi = f^\star$. The formulas u and o are under- and overapproximations, respectively, of φ, and as such every state satisfies the invariant that $u \rightarrow \varphi \rightarrow o$.

On input $g \wedge f^\star$, the algorithm starts at the initial state $\langle g, f^\star \rangle$. From the initial state it follows the $\star$-Init rule and transitions to the internal state $\langle g, \boldsymbol{x} = 0, f^\star, \mathsf{true} \rangle$ that maintains in addition to g and $f^\star$ also approximations of $f^\star$. $\star$-Init initializes the underapproximation of $f^\star$ to include only $\mathbf{0}$, and initializes the overapproximation to true.

Transitions between (internal) states refine the approximations according to the inference rules: weaken the underapproximation of $f^\star$ (rule $\star$-Weaken) or strengthen its overapproximation (rule $\star$-Strengthen). These transitions take the form

$$\langle g, u, \varphi, o \rangle \implies \langle g, u', \varphi, o' \rangle \qquad \text{such that} \qquad u \rightarrow u' \rightarrow \varphi \rightarrow o' \rightarrow o \ .$$

We explain $\star$-Strengthen in Section 4.1, and $\star$-Weaken in Section 4.2.

The $\star$-Converge rule identifies the case where the underapproximation u has become an over-approximation of $f^\star$. This happens when u satisfies Equation (5) (recall that $f^\star$ is the least solution of these equations). $\star$-Converge

Algorithm 1: Procedure for checking satisfiability of a LIA* formula $g \wedge \boldsymbol{x} \in \{\boldsymbol{y} \mid f\}^\star$.

Initial states: $\langle g, \varphi \rangle \in \text{LIA} \times \text{LIA}^\star$
Internal states: $\langle g, u, \varphi, o \rangle \in \text{LIA} \times \text{LIA} \times \text{LIA}^\star \times \text{LIA}$ s.t. $o \rightarrow \varphi \rightarrow u$
Terminal states: $[u, o] \in \text{LIA} \times \text{LIA}$

$$\langle g, f^\star \rangle \implies \langle g, \boldsymbol{x} = 0, f^\star, \mathsf{true} \rangle \qquad \star\text{-Init}$$

$$\frac{g \wedge o \text{ is UNSAT}}{\langle g, u, f^\star, o \rangle \implies [u, o]} \text{ Exit-UNSAT} \qquad \frac{g \wedge u \text{ is SAT}}{\langle g, u, f^\star, o \rangle \implies [u, o]} \text{ Exit-SAT}$$

$$\frac{f(\boldsymbol{x}) \wedge \neg u(\boldsymbol{x}) \text{ is UNSAT}}{\langle g, u, f^\star, o \rangle \implies \langle g, u, f^\star, u \rangle} \star\text{-Converge}$$

$$\frac{\begin{array}{c}\boldsymbol{x} = \boldsymbol{v} \models f(\boldsymbol{x}) \wedge \neg u(\boldsymbol{x}) \\ u' = \textsc{WeakenUnder}(u, \boldsymbol{v})\end{array}}{\langle g, u, f^\star, o \rangle \implies \langle g, u', f^\star, o \rangle} \star\text{-Weaken}$$

$$\frac{\begin{array}{c}u(\boldsymbol{x}) \wedge f^{\leq 2n}(\boldsymbol{y}) \wedge g(\boldsymbol{x} + \boldsymbol{y}) \text{ is UNSAT} \\ o' = \textsc{StrengthenOver}(o, u, f^{\leq n}, g)\end{array}}{\langle g, u, f^\star, o \rangle \implies \langle g, u, f^\star, o' \rangle} \star\text{-Strengthen}$$

recognizes this case by unsatisfiability of the test $f(\boldsymbol{x}) \wedge \neg u(\boldsymbol{x})$ since this test is equi-satisfiable to the "inductiveness" test $u(\boldsymbol{y}) \wedge f(\boldsymbol{x}) \wedge \neg u(\boldsymbol{x}+\boldsymbol{y})$ (because $u(\boldsymbol{0})$ and u is closed under addition, as we will see in Section 4.2). When this condition holds, it indicates that the under-approximation has converged to a LIA formula that is equivalent to $f^\star$, and satisfiability of $g \wedge f^\star$ reduces to satisfiability of $g \wedge u$, as they are equi-satisfiable.

In fact, u need not be equivalent to $f^\star$ to enable determining satisfiability of $g \wedge f^\star$. Equi-satisfiability of u and $f^\star$ with respect to g is a sufficient condition for that, which is in turn ensured by equi-satisfiability of u and o with respect to g (since $u \rightarrow f^\star \rightarrow o$). Accordingly, we say that:

Definition 2. *An internal state $\langle g, u, \varphi, o \rangle$ is* determined *when $g \wedge u$ and $g \wedge o$ are equi-satisfiable.*

Such a state is called determined since equi-satisfiability of the under- and over-approximations with respect to g implies that they are both equi-satisfiable to $f^\star$ with respect to g. Equivalently, the under-approximation is satisfiable or the over-approximation is unsatisfiable when conjoined with g:

Lemma 1. *An internal state $\langle g, u, \varphi, o \rangle$ is* determined *if and only if $g \wedge u$ is SAT or $g \wedge o$ is UNSAT.*

Algorithm 2: StrengthenOver(o,u,$f^{\leq n}$,g)

```
/* Procedure for computing an over-approximation o of f*   */
```
1 $f_1 := u(\boldsymbol{y}) \wedge f^{\leq n}(\boldsymbol{z}) \wedge \boldsymbol{x} = \boldsymbol{y} + \boldsymbol{z}$; $f_2 := \neg(\boldsymbol{x} = \boldsymbol{y'} + \boldsymbol{z'} \wedge f^{\leq n}(\boldsymbol{y'}) \wedge g(\boldsymbol{z'}))$
2 **if** $f_1 \rightarrow f_2$ **then**
3 $\quad Itp(\boldsymbol{x})$:= interpolant between f_1 and f_2
4 $\quad C$:= conjunction of all interpolants produced so far
5 $\quad o$:= the maximal subset of C such that $o(\boldsymbol{x}) \wedge f(\boldsymbol{y}) \rightarrow o(\boldsymbol{x} + \boldsymbol{y})$
6 **return** o

The Exit-SAT and Exit-UNSAT rules establish these cases as exit criteria that lead to terminal states.

Note that the exit rules may be applicable before the approximations converge to a formula that is equivalent to $f^\star$. However, in the worst case the algorithm terminates after $\star$-Converge is applied.

We discuss correctness and termination of the algorithm in Section 4.3, after we fill in the missing details for weakening and strengthening the approximations.

4.1 Computing Over-Approximations of $f^\star$

Our approach for obtaining an over-approximation of $f^\star$, depicted in Algorithm 2, is through reverse interpolation against g. Recall that $f^\star$ is the least solution of Equation (5). Hence, any solution to these equations is an overapproximation of $f^\star$. Recall further that the overapproximation o is used for early detection of unsatisfiability of $g \wedge f^\star$ (rule Exit-UNSAT). Hence, the "optimal" overapproximation (in case $g \wedge f^\star$ is unsatisfiable) is a solution for the following set of equations:

$$\begin{aligned} \boldsymbol{x} = \boldsymbol{0} &\longrightarrow o(\boldsymbol{x}) \\ o(\boldsymbol{y}) \wedge f(\boldsymbol{z}) \wedge \boldsymbol{x} = \boldsymbol{y} + \boldsymbol{z} &\longrightarrow o(\boldsymbol{x}) \\ o(\boldsymbol{x}) &\longrightarrow \neg g(\boldsymbol{x}) \end{aligned}$$

As a step towards finding such a solution, we use interpolation. For a given underapproximation u that covers in general an unbounded number of $f^\star$ solutions (including $\boldsymbol{x} = 0$) and where $u(\boldsymbol{y}) \wedge f(\boldsymbol{z}) \wedge \boldsymbol{x} = \boldsymbol{y} + \boldsymbol{z} \wedge g(\boldsymbol{x})$ is UNSAT, we can query an interpolation procedure for a predicate $Itp(\boldsymbol{x})$ such that

$$u(\boldsymbol{y}) \wedge f(\boldsymbol{z}) \wedge \boldsymbol{x} = \boldsymbol{y} + \boldsymbol{z} \rightarrow Itp(\boldsymbol{x}) \qquad \text{and} \qquad Itp(\boldsymbol{x}) \rightarrow \neg g(\boldsymbol{x}),$$

The interpolant $Itp(\boldsymbol{x})$ is disjoint from g. However, it is not in general an over-approximation of $f^\star(\boldsymbol{x})$; rather, it is an over-approximation of a single unfolding of f from u. We therefore do not use Itp as is, but use it as the basis for obtaining an overapproximation of $f^\star$.

Similar to how IC3 propagates clauses through frames that represent increasing unfoldings of the transition relation, and in the essence of the Houdini

approach for learning conjunctions of inductive predicates from a candidate set of predicates, our approach is to use conjunctions generated from all the interpolation queries to strengthen a global "inductive invariant", i.e., a formula $o(\boldsymbol{x})$ such that $\boldsymbol{x} = \mathbf{0} \longrightarrow o(\boldsymbol{x})$ and $o(\boldsymbol{y}) \wedge f(\boldsymbol{z}) \wedge \boldsymbol{x} = \boldsymbol{y} + \boldsymbol{z} \longrightarrow o(\boldsymbol{x})$. Such a formula may not be disjoint from g but it is guaranteed to overapproximate $f^\star$ (which is the least solution of these equations). Our task of producing a global invariant concludes when it implies $\neg g(\boldsymbol{x})$; or $u(\boldsymbol{x})$ witnesses satisfiability.

A drawback of posing the interpolation query with only one unfolding of f is that it could easily find a biased interpolant based on $g(\boldsymbol{x})$ or $u(\boldsymbol{x})$. We therefore pose more general interpolation queries that are forced to produce separating predicates that generalize beyond 0 or 1 unfoldings with f as follows. We consider n unfoldings of f:

Definition 3 ($f^{\leq n}(\boldsymbol{y})$).

$$\begin{array}{rcl} f^{\leq 0}(\boldsymbol{y}) & \triangleq & \boldsymbol{y} = 0 \\ f^{\leq n+1}(\boldsymbol{y}) & \triangleq & \boldsymbol{y} = 0 \vee (\exists \boldsymbol{y}_1, \boldsymbol{y}_2 \,.\, \boldsymbol{y} = \boldsymbol{y}_1 + \boldsymbol{y}_2 \wedge f(\boldsymbol{y}_1) \wedge f^{\leq n}(\boldsymbol{y}_2)) \end{array}$$

Given some choice of n such that $u(\boldsymbol{y}) \wedge f^{\leq 2n}(\boldsymbol{x}) \wedge g(\boldsymbol{x} + \boldsymbol{y})$ is unsatisfiable, Algorithm 2 computes an interpolant:

$$u(\boldsymbol{y}) \wedge f^{\leq n}(\boldsymbol{z}) \wedge \boldsymbol{x} = \boldsymbol{y} + \boldsymbol{z} \rightarrow \mathit{Itp}(\boldsymbol{x}) \qquad \text{and}$$
$$\mathit{Itp}(\boldsymbol{x}) \rightarrow \neg(\boldsymbol{x} = \boldsymbol{y'} + \boldsymbol{z'} \wedge f^{\leq n}(\boldsymbol{y'}) \wedge g(\boldsymbol{z'}))$$

and uses it as the basis for computing an over approximation as explained above. (If the above formula is satisfiable, the over approximation is not modified.)

4.2 Computing Under-Approximations of $f^\star$

The procedure WeakenUnder extends the current under-approximation u of $f^\star$ to include a solution $\boldsymbol{v}$ of f (and hence also of $f^\star$) that is not yet covered by u. Recall that such a solution also establishes that u is not yet inductive since the test $f(\boldsymbol{x}) \wedge \neg u(\boldsymbol{x})$ is equi-satisfiable to the test $u(\boldsymbol{y}) \wedge f(\boldsymbol{x}) \wedge \neg u(\boldsymbol{x} + \boldsymbol{y})$. The procedure returns a weaker under-approximation u' such that $u \rightarrow u' \rightarrow f^\star$ and $\boldsymbol{x} = \boldsymbol{v} \models u'$. The procedure relies on (i) computing a semilinear set U that underapproximates the solutions of f and includes $\boldsymbol{v}$, and (ii) using Theorem 2 to express its star using a LIA formula. Since the star operator is monotone, we are guaranteed that applying the star operator on the underapproximation of f results in an underapproximation of $f^\star$.

We start by describing a procedure, called LIA2SLS, for computing a semilinear representation of the LIA formula f with access to a LIA oracle only. WeakenUnder does not invoke that procedure per se, but it uses some of its ingredients, where it acts as the LIA oracle, as we explain in the sequel.

Definition 4 (LIA oracle). *By a* LIA *oracle we will understand a decision procedure for* LIA*, which, given a* LIA *formula f, returns a model for f if it is satisfiable, and returns UNSAT otherwise.*

Generating semilinear representation of a LIA *formula via underapproximations.* The LIA2SLS procedure, displayed in Algorithm 3, generates increasing under-approximations of f in the form of semilinear sets that converge to a representation of f.

One should observe that at any given point, LIA2SLS maintains a semilinear set $SLS(LS(\boldsymbol{a_1}, B_1), \ldots, LS(\boldsymbol{a_k}, B_k))$ that under-approximates (the set of solutions of) f. The semilinear set is represented as a set U of linear sets $LS(\boldsymbol{a_i}, B_i)$, where each of them is represented by its shift vector and offset vectors. In the sequel, we sometimes identify U with the semilinear set. Using this terminology, an invariant of the procedure is that $\textsc{Lia}(U) \rightarrow f$ (where $\textsc{Lia}(U)$ denotes the formula associated with the semilinear set, as defined in eq. (3)).

Initially, the under-approximation U is the empty set (rule **Init**). The procedure then augments the under-approximation until $f \rightarrow \textsc{Lia}(U)$, in which case $f \equiv \textsc{Lia}(U)$ and U is its representation as a semilinear set (rule **Exit**). As long as this is not the case, **Augment** extends U by adding a solution of f that is not yet covered, followed by a saturation procedure. Saturation applies the rules **Merge**, **Shift Down** and **Offset Down** that use coordinate-wise comparison between vectors, defined below, in order to minimize shift and offset vectors and, as we will see, ensure termination.

Definition 5 $(\boldsymbol{a} \preceq \boldsymbol{b})$**.** *For two integer vectors* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *define*

$$\boldsymbol{a} \preceq \boldsymbol{b} \quad \triangleq \quad \bigwedge_{0<i\leq \dim(\boldsymbol{a})} (0 \leq a_i \leq b_i \vee 0 \geq a_i \geq b_i) \tag{6}$$

Algorithm 3: LIA2SLS

Init $U := \emptyset$.

Augment Let $\boldsymbol{v}$ be a solution to $f(\boldsymbol{x}) \wedge \neg\textsc{Lia}(U)(\boldsymbol{x})$. Add the linear set $LS(\boldsymbol{v}, \emptyset)$ to U, and apply $\textsc{Saturate}(U)$ until convergence.

Exit If $f \wedge \neg\textsc{Lia}(U)$ is UNSAT, then return $\textsc{Lia}(U)$.

$\textsc{Saturate}(U)$:

Merge Let $LS(\boldsymbol{a_1}, B_1)$ and $LS(\boldsymbol{a_2}, B_2)$ be two linear sets in U such that $\boldsymbol{a_2} \preceq \boldsymbol{a_1}$. If $\forall \boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2, \lambda_3 \,.\, f(\boldsymbol{a_2} + \boldsymbol{\lambda}_1 B_1 + \boldsymbol{\lambda}_2 B_2 + \lambda_3(\boldsymbol{a_1} - \boldsymbol{a_2}))$ is valid (equivalently, $\neg f(\boldsymbol{a_2} + \boldsymbol{\lambda}_1 B_1 + \boldsymbol{\lambda}_2 B_2 + \lambda_3(\boldsymbol{a_1} - \boldsymbol{a_2}))$ is unsatisfiable) then replace the two linear sets by $LS(\boldsymbol{a_2}, B_1 \cup B_2 \cup \{\boldsymbol{a_1} - \boldsymbol{a_2}\})$ in U.

Shift Down Let $LS(\boldsymbol{a_1}, B_1)$ be a linear set in U. If there is a $\boldsymbol{b} \in B_1$, such that $\boldsymbol{b} \preceq \boldsymbol{a_1}$ and $\forall \boldsymbol{\lambda} \,.\, f(\boldsymbol{a_1} - \boldsymbol{b} + \boldsymbol{\lambda} B_1)$ is valid, then replace $LS(\boldsymbol{a_1}, B_1)$ by $LS(\boldsymbol{a_1} - \boldsymbol{b}, B_1)$.

Offset Down Let $LS(\boldsymbol{a_1}, B_1)$ be a linear set in U. If there are $\boldsymbol{b_1}, \boldsymbol{b_2} \in B_1$, such that $\boldsymbol{b_2} \preceq \boldsymbol{b_1}$ and $\forall \boldsymbol{\lambda} \,.\, f(\boldsymbol{a_1} + \boldsymbol{\lambda} B_1')$ is valid for $B_1' := (B_1 \setminus \{\boldsymbol{b_1}\}) \cup \{\boldsymbol{b_1} - \boldsymbol{b_2}\}$, then replace $LS(\boldsymbol{a_1}, B_1)$ by $LS(\boldsymbol{a_1}, B_1')$ in U.

It follows by inspecting the steps that the procedure always augments U to an improved under-approximation of f. In other words, it maintains the invariant $\textsc{Lia}(U) \rightarrow f$.

Lemma 2. *Let U be the set computed after any number of steps of Algorithm 3, and let $\textsc{Lia}(U)$ be the formula associated with it (per eq. (3)). Then $\textsc{Lia}(U) \rightarrow f$.*

Proof (sketch). The **Augment** rule adds to U a single solution of f. Any of the other rules checks whether the newly added linear set, when converted into a LIA formula via eq. (3), implies f.

In each step, the under-approximation is improved as it has more solutions or a smaller representation. Termination is obtained since $\preceq$ is a *well quasi order* (wqo) [11] (a reflexive and transitive relation where any infinite sequence of elements $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots$ contains an increasing pair $\boldsymbol{v}_i \preceq \boldsymbol{v}_j$ with $i < j$).

Lemma 3 (Termination). *Algorithm 3 terminates in a finite number of steps.*

Proof. Observe that $\preceq$ is a pointwise application of well quasi orders. Hence, by Dickson's lemma [7], it is also a well quasi order. This ensures that for any finite set U, any of the rules **Merge**, **Shift Down** and **Offset Down** may only be applied finitely many times (since a wqo does not have infinite descending sequences). Hence, to establish termination it remains to show that **Augment** cannot be applied infinitely many times. Assume to the contrary that **Augment** generates an infinite sequence $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots$. Each vector in the sequence belongs to one of the finitely many linear sets defining f. Hence, there is an infinite subsequence of $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots$ where all vectors are members of the same linear set $L(\boldsymbol{a}, B)$, and further are all merged together by **Merge**. Further, since $\preceq$ is a wqo, this subsequence has an infinite increasing subsequence $\boldsymbol{v}_{i_1} \preceq \boldsymbol{v}_{i_2} \preceq \ldots$.

For each vector $\boldsymbol{v}_{i_j}$, we denote by $\mathit{set}(i_j) = LS(\boldsymbol{a}_{i_j}, B_{i_j})$ the linear set in U to which $\boldsymbol{v}_{i_j}$ belongs after the (single) application of **Augment** that generated it followed by an iterative application of **Merge**, **Offset Down**, **Shift Down**, until they converge. An invariant that follows by induction is that $\boldsymbol{a}_{i_j} = \boldsymbol{a} + \boldsymbol{\lambda} B$ for some $\boldsymbol{\lambda}$ and, similarly, each vector in B_{i_j} is a linear combination of vectors in B, i.e., is equal to $\boldsymbol{\lambda} B$ for some $\boldsymbol{\lambda}$ (it is easy to verify that **Offset Down** and **Shift Down** preserve this property; for **Merge** we rely on our choice of vectors). The vector $\boldsymbol{a}_{i_j}$ may be decreased only finitely many times, hence at some point it stabilizes to some vector $\widetilde{\boldsymbol{a}}$. Similarly, each vector b in B_{i_j} may be decreased by **Offset Down** at most finitely many times. Hence, for each B_{i_j} there exists a time step after which all the vectors that originated from it are no longer decremented. Denote the set that contains the vectors originating from B_{i_j} after stabilization by $\widetilde{B}_{i_j}$.

Hence, the infinite sequence $\boldsymbol{v}_{i_1} \preceq \boldsymbol{v}_{i_2} \preceq \ldots$ gives rise to an infinite sequence of sets of vectors $\widetilde{B}_{i_1} \subseteq \widetilde{B}_{i_2}, \ldots$ as defined above (inclusion follows since the vectors in $\widetilde{B}_{i_j}$ no longer evolve). Note that by construction, $LS(\widetilde{\boldsymbol{a}}, \bigcup_j \widetilde{B}_{i_j})$ spans all vectors in $\boldsymbol{v}_{i_1}, \boldsymbol{v}_{i_2}, \ldots$. Further, $\bigcup \widetilde{B}_{i_j}$ must be finite since all the vectors in it are incomparable (as all vectors have stabilized) and $\preceq$ is a wqo. However,

Algorithm 4: WeakenUnder(u,$\boldsymbol{v}$)

$U :=$ getSLS(u)
Add $LS(\boldsymbol{v}, \emptyset)$ to U and apply Saturate(U) until convergence.
Return StarLia(U).

this implies that $LS(\widetilde{\boldsymbol{a}}, \bigcup_j \widetilde{B}_{i_j})$ is added to U after a finite number of steps, after which no vector from $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots$ may be generated by **Augment**, in contradiction to our assumption that infinitely many of them are generated. □

Note that by strengthening the queries into the LIA oracle to find minimal solutions modulo $\preceq$ we can effectively bound the number of queries that produce new vectors to be the same as the size of a minimal semilinear set representation. Our proof doesn't assume minimality of vectors and therefore relies on using properties of well quasi orderings.

Computing under-approximations of $f^\star$. WeakenUnder (Algorithm 4) relies on Algorithm 3 to generate a semilinear set U that under-approximates f. From U it produces an under-approximation StarLia(U) of $f^\star$ through eq. (4).

In order to compute U, WeakenUnder first extracts from u, the current under-approximation of f, the semilinear set U such that $u =$ StarLia(U). Since all underapproximations are computed by WeakenUnder, all of them follow eq. (4), which makes it easy to extract U from u. WeakenUnder then simulates an iteration of Algorithm 3 (a step of **Augment** followed by saturation) that extends U based on a new solution to f, except that it uses the provided uncovered solution $\boldsymbol{v}$ of f rather than obtaining one from the LIA-oracle.

Recall that the solution $\boldsymbol{v}$ provided to WeakenUnder is taken from $\boldsymbol{x} = \boldsymbol{v} \models f(\boldsymbol{x}) \wedge \neg u(\boldsymbol{x})$. Hence, $\boldsymbol{v}$ is a solution to f that is not yet covered by u. This means that $\boldsymbol{v}$ is not yet covered neither by U nor by $U^\star$.

Iteratively applying WeakenUnder results in a variant of Algorithm 3, where in each iteration, U is extended not with an arbitrary solution of $f \wedge \neg$Lia(U) (that may or may not be covered by $U^\star$), but rather with a solution of $f \wedge \neg$StarLia(U) as the algorithm is geared towards computing a representation of $f^\star$. Similarly to Algorithm 3, iterative application of WeakenUnder is guaranteed to converge to a precise representation U of f within a finite number of iterations, in which case StarLia(U), returned as the under-approximation of $f^\star$, is also precise (i.e., equivalent to $f^\star$). It may terminate earlier, as StarLia(U) may be equivalent to $f^\star$ even though Lia(U) is not yet equivalent to f.

4.3 Correctness

The following lemma is a simple corollary of the invariants maintained by the algorithm.

Lemma 4 (Partial Correctness). *If* $\langle g, f^\star \rangle \Longrightarrow^* [u, o]$ *then* $g \wedge u$, $g \wedge o$ *and* $g \wedge f^\star$ *are all equi-satisfiable.*

To argue termination of Algorithm 1, we must require a fair scheduling of the transitions: namely, each of the rules must be scheduled infinitely often.

Lemma 5 (Termination). *Any fair execution of Algorithm 1 starting from state* $\langle g, f^\star \rangle$ *terminates in a finite number of steps.*

As explained in the previous section, iterative application of weakening, which gradually refines u, mimics Algorithm 3. Hence, u must converge to a LIA formula that is equivalent to $f^\star$ within a finite number of steps, in which case when Algorithm 1 applies $\star$-CONVERGE it reaches a determined state, and terminates in one of the exit rules. Note that the termination argument relies only on the under-approximations and their convergence to $f^\star$. However, in practice, the over-approximations are also important for termination as they facilitate early termination without convergence to a LIA formula that is equivalent to $f^\star$.

5 Evaluation

To empirically test our decision procedure, we implemented Algorithm 1. In addition, we also implemented the translation algorithm given in Fig. 4. This way we can evaluate our tool on real-world MAPA problems. The implementation is written in Python, using the Python binding for Z3 as our LIA oracle. The implementation and benchmarks are publicly available at `https://github.com/mlevatich/sls-reachability`.

As it was pointed out in [18], there is a lack of native MAPA benchmarks. For our evaluation, we tested the code on 240 BAPA benchmarks derived from a set of benchmarks used for reasoning about distributed algorithms [1]. Since the BAPA problems involve reasoning about sets and not multisets, we used the $\mathsf{set}(\cdot)$ operator which explicitly states that a multiset variable M is a set, meaning that an element can appear at most once.

Before we expand further upon the results for each table presented here, we divide the benchmarks into classes based on their size, where the size of a benchmark is determined by the number of conjunctions in its LIA* representation. Due to our translation, this value also scales evenly with the number of free variables in the formula, and is a rough measure of a problem's complexity. For each class, we give the number of benchmarks in that class, and how many of them were sat or unsat, or timed out. We provide average statistics for the solved examples in that class about the size of the final computed semilinear set (measured as total number of vectors in its linear sets, including the offset vector for each set), the number of calls made to z3, and the total runtime of the algorithm. For all evaluations, we arbitrarily chose a timeout of 50 seconds.

The results of our initial evaluation are given by Table 1. We found that our tool handled the BAPA benchmarks very effectively – most benchmarks finished quickly and severely under-approximated the full semilinear set representation

of the problem. This experiment used a single unfolding when computing interpolants in Algorithm 2.

We noticed that the $\mathsf{set}(\cdot)$ operator and at-most-one appearance constraints increase each benchmark's difficulty, reflected by the larger semilinear sets, many Z3 calls, and longer average running times. To test how our decision procedure performs on its native theory, MAPA. We converted all 240 of the BAPA benchmarks into genuine MAPA problems by simply omitting the $\mathsf{set}(\cdot)$ constraints from the translation. This change means that the set variables in the original benchmarks are no longer considered sets but multisets. Our only intent in doing this was to create suitable benchmarks for evaluating our tool – we are not concerned with whether or not MAPA is suitable for modeling the same problems as the original benchmarks. By turning the BAPA benchmarks into MAPA benchmarks, we could exercise true multiset reasoning. The results of MAPA benchmarks are given by Table 2, in which we see a considerable speedup – even though multisets are more complex objects than sets, the omission of the multiplicity constraints results in a shorter and more efficient representation, showing the effectiveness of our tool on genuine MAPA problems.

Using the MAPA representation of the benchmarks, we further studied the reverse interpolation procedure for computing over-approximations. We applied the unfolding method given by Definition 3 with $n = 5$ to produce more general interpolants. Table 3 presents the performance of the benchmarks with unfolding added (also using MAPA semantics). By unfolding, we force Z3 to generate interpolants which are more likely to be inductive, resulting in a significant speedup and the ability to solve far more of the hard problems in the 13-16 size range.

To demonstrate the need for interpolation, we also ran our procedure with no interpolation at all. Without interpolation, unsatisfiability can only be shown when the entire semilinear set representation is computed, which is prohibitively expensive. The summary of the results is given in Table 4 (for MAPA semantics). In this case, the algorithm struggles to prove that complex examples are unsatisfiable, and must resort to generating larger semilinear sets.

Problem Size	# of Problems	Sat/Unsat/TO	SLS Size	Z3 Invocations	Time (s)
6	106	76/30/0	6	76	1.6
7 - 9	64	34/30/0	7	75	1.8
10 - 12	13	1/9/3	18	575	21.7
13 - 16	46	3/0/43	20	780	33.9
19 - 22	11	0/0/11	N/A	N/A	N/A

Table 1: BAPA evaluation summary for $n = 1$ unfoldings.

Finally, in Table 5 we provide the running times of our procedure, giving the average time spent by each evaluation on different parts of the procedure. In general, the algorithm performs very well for smaller problem sizes and the intrinsic complexity of the problem is visible on the problems of a bigger size.

Problem Size	# of Problems	Sat/Unsat/TO	SLS Size	Z3 Invocations	Time (s)
6	106	76/30/0	4	22	0.6
7 - 9	64	34/30/0	5	30	0.9
10 - 12	13	2/8/3	11	225	7.5
13 - 16	46	2/2/42	10	200	8.4
19 - 22	11	0/0/11	N/A	N/A	N/A

Table 2: MAPA evaluation summary for $n = 1$ unfoldings.

Problem Size	# of Problems	Sat/Unsat/TO	SLS Size	Z3 Invocations	Time (s)
6	106	76/30/0	4	17	0.6
7 - 9	64	34/30/0	3	15	0.7
10 - 12	13	0/11/2	2	11	0.8
13 - 16	46	3/15/28	4	76	7.9
19 - 22	11	0/0/11	N/A	N/A	N/A

Table 3: MAPA evaluation summary for $n = 5$ unfoldings.

Problem Size	# of Problems	Sat/Unsat/TO	SLS Size	Z3 Invocations	Time (s)
6	106	76/30/0	5	18	0.6
7 - 9	64	34/30/0	5	20	0.7
10 - 12	13	0/0/13	N/A	N/A	N/A
13 - 16	46	8/0/38	10	93	4.7
19 - 22	11	0/0/11	N/A	N/A	N/A

Table 4: MAPA evaluation summary without interpolation.

	Augmentation (s)	Interpolation (s)	Reduction (s)	Sat Checking (s)
BAPA	0.23	0.87	0.92	1.09
MAPA	0.07	0.48	0.17	0.33
UNFOLD-5	0.04	0.86	0.07	0.17
NO-INTERP	0.08	0	0.2	0.32

Table 5: Runtime performance profile of the procedure.

One observation is that the MAPA evaluation is much faster than BAPA – while our algorithm generalizes to BAPA, the $\mathsf{set}(\cdot)$ operator results in the increase of the $\mathsf{ite}(\cdot,\cdot,\cdot)$ expressions, which can potentially lead to an exponential blow up in size of the input formula. On the positive side, our efficient representation means that modeling multisets is comparatively easy despite their complexity, opening the opportunity for easy use of multisets in verification.

The MAPA evaluation, when compared to NO-INTERP (Table 4), also showcases the benefits of using the semilinear set over-approximation. NO-INTERP was unable to prove a single complex problem unsatisfiable, because the full semilinear set representation that witnesses unsatisfiability is too large to compute even with our reduction and augmentation cycle. NO-INTERP solved slightly more satisfiable examples than MAPA, since the algorithm could spend more time growing the semilinear set before timing out (because it was not interpolating).

The most effective evaluation was UNFOLD-5 (Table 3). Compared to MAPA, UNFOLD-5 solved 14 more of the problems of size 13-16, out of 46 total, and was faster on average for all classes of problems. The general interpolants that unfolding demands are far more likely to be inductive and, for many real-world MAPA problems, can prove unsatisfiability almost instantly. The trade-off is that the interpolation problems become heavier, as shown in Table 5, and because interpolants serve as over-approximations, they do not help for satisfiable problems.

It is possible that by tuning the unfolding by experimenting more thoroughly with different values for n, we could increase the speed and effectiveness of the algorithm even further. We can also apply the benefits of unfolding to satisfiable MAPA problems by introducing unfoldings when checking for satisfiability – even before the semilinear set underapproximation is able to reach a solution, a finite unfolding allows it to flexibly step outside itself and look for nearby solutions.

Overall, our initial results are quite promising, and there is still room for potential optimizations to be made to the basic algorithm.

6 Related Work

Several decidable extensions of LIA have been studied, such as LIA with divisibility constraints [10] and Büchi arithmetic [4, 5] that has a predicate that can distinguish whether a number is a power of two. The existential fragment of LIA* with unbounded nesting of stars, $\exists$LIA*, was established to be NEXP-complete in [9]. Although quantifier-free LIA formulas with bounded nesting of star operators lie in the NP-complete fragment, as established in [17], there is no implementation and the proposed algorithm relies on computing the semilinear representation of the solution, which is mainly unfeasible in practice. In general semilinear sets require a number of generators that is exponential in the size of the input LIA formula [19]. There are algorithms that are based on enumerative search for possible generators of a semilinear set [6], following the size ordering and yielding a potentially doubly exponential number of vectors that need to be considered. The other approach, suggested in [19], uses the bounds on the vectors in a standard basis (as obtained from a Hilbert basis). The fact that the number of basis vectors easily explodes precludes implementations that can efficiently find semilinear sets for a given formula.

To avoid explicit computation of semilinear sets, Piskac and Kuncak [17] devised a novel decision procedure that for a given LIA* formula $F_1 \wedge \boldsymbol{x} \in \{\boldsymbol{y} \mid F_2\}^\star$, constructs an equisatisfiable LIA formula $F_1 \wedge F_2'$ by using only solution vectors for formula F_2. The number of the solution vectors is high: it is bounded by $\mathcal{O}(n^2 \log n)$, where n is the size of the formula. Although this approach does not compute semilinear sets, the algorithm was still not applicable in practice. The decision procedure constructs formula F_2' in a monolithic way, producing immediately a very large formula that could not be solved by existing tools, not

even for the most simple cases. It should be clear that for modest values of n, the bound $n^2 \log n$ grows very quickly.

Zarba [20] studied a combination of multisets and linear integer arithmetic. The logic did not support the cardinality operator, but there was a count operator that would return how many times an element appears in a multiset. Lugiez [15] considered a logic of multisets with a limited cardinality operator that would return only the number of distinct elements. Piskac and Kuncak [16] introduced a more general logic that allows the standard definition of the cardinality operator. We use MAPA, a simplified, but equally expressive version of their logic. This name is chosen to also indicate that MAPA can be seen as a generalization of BAPA (Boolean Algebra and Presburger Arithmetic) [12, 13], a logic that is used to express properties about sets with cardinality constraints. The BAPA logic is used in verification of data structures [3] and distributed protocols [1].

7 Conclusion

In this paper we developed and evaluated a decision procedure for LIA*. The evaluation, using our prototype, suggested that samples extracted from BAPA applications benefited from the incremental nature of our solver. In addition, it suggested that interpolants based on bounded unfoldings were useful for finding over-approximations that were helpful determining unsatisfiability. The prototype could be improved in many ways, including notably a tighter integration within a native LIA* solver. The benefits of a native integration includes incrementality, access to preprocessing simplifications, and alternative heuristics such as sampling f for creating a large initial basis, and sound, but incomplete, inference rules. Nevertheless, we feel encouraged by the overall approach given the promising results from the prototype.

While our initial motivation for this work was to find an efficient decision procedure for reasoning about multisets with cardinality constraints, reasoning about LIA* formulas suggested new application areas. For instance, there are numerous classes of integer vector addition systems with states (VASS), where the set of reachable states is described with a semilinear set (for a classification of VASS see for example [2,14]). We conjecture that our solver for LIA* formulas could be used for checking VASS reachability for those classes.

References

1. Idan Berkovits, Marijana Lazic, Giuliano Losa, Oded Padon, and Sharon Shoham. Verification of threshold-based distributed algorithms by decomposition to decidable logics. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 245–266. Springer, 2019.
2. Michael Blondin, Christoph Haase, and Filip Mazowiecki. Affine extensions of integer vector addition systems with states. In *CONCUR*, volume 118 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

3. Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin C. Rinard. Using first-order theorem provers in the jahob data structure verification system. In *VMCAI*, volume 4349 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2007.
4. Véronique Bruyère, Georges Hansel, Christian Michaux, and Roger Villemaire. Logic and p-recognizable sets of integers. *Bull. Belg. Math. Soc*, 1:191–238, 1994.
5. J. Richard Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.
6. Evelyne Contejean and Hervé Devie. An efficient incremental algorithm for solving systems of linear diophantine equations. *Inf. Comput.*, 113(1):143–172, 1994.
7. Leonard Eugene Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35:413–422, 1913.
8. Seymour Ginsburg and Edwin H. Spanier. Semigroups, presburger formulas, and languages. *Pacific J. Math.*, 16(2):285–296, 1966.
9. Christoph Haase and Georg Zetzsche. Presburger arithmetic with stars, rational subsets of graph groups, and nested zero tests. In *LICS*, pages 1–14. IEEE, 2019.
10. Dejan Jovanovic and Leonardo Mendonça de Moura. Cutting to the chase - solving linear integer arithmetic. *J. Autom. Reasoning*, 51(1):79–108, 2013.
11. Joseph B. Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *J. Comb. Theory, Ser. A*, 13(3):297–305, 1972.
12. Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. An algorithm for deciding BAPA: boolean algebra with presburger arithmetic. In *CADE*, volume 3632 of *Lecture Notes in Computer Science*, pages 260–277. Springer, 2005.
13. Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. Deciding boolean algebra with presburger arithmetic. *J. Autom. Reasoning*, 36(3):213–239, 2006.
14. Jérôme Leroux. The general vector addition system reachability problem by presburger inductive invariants. *Logical Methods in Computer Science*, 6(3), 2010.
15. D. Lugiez. Multitree automata that count. *Theor. Comput. Sci.*, 333(1-2):225–263, 2005.
16. Ruzica Piskac and Viktor Kuncak. Decision procedures for multisets with cardinality constraints. In *VMCAI*, volume 4905 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2008.
17. Ruzica Piskac and Viktor Kuncak. Linear arithmetic with stars. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 268–280. Springer, 2008.
18. Ruzica Piskac and Viktor Kuncak. MUNCH - automated reasoner for sets and multisets. In *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 149–155. Springer, 2010.
19. Loic Pottier. Minimal solutions of linear diophantine systems: Bounds and algorithms. In *RTA*, volume 488 of *Lecture Notes in Computer Science*, pages 162–173. Springer, 1991.
20. Calogero G. Zarba. Combining multisets with integers. In *CADE-18*, 2002.