Linear Disjunctive Invariant Generation with Farkas' Lemma

HONGMING LIU, Shanghai Jiao Tong University, China
JINGYU KE, Shanghai Jiao Tong University, China
HONGFEI FU*, Shanghai Jiao Tong University, China
LIQIAN CHEN, National University of Defense Technology, China

Invariant generation is the classical problem of automatically generating logical assertions that over-approximates the set of reachable program states in a program. We consider the problem of generating linear invariants over affine while loops (i.e., loops with affine loop guards, conditional branches and assignment statements), and explore the automated generation of disjunctive linear invariants. Disjunctive invariants are an important class of invariants that capture disjunctive features of programs such as multiple phases, transitions between different modes, etc., and are much more precise than conjunctive invariants over programs with these features. To generate accurate invariants, existing approaches have investigated the application of Farkas' Lemma to conjunctive linear invariant generation, but none of them considers disjunctive linear invariants.

In this work, we propose a novel approach to generate linear disjunctive invariants via Farkas' Lemma. While our approach is based on the previous framework to apply Farkas' Lemma and the existing disjunctive patterns, our main novelties include (i) an invariant propagation technique (that propagates the invariant at the initial program location to other program locations) closely related to the disjunctive pattern we follow to improve the time efficiency, (ii) an extension of our approach to disjunctive loop summary, and (iii) the use of loop summary to improve the accuracy of disjunctive linear invariant generation over nested while loops. We implement our approach as a prototype tool under the Frama-C platform and show by experimental results that our approach can derive substantially more accurate disjunctive linear invariants and loop summaries than existing approaches, while remaining to be time efficient.

1 INTRODUCTION

Invariant generation is the classical problem of automatically generating invariants at program locations that can be used to aid the verification of critical program properties. An invariant at a program location is a logical assertion that over-approximates the set of program states reachable to that location, i.e., every reachable program state to the location is guaranteed to satisfy the logical assertion. Since invariants provide an over-approximation for the reachable program states, they play a fundamental role in program verification and can be used for safety [2, 58, 61], reachability [3, 6, 11, 17, 19, 27, 63] and time-complexity [14] analysis in program verification.

Automated approaches for invariant generation have been studied over decades and there have been an abundance of literature along this line of research. From the different types of target program objects, invariant generation can be divided into the automated generation of invariants over numerical values (e.g., integers or real numbers) [7, 9, 15, 18, 67, 77], arrays [53, 79], pointers [12, 54], algebraic data types [48], etc. By the different methodologies in existing approaches, invariant generation can be solved by abstract interpretation [9, 21, 24, 38], constraint solving [15, 18, 20, 39], logical inference [12, 30, 34, 35, 59, 74, 78, 84], recurrence analysis [32, 50, 51], machine learning [36, 42, 69, 86], data-driven approaches [16, 26, 54, 60, 66, 76], etc. Most results in the literature consider a strengthened version of invariants, called *inductive invariants*, that requires in extra the inductive condition that the invariant at a program location is preserved upon every

*Corresponding Author

Authors' addresses: Hongming Liu, Shanghai Jiao Tong University, Shanghai, China, hm-liu@sjtu.edu.cn; Jingyu Ke, Shanghai Jiao Tong University, Shanghai, China, windocotber@gmail.com; Hongfei Fu, Shanghai Jiao Tong University, Shanghai, China, jt002845@sjtu.edu.cn; Liqian Chen, National University of Defense Technology, Changsha, China, lqchen@nudt.edu.cn.

program execution back and forth to the location (i.e., under the assumption that the invariant holds at the location, it continues to hold whenever the execution goes back to the location).

An important criterion on the quality of the generated invariants is the accuracy against the exact set of reachable program states. Invariants that have too much accuracy loss (i.e., including too much program states that actually are not reachable) may cause the verification of a program property fail, while invariants with better accuracy can verify more program properties. Thus, to increase the accuracy of the generated invariants is a important problem in invariant generation. In this work, we consider the automated generation of disjunctive invariants, i.e., invariants that are in the form of a disjunction of conjunctive logical assertions. Compared with conjunctive invariants, disjunctive invariants are capable of capturing disjunctive features such as multiple phases and mode transitions in programs, and thus can be substantially more accurate than conjunctive ones.

In this work, we consider the automated generation of numerical invariants (i.e., invariants over the numerical values of program variables). Numerical invariants are an important subclass of invariants that is closely related to numerical program failures such as array out-of-bound and division by zero. To be more precise, we focus on linear disjunctive invariants over affine while loops. An affine while loop is a while loop in which each conditional branch and loop guard is specified by linear inequalities, and each assignment statement is in the form of an affine expression over program variables that specifies an affine update on the current program state. Moreover, we consider the method of constraint solving that typically ensures good accuracy on the generated invariants. We follow the existing constraint-solving approaches [18, 47, 57, 71] that apply Farkas' Lemma (a fundamental mathematical theorem in the theory of linear inequalities) to linear invariant generation, and extend these approaches to the automated generation of disjunctive linear-invariant and loop-summary generation. Our detailed contributions are as follows.

- First, we apply Farkas' Lemma and the disjunctive pattern of path dependency automata (PDA) [83] to disjunctive linear invariant generation, which to our best knowledge has not been investigated in the literature.
- Second, we improve the existing Farkas-based approaches [18, 57, 71] by a novel invariant propagation technique that first generates the invariants at the initial location and then obtains the invariants at other locations by a propagation from the initial location. The invariant propagation technique improves the overall time efficiency by reducing the amount of costly polyhedron operations (e.g., polyhedron projection and the generator computation of polyhedral cones).
- Third, we extend our approach to disjunctive loop summary of affine while loops. Loop summary is the classical problem of automatically deriving the input-output relationship for a while loop. In the extension, we follow a standard method (see e.g. Boutonnet and Halbwachs [9]) that incorporates fresh variables to represent the initial values of the program variables in the loop.
- Fourth, we handle nested while loops by integrating the loop summaries of the nesting loops into the PDA pattern to improve the accuracy of the generated invariants.
- Finally, we implement our approach as a prototype tool that can be embedded into the Frama-C program analysis platform [33].

Experimental results show that our approach outperforms previous approaches on the accuracy of disjunctive linear invariants and loop summaries, while remaining to be time efficient.

2 PRELIMINARIES

In this section, we review the model of linear transition systems [71] and linear invariant generation over such model via Farkas' Lemma. In our invariant generation algorithm, we use linear transition

systems as the abstract model for programs with affine conditions and updates. We first present the necessary definitions for linear transition systems and invariants, and then the application of Farkas' Lemma to linear invariant generation.

2.1 Linear Transition Systems and Invariants

To present linear transition systems, we first define several basic concepts related to linear inequalities as follows. A *linear inequality* (resp. *linear equality*) over a set $V = \{x_1, \ldots, x_n\}$ of real-valued variables is of the form $a_1x_1 + \cdots + a_nx_n + b \bowtie 0$, where a_i's and b are real coefficients, and $\bowtie \in \{\geq\}$ (resp. $\bowtie \in \{=\}$), respectively. A *linear assertion* over V is a conjunction of linear inequalities and equalities over X. Moreover, a *propositional linear predicate* (PLP) over X is a propositional formula whose atomic propositions are linear inequalities and equalities. A PLP is in disjunctive (resp. conjunctive) normal form (DNF) (resp. CNF) if it is a finite disjunction of linear assertions (resp. a finite conjunction of finite disjunctions of linear assertions), respectively. Note that we only consider non-strict no-smaller-than operator $\geq$ for linear inequalities, and the no-greater-than inequalities $\alpha \leq \beta$ could be equivalently transformed into $-\alpha \geq -\beta$. Moreover, although a linear equality $\alpha = \beta$ could be equivalently expressed by the conjunction of the two inequalities $\alpha \leq \beta$ and $\alpha \geq \beta$, we handle each linear equality directly since one can apply algorithmic optimizations to linear equalities. Below we present the definition of linear transition systems.

Definition 1 (Linear Transition Systems [71]). *A* linear transition system *(LinTS) is a tuple* $\Gamma = \langle X, X', L, \mathfrak{T}, \ell^*, \theta\rangle$ *where we have:*

- *X is a finite set of real-valued variables and $X' = \{x' \mid x \in X\}$ is the set of primed variables from X. Throughout the work, we abuse the notations so that (i) each variable $x \in X$ also represents its value in the current execution step of the system and (ii) each primed variable $x' \in X'$ represents the value of the unprimed counterpart $x \in X$ in the next execution step.*
- *L is a finite set of* locations *and $\ell^* \in L$ is the initial location.*
- *$\mathfrak{T}$ is a finite set of* transitions *such that each transition τ is a triple $\langle \ell, \ell', \rho\rangle$ that specifies the jump from the current location ℓ to the next location ℓ' with the guard condition ρ as a PLP over $X \cup X'$.*
- *θ is a PLP in DNF over the variables X. Informally, each disjunctive clause of PLP θ in DNF specifies an independent initial condition at the initial location ℓ^*, which is processed separately.*

We define the directed graph $DG(\Gamma)$ of a LinTS Γ as the graph in which the vertices are the locations of Γ and there is an edge (ℓ, ℓ') iff there is a transition $\langle \ell, \ell', \rho\rangle$ with source location ℓ and target location ℓ'. To describe the semantics of a LinTS, we further define the notions of valuations, configurations and their associated satisfaction relation as follows.

A *valuation* over a finite set V of variables is a function $\sigma : V \to \mathbb{R}$ that assigns to each variable $x \in V$ a real value $\sigma(x) \in \mathbb{R}$. In this work, we mainly consider valuations over the variables X of a LinTS and simply abbreviate "valuation over X" as "valuation" (i.e., omitting X). Given a LinTS, a *configuration* is a pair (ℓ, σ) with the intuition that ℓ is the current location and σ is a valuation that specifies the current values for the variables. For the sake of convenience, we always assume an implicit linear order over the variable set V and treat each valuation σ over V equivalently as a real vector so that its ith coordinate $\sigma[i]$ is the value for the ith variable in the linear order.

We introduce the following satisfaction relations. Given a linear assertion φ over a variable set V and a valuation σ, we write $\sigma \models \varphi$ to mean that σ satisfies φ, i.e., φ is true when one substitutes the corresponding values $\sigma(x)$ to all the variables x in φ. Given a LinTS Γ, two valuations σ, σ' (over X) and a linear assertion φ over $X \cup X'$, we write $\sigma, \sigma' \models \varphi$ to mean that φ is true when one substitutes every variable $x \in X$ by $\sigma(x)$ and every variable $x' \in X'$ by $\sigma'(x)$ in φ. Moreover, given

two linear assertions φ, ψ over a variable set V, we write $\varphi \models \psi$ to mean that φ implies ψ, i.e., for every valuation σ over V we have that $\sigma \models \varphi$ implies $\sigma \models \psi$.

Below we describe the semantics of linear transition systems. Formally, the semantics of a LinTS Γ is specified by the notion of paths. A *path* π of the LinTS Γ is a finite sequence of configurations $(\ell_0, \sigma_0) \dots (\ell_k, \sigma_k)$ such that

- (**Initialization**) $\ell_0 = \ell^*$ and $\sigma_0 \models \theta$, and
- (**Consecution**) for every $0 \leq j \leq k-1$, there exists a transition $\tau = \langle \ell, \ell', \rho \rangle$ such that $\ell = \ell_j$, $\ell' = \ell_{j+1}$ and $\sigma_j, \sigma_{j+1} \models \rho$.

We say that a configuration (ℓ, σ) is *reachable* if there exists a path $(\ell_0, \sigma_0) \dots (\ell_k, \sigma_k)$ such that $(\ell_k, \sigma_k) = (\ell, \sigma)$. Intuitively, a path starts with some legitimate initial configuration (as specified by **Initialization**) and proceeds by repeatedly applying the transitions to the current configuration (as described in **Consecution**). Thus, any path $\pi = (\ell_0, \sigma_0) \dots (\ell_k, \sigma_k)$ corresponds to a possible execution of the underlying LinTS. Informally, a LinTS starts at the initial location ℓ^* with an arbitrary initial valuation σ^* such that $\sigma^* \models \theta$, constituting an initial configuration (ℓ_0, σ_0); then at each step j ($j \geq 0$), given the current configuration (ℓ_j, σ_j), the LinTS determines the next configuration $(\ell_{j+1}, \sigma_{j+1})$ by first selecting a transition $\tau = \langle \ell, \ell', \rho \rangle$ such that $\ell = \ell_j$ and then choosing $(\ell_{j+1}, \sigma_{j+1})$ to be any configuration that satisfies $\ell_{j+1} = \ell'$ and $\sigma_j, \sigma_{j+1} \models \rho$.

In the following, we assume that the guard condition ρ of each transition in a LinTS is a linear assertion. This follows from the fact that one can always transform the guard condition into a DNF and then split the transition into multiple sub-transitions where the guard condition of each sub-transition is a linear assertion that is a disjunctive clause of the DNF; a small detail here is that to handle strict inequalities such as $\alpha < \beta$ which arise from taking the negation of a non-strict linear inequality, we either have the over-approximation $\alpha \leq \beta$ or tighten it as $\alpha \leq \beta - 1$ in the integer case (i.e., every variable is integer valued, and every coefficient is an integer).

Linear transition systems can be used to model the transitions between the program states of a program with affine conditions and updates. In this work, we follow the pattern of *path dependency automata* (PDAs) proposed by Xie et al. [83] that each location of a LinTS corresponds to the choices of truth values (i.e., either true or false) of the conditional branches the loop body of a while loop. Below we present an example on the PDA pattern.

```
x = 0;
y = 50;
while (x < 100) {
    x = x + 1;
    if (x > 50)
        y = y + 1;
}
```

(a) The source code

$$X = \{x, y\}, L = \{\ell_1, \ell_2, \ell_e\}, \mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6\}, \theta : x = 0 \wedge y = 50, \tau_1 : \langle \ell_1, \ell_1, \rho_1 \rangle,$$
$$\tau_2 : \langle \ell_1, \ell_2, \rho_2 \rangle, \tau_3 : \langle \ell_1, \ell_e, \rho_3 \rangle, \tau_4 : \langle \ell_2, \ell_2, \rho_4 \rangle, \tau_5 : \langle \ell_2, \ell_1, \rho_5 \rangle, \tau_6 : \langle \ell_2, \ell_e, \rho_6 \rangle,$$
$$\rho_1 : \begin{bmatrix} 50 \leq x \leq 99 \\ 50 \leq x' \leq 99 \\ x' = x + 1 \\ y' = y + 1 \end{bmatrix}, \rho_2 : \begin{bmatrix} 50 \leq x \leq 99 \\ x' \leq 49 \\ x' = x + 1 \\ y' = y + 1 \end{bmatrix}, \rho_3 : \begin{bmatrix} 50 \leq x \leq 99 \\ 100 \leq x' \\ x' = x + 1 \\ y' = y + 1 \end{bmatrix},$$
$$\rho_4 : \begin{bmatrix} x \leq 49 \\ x' \leq 49 \\ x' = x + 1 \\ y' = y \end{bmatrix}, \rho_5 : \begin{bmatrix} x \leq 49 \\ 50 \leq x' \leq 99 \\ x' = x + 1 \\ y' = y \end{bmatrix}, \rho_6 : \begin{bmatrix} x \leq 49 \\ 100 \leq x' \\ x' = x + 1 \\ y' = y \end{bmatrix}$$

(b) The LinTS

Fig. 1. An affine loop from Sharma et al. [75] and its corresponding LinTS

Example 1. *Consider an affine program in Figure 1a taken from Sharma et al. [75]. The program first initializes the values of the variables x, y to $0, 50$, respectively. In the loop body, the execution passes through the **if**-branch and increments the value of y if the value of the variable x at the branch is greater than 50 (i.e., requiring that the value of x is no less than 50 at the start of the loop body);*

otherwise, the execution goes through the ***else****-branch (omitted in the program of Figure 1a) and does nothing if the value of* x *is no greater than* 50 *(i.e., requiring that the value of* x *is no greater than* 49 *at the start of the loop body).*

In Figure 1b, we present a LinTS as a transformation of the program in Figure 1a that follows the PDA pattern by Xie et al. [83]. The details are as follows.

- *We have the set of variables* $X = \{x, y\}$ *and the initial condition* θ *corresponds to the initialization of the variables before the while loop in Figure 1a.*
- *We have three locations. The location* ℓ_1 *(resp.* ℓ_2*) represents the execution in the loop body that passes through the* ***if****-branch (resp.* ***else****-branch), and the location* ℓ_e *is the termination location after the whole execution of the while loop. The initial location is* ℓ_2 *since under the initial condition* θ*, the first execution of the loop body does not pass into the* ***if****-branch.*
- *We have six transitions, namely* $\tau_1, \ldots, \tau_6$*. The transition* τ_1 *specifies the situation that the executions of both the current and the next loop iterations pass into the* ***if****-branch, for which the guard condition* ρ *is the conjunction of* $50 \leq x \leq 99$ *(which specifies the condition for the values at the start of the current loop iteration derived from the loop guard and the pass into the* ***if****-branch) and* $50 \leq x' \leq 99$ *(which specifies the similar condition for the values at the start of the next loop iteration). The transitions* τ_3, τ_4, τ_5 *can be derived by similar reasonings (i.e., from the loop guard and the pass into the* ***if****/****else****-branch) to the transition* τ_1*. Note that, the transitions* τ_2, τ_6 *will be rejected. Since the transition* τ_6 *is infeasible to jump from* ℓ_2 *to* ℓ_e*, as the condition to pass into the* ***else****-branch is* $x \leq 49$*, making a jump out of the while loop after the current loop iteration impossible. The transition* τ_2 *from* ℓ_1 *to* ℓ_2 *is infeasible following a similar argument.*

A path of the LinTS that describes the whole execution of the while loop is $(\ell_2, (x, y) = (0, 50)), \ldots,$ $(\ell_2, (49, 50)), (\ell_1, (50, 50)), \ldots, (\ell_1, (99, 99)), (\ell_e, (100, 100))$. □

Below we define invariants over linear transition systems. An *invariant* at a location ℓ of a LinTS is a logical formula φ such that for every path $\pi = (\ell_0, \sigma_0) \ldots (\ell_k, \sigma_k)$ of the LinTS and each $0 \leq i \leq k$, it holds that $\ell_i = \ell$ implies $\sigma_i \models \varphi$. An invariant φ is *(conjunctively) linear* if φ is a linear assertion over the variable set X, and is *disjunctively linear* if φ is a PLP in DNF over the variable set X. Intuitively, an invariant φ at a location ℓ is a logical formula in any form that over-approximates the set of reachable configurations at ℓ; the invariant is linear if it is in the form of a linear assertion, and disjunctively linear if it is in the form of a disjunction of linear assertions.

To automatically generate invariants, one often investigates a strengthened notion called *inductive invariants*. In this work, we present inductive linear invariants in the form of inductive linear assertion maps [18, 57, 71] as follows. We say that a *linear assertion map* (LAM) over a LinTS is a function η that maps every location ℓ of the LinTS to a linear assertion $\eta(\ell)$ over the variables X. Then an LAM η is *inductive* if the following conditions hold:

- (**Initialization**) $\theta \models \eta(\ell^*)$;
- (**Consecution**) For every transition $\tau = \langle \ell, \ell', \rho \rangle$, we have that $\eta(\ell) \wedge \rho \models \eta(\ell')'$, where $\eta(\ell')'$ is the linear assertion obtained by replacing every variable $x \in X$ in $\eta(\ell')$ with its next-value counterpart $x' \in X'$.

Informally, an LAM is inductive if it is (i) implied by the initial condition given by θ at the initial location ℓ^* (i.e., **Initialization**) and (ii) preserved under the application of every transition (i.e., **Consecution**). By a straightforward induction on the length of a path under a LinTS, one could verify that the linear assertion in an inductive LAM is indeed an invariant. In the rest of the work, we focus on the automated synthesis of inductive LAMs, and the disjunctive linear invariants are obtained by taking a disjunction of relevant linear assertions in an LAM.

Sometimes we need to consider the LinTS $\Gamma[\ell, K]$ derived from a LinTS Γ, a location ℓ of Γ and a subset K of valuations. In detail, the LinTS $\Gamma[\ell, K]$ is obtained by having the location ℓ as the only

location, the self-loop transitions at ℓ (i.e., transitions $\langle \ell'', \ell', \rho \rangle$ in Γ such that $\ell'' = \ell' = \ell$) as the only transitions, and the initial condition as the subset K. Here we slightly abuse the type of the initial condition so that the initial condition can also be a subset of valuations. This will not cause any problem as we consider any initial condition equivalently as the set of valuations that satisfy it.

In this work, we also consider the problem of loop summary. Loop summary is the classical subject to generate logical formulas that over-approximate the relationship between the input and output of a while loop. Given a LinTS that describes the execution of a while loop, we denote by $X_{\mathsf{in}} := \{x_{\mathsf{in}} \mid x \in X\}$ a copy of input variables from X and $X_{\mathsf{out}} := \{x_{\mathsf{out}} \mid x \in X\}$ a copy of output variables. We write $\mathbf{x}_{\mathsf{in}}$ (resp. $\mathbf{x}_{\mathsf{out}}$) for the vector of input (resp. output) variables, respectively. With the designated termination location ℓ_e at the end of the while loop, a loop summary S is a logical formula $S(\mathbf{x}_{\mathsf{in}}, \mathbf{x}_{\mathsf{out}})$ with free variables $\mathbf{x}_{\mathsf{in}}, \mathbf{x}_{\mathsf{out}}$ such that for all paths $\pi = (\ell_0, \sigma_0) \dots (\ell_k, \sigma_k)$ such that $\ell_k = \ell_e$, we have $S(\sigma_0, \sigma_k)$.

2.2 Applying Farkas' Lemma to Linear Invariant Generation

Farkas' Lemma [31] is a classical theorem in the theory of linear inequalities and previous results have applied the theorem to linear invariant generation. In this work, we follow the previous results [18, 57, 71] and consider a variant form of Farkas' Lemma [72, Corollary 7.1h] as follows.

Theorem 2.1 (Farkas' Lemma). *Consider a linear assertion φ over a set $V = \{x_1, \dots, x_n\}$ of real-valued variables as in Figure 2a. When φ is satisfiable (i.e., there is a valuation over V that satisfies φ), it implies a linear inequality ψ as in Figure 2b (i.e., $\varphi \models \psi$) if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, \dots, \lambda_m$ such that (i) $c_j = \sum_{i=1}^{m} \lambda_i \cdot a_{ij}$ for all $1 \le j \le n$, and (ii) $d = \lambda_0 + \sum_{i=1}^{m} \lambda_i \cdot b_i$. Moreover, φ is unsatisfiable if and only if the inequality $-1 \ge 0$ (as ψ) can be derived from above.*

$$\varphi : \begin{array}{c} a_{11} \cdot x_1 + \cdots + a_{1n} \cdot x_n + b_1 \ge 0 \\ \vdots \qquad\qquad \vdots \qquad \vdots \\ a_{m1} \cdot x_1 + \cdots + a_{mn} \cdot x_n + b_m \ge 0 \end{array}$$

(a) φ in Farkas' Lemma

$$\psi : c_1 \cdot x_1 + \cdots + c_n \cdot x_n + d \ge 0$$

(b) ψ in Farkas' Lemma

$$\begin{array}{c|l} \lambda_0 & 1 \ge 0 \\ \lambda_1 & a_{11} \cdot x_1 + \cdots + a_{1n} \cdot x_n + b_1 \bowtie_1 0 \\ \vdots & \vdots \qquad\qquad \vdots \quad \vdots \\ \lambda_m & a_{m1} \cdot x_1 + \cdots + a_{mn} \cdot x_n + b_m \bowtie_m 0 \\ \hline & c_1 \cdot x_1 + \cdots + c_n \cdot x_n + d \ge 0 \leftarrow \psi \\ & -1 \ge 0 \leftarrow \mathit{false} \end{array} \quad \Big\} \varphi$$

(c) The Tabular Form for Farkas' Lemma [18, 71]

Fig. 2. The φ and ψ and Tabular Form for Farkas' Lemma

One direction of Farkas' Lemma is straightforward, as one easily sees that if we have a non-negative linear combination of the inequalities in φ that can derive ψ, then it is guaranteed that ψ holds whenever φ is true. Farkas' Lemma further establishes that the other direction is also valid. In general, Farkas' Lemma simplifies the inclusion of a polyhedron inside a halfspace into the satisfiability of a system of linear inequalities.

Remark 1. *In the statement of Farkas' Lemma above, if we strengthen a linear inequality $a_{j1}x_1 + \cdots + a_{jn}x_n + b_j \ge 0$ in φ to equality (i.e., $a_{j1}x_1 + \cdots + a_{jn}x_n + b_j = 0$), then the theorem holds with the relaxation that we do not require $\lambda_j \ge 0$. This could be observed by first replacing the equality equivalent with both $a_{j1}x_1 + \cdots + a_{jn}x_n + b_j \ge 0$ and $a_{j1}x_1 + \cdots + a_{jn}x_n + b_j \le 0$, and then applying Farkas' Lemma. By similar arguments, the theorem statement holds upon changing multiple linear inequalities into equalities with the relaxation of non-negativity for their corresponding λ_j's.*

The application of Farkas' Lemma can be visualized in Figure 2c (taken from Colón et al. [18]), where $\bowtie_1, \ldots, \bowtie_m \in \{=, \geq\}$ and we multiply $\lambda_0, \lambda_1, \ldots, \lambda_m$ with their inequalities in φ and sum up them together to get ψ. For $1 \leq j \leq m$, if $\bowtie_j$ is $\geq$, we require $\lambda_j \geq 0$, otherwise (i.e., $\bowtie_j$ is $=$) we do not impose restriction on λ_j.

Below we review the existing approaches [18, 57, 71] that apply Farkas' Lemma to (conjunctive) linear invariant generation. We first present a high-level description, and then the technical details. All these existing approaches follow the template-based paradigm as follows:

- Establish a linear template with unknown coefficients over the input LinTS that represents the inductive LAM to be solved.
- Apply the initiation and consecution conditions to the template to obtain the constraints for an LAM.
- Use Farkas' Lemma to simplify the constraints obtained in the previous step.
- Solve the simplified constraints from the previous step to obtain concrete solutions to the unknown coefficients in the template. Each solution corresponds to one inductive LAM for the input LinTS.

Below we present the technical details of the steps above as (**Step A1** – **Step A4**) below. We fix an input LinTS with variable set $X = \{x_1, \ldots, x_n\}$.

Step A1. In the first step, all the existing approaches establish a template for an inductive LAM. A template η involves a linear inequality $\eta(\ell) = c_{\ell,1} \cdot x_1 + \cdots + c_{\ell,n} \cdot x_n + d \geq 0$ at each location ℓ of the LinTS with the unknown coefficients $c_{\ell,1}, \ldots, c_{\ell,n}, d$ to be resolved. Note that, although there is only one template at each location, the approaches can obtain a conjunctive linear invariants where one solution of the unknown coefficients corresponds to one conjunctive linear inequality.

Step A2. In the second step, all the approaches establish constraints from the initialization and the consecution conditions for an inductive invariant. Recall that the initialization condition specifies that the linear inequality $\eta(\ell^*)$ at the initial location ℓ^* should be implied by the initial condition θ, i.e., $\theta \models \eta(\ell^*)$, and the consecution condition specifies that every transition preserves the linear assertion map η, i.e., for every transition $\langle \ell, \ell', \rho \rangle$ we have that $\eta(\ell) \wedge \rho \models \eta(\ell')'$.

Step A3. In the third step, all the approaches apply Farkas' Lemma to the constraints collected from the initialization condition $\theta \models \eta(\ell^*)$ and the consecution condition $\eta(\ell) \wedge \rho \models \eta(\ell')'$ for every transition $\langle \ell, \ell', \rho \rangle$. For initialization, we apply the tabular in Figure 2c to obtain Figure 3a which results in a linear assertion over the unknown coefficients $c_{\ell^*,1}, \ldots, c_{\ell^*,n}, d$ and the fresh variables $\lambda_0, \lambda_1, \ldots, \lambda_m$. Similarly, the tabular applied to the consecution condition of a transition $\langle \ell, \ell', \rho \rangle$ gives Figure 3b where in addition to $\lambda_j, c_{\ell,j}, d_\ell, c_{\ell',j}, d_{\ell'}$ we have a fresh variable μ as the non-negative multiplier for $\eta(\ell)$. Note that for the consecution condition, the constraint obtained is no longer linear since the fresh variable μ is multiplied to $\eta(\ell)$ in the tabular.

Step A4. In the last step, the (non-linear) constraints from the previous step are solved to obtain the concrete values for the unknown coefficients in η, so that a concrete inductive LAM would be obtained. It is from this point on that the existing approaches become diverse:

- By Colón et al. [18], the non-linear constraints were solved through the complete but costly method of quantifier elimination;
- By Sankaranarayanan et al. [71], the non-linear constraints were solved through (i) several reasonable heuristics to guess possible values for the key parameter μ in Figure 3b so as to remove the non-linearity and obtain a linear under-approximation of the original non-linear constraints, and (ii) the generator computation over polyhedral cones to obtain the invariants. A major heuristics there is to guess possible values for μ through some practical rules such

$$\begin{array}{l|l}
\lambda_0 & 1 \geq 0 \\
\lambda_1 & a_{11}x_1+\cdots+a_{1n}x_n+b_1 \bowtie_1 0 \\
\vdots & \vdots \quad\quad \vdots \quad \vdots \\
\lambda_m & a_{m1}x_1+\cdots+a_{mn}x_n+b_m \bowtie_m 0 \\
\hline
 & c_{\ell^*,1}x_1+\cdots+c_{\ell^*,n}x_n+d_{\ell^*} \geq 0 \leftarrow \eta(\ell^*) \\
 & -1 \geq 0 \leftarrow \mathit{false}
\end{array} \quad \theta$$

(a) Initialization Tabular

$$\begin{array}{l|l}
\mu & c_{\ell,1}x_1+\cdots+c_{\ell,n}x_n + d_\ell \geq 0 \leftarrow \eta(\ell) \\
\lambda_0 & 1 \geq 0 \\
\lambda_1 & a_{11}x_1+\cdots+a_{1n}x_n+a'_{11}x'_1+\cdots+a'_{1n}x'_n+b_1 \bowtie_1 0 \\
\vdots & \vdots \quad\quad \vdots \quad\quad \vdots \quad \vdots \\
\lambda_m & a_{m1}x_1+\cdots+a_{mn}x_n+a'_{m1}x'_1+\cdots+a'_{mn}x'_n+b_m \bowtie_m 0 \\
\hline
 & c_{\ell',1}x'_1+\cdots+c_{\ell',n}x'_n+d_{\ell'} \geq 0 \leftarrow \eta(\ell')' \\
 & -1 \geq 0 \leftarrow \mathit{false}
\end{array} \quad \rho$$

(b) Consecution Tabular

Fig. 3. The Tabular Form for Initialization and Consecution [18, 71]

as factorization and setting μ manually to 0, 1 (where 0 means an invariant local to the guard of the transition and 1 means one incremental to the previous execution).

- By Liu et al. [57], a substantial improvement on the scalability to the approach by Sankaranarayanan et al. [71] is proposed by generating linear invariants one location at time. The main advantage of this approach is that redundant invariants can be detected more efficiently in the solving of the constraints.

Below we showcase the application of Farkas' Lemma to linear invariant generation by an example. We focus on the approaches by Liu et al. [57], Sankaranarayanan et al. [71] that have a better scalability.

Example 2. *Consider the LinTS in Figure 1b. The approach [71] first establishes a template at each location by setting* $\eta(\ell_i) := c_{\ell_i,1}x + c_{\ell_i,2}y + d_{\ell_i} \geq 0$ *for* $i \in \{1, 2, e\}$ ***(Step A1)****. Then, it generates the constraints from the initialization and consecution conditions* ***(Step A2)*** *and simplifies the constraints by the tabular in Figure 3* ***(Step A3)****. Note that, the fresh variables* λ_j*'s and* μ *in an instantiation of the tabular are local and do not overlap with the fresh variables in other instantiations. For initialization, the tabular in Figure 4a gives the simplified constraints* $[c_{\ell_2,1} = \lambda_1, c_{\ell_2,2} = \lambda_2, d_{\ell_2} \geq -50\lambda_2]$ *(recall Remark 1, where* $\lambda_0 \geq 0$ *but we do not impose restriction on* λ_1, λ_2*) and finally generates the constraints* $[50c_{\ell_2,2}+d_{\ell_2} \geq 0]$ *by projecting away the fresh variables* λ_j*'s. For consecution, we present the application of the tabular to the transition* τ_5 *as in Figure 4b; after projecting away the fresh variables* λ_j*'s, the approach further eliminates the fresh variable* μ *by heuristics that guess its value through either some practical rules such as factorization or setting* μ *manually to* 0, 1*. Other transitions are treated in a similar way.*

$$\begin{array}{l|l}
\lambda_0 & 1 \geq 0 \\
\lambda_1 & x = 0 \\
\lambda_2 & y - 50 = 0 \\
\hline
 & c_{\ell_2,1}x + c_{\ell_2,2}y + d_{\ell_2} \geq 0 \leftarrow \eta(\ell_2)
\end{array} \quad \theta$$

(a) Initialization Tabular in Example 2

$$\begin{array}{l|l}
\mu & c_{\ell_2,1}x + c_{\ell_2,2}y + d_{\ell_2} \geq 0 \leftarrow \eta(\ell_2) \\
\lambda_0 & 1 \geq 0 \\
\lambda_1 & -x + 49 \geq 0 \\
\lambda_2 & x' - 50 \geq 0 \\
\lambda_3 & -x' + 99 \geq 0 \\
\lambda_4 & -x + x' - 1 = 0 \\
\lambda_5 & -y + y' = 0 \\
\hline
 & c_{\ell_1,1}x' + c_{\ell_1,2}y' + d_{\ell_1} \geq 0 \leftarrow \eta(\ell_1)' \\
 & -1 \geq 0 \leftarrow \mathit{false}
\end{array} \quad \rho_5$$

(b) Consecution Tabular in Example 2

Fig. 4. The Tabular Form of Initialization and Consecution in Example 2

The simplified constraints obtained from the previous step constitutes a PLP Φ *in CNF where each conjunctive clause is the constraint derived from either the initialization or the consecution of a transition, and every disjunctive linear assertion in such a conjunctive clause results from a distinct*

*guessed value for a non-linear μ parameter in the tabular for consecution. In the last step (**Step A4**), the approach [71] expands the PLP Φ equivalently into a DNF PLP (where each disjunctive clause is a linear assertion that defines a polyhedral cone) and obtains the linear invariants by the generator computation of each polyhedral cone in the DNF PLP. For this example, one disjunctive clause (treated as a polyhedral cone) from the DNF formula is shown in Figure 5a (where we abbreviate $c_{\ell_i,j}$, d_{ℓ_i} as c_{ij}, d_i); further by computing the generators of the polyhedral cone in Figure 5a, we obtain the corresponding generators and their invariants in Figure 5b, where in the left part each row specifies a generator with a type (a point generator or a ray generator or a line generator) over the unknown coefficients c_{ij}'s and d_i's, and in the right part we instantiate the generator to the unknown coefficients in the template η to obtain the invariants at location ℓ_1, ℓ_2 and ℓ_e.*

$$\begin{bmatrix} c_{12} - c_{e2} = 0, c_{12} - c_{22} = 0, \\ c_{21} \geq 0, c_{11} + c_{12} \geq 0, \\ 50c_{12} + d_2 \geq 0, \\ -99c_{11} + c_{12} - d_1 + 100c_{e1} + d_e \geq 0, \\ 50c_{11} + d_1 - 49c_{21} - d_2 \geq 0 \end{bmatrix}$$

(a) A disjunctive clause in the DNF

type	c_{11}	c_{12}	d_1	c_{21}	c_{22}	d_2	c_{e1}	c_{e2}	d_e	$\eta(\ell_1)$	$\eta(\ell_2)$	$\eta(\ell_e)$
point	0	0	0	0	0	0	0	0	0	$0 \geq 0$	$0 \geq 0$	$0 \geq 0$
line	1	−1	0	0	−1	50	0	−1	100	$x - y = 0$	$-y + 50 = 0$	$-y + 100 = 0$
line	0	0	0	0	0	0	1	0	−100	$0 = 0$	$0 = 0$	$x - 100 = 0$
ray	0	0	49	1	0	0	0	0	49	$49 \geq 0$	$x \geq 0$	$49 \geq 0$
ray	0	0	0	0	0	0	0	0	1	$0 \geq 0$	$0 \geq 0$	$1 \geq 0$
ray	0	0	1	0	0	0	0	0	1	$1 \geq 0$	$0 \geq 0$	$1 \geq 0$
ray	1	0	−50	0	0	0	0	0	49	$x - 50 \geq 0$	$0 \geq 0$	$49 \geq 0$
ray	0	0	1	0	0	1	0	0	1	$1 \geq 0$	$1 \geq 0$	$1 \geq 0$

(b) generators (left) and their invariants (right) for Figure 5a

Fig. 5. Example of a disjunctive clause and its generators and invariants

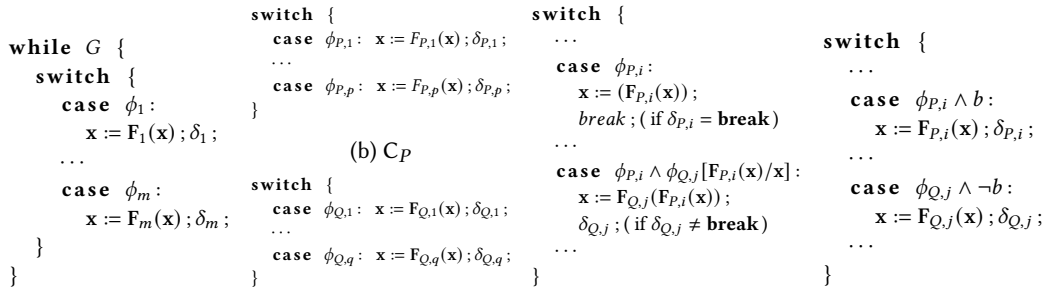
The linear invariants obtained from the generator computation can be further minimized by removing trivial invariants such as $0 \geq 0$ and redundant inequalities. After processing all the disjunctive clauses of the DNF and grouping all the generated invariants together, the final invariants generated are $\eta(\ell_1) = [x = y, 50 \leq x \leq 99]$, $\eta(\ell_2) = [y = 50, 0 \leq x \leq 49]$ and $\eta(\ell_e) = [x = y = 100]$. The approach [57] further improves the scalability by the idea of generating the invariants one location at a time that allows to detect redundant invariants more efficiently. □

3 DISJUNCTIVE LINEAR INVARIANT GENERATION FOR UNNESTED LOOPS

In this section, we present our approach for generating linear disjunctive loop invariants over unnested affine while loops. Throughout the section, we fix the set of program variables as $X = \{x_1, \ldots, x_n\}$ and identify the set X as the set of variables in the LinTS to be derived from the loop. We consider the canonical form of an unnested affine while loop as in Figure 6a, where we have:

- The PLP G is the loop condition (or loop guard) for the while loop.
- The vector $\mathbf{x} = (x_1, \ldots, x_n)^\mathrm{T}$ represents the column vector of program variables, and each $\mathbf{F}_i$ $(1 \leq i \leq m)$ is an affine function, i.e., $\mathbf{F}_i(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$ where $\mathbf{A}$ (resp. $\mathbf{b}$) is an $n \times n$ square matrices (resp. n-dimensional column vector) that specifies the affine update under the linear assertion ϕ_i (as a conditional branch). The assignment $\mathbf{x} := \mathbf{F}_i(\mathbf{x})$ is considered simultaneously for the variables in $\mathbf{x}$ so that in one execution step, the current valuation σ is updated to $\mathbf{F}_i(\sigma)$.
- The **switch** keyword represents a special conditional branching (i.e., different from its original meaning in e.g. C programming language) that if the current values of the program variables satisfy the condition ϕ_i, then the assignment at the ith conditional branch (i.e., $\mathbf{x} := \mathbf{F}_i(\mathbf{x})$) is executed. Note that the branch conditions $\phi_1, \ldots, \phi_m$ need not to be logically pairwise disjoint (i.e., there can be some valuation σ that satisfies both ϕ_i, ϕ_j $(i \neq j)$, so that our setting covers nondeterminism in imperative programs.

- The statements $\delta_1, \ldots, \delta_m$ specify whether the loop continues after the affine update of the conditional branches $\phi_1, \ldots, \phi_m$. Each statement δ_i is either the **skip** statement that does nothing (which means that the loop continues after the affine update of $\mathbf{F}_i$) or the **break** statement (which means that the loop exits after the affine update).



(a) Canonical form (b) C_P (c) C_Q (d) The sequential case (e) The conditional case

Fig. 6. The canonical form of unnested loop and transformation (TF) for P, Q with two recursive cases

The canonical form in Figure 6a coincides with the PDA pattern [83] in the sense that each branch condition ϕ_i encodes the truth values of all the conditional branches in the original loop body, so that each ϕ_i corresponds to an execution of the whole loop body. Note that we do not consider the **goto** statement since they would otherwise lead to infinite executions within the loop body, and thus breaking the loop structure.

Any unnested affine while loop with break statement can be transformed into the canonical form in Figure 6a by recursively examining the substructures of the loop body of the loop, at the cost of a possible exponential blow-up in the number of conditional branches in the loop body. A recursive algorithm that transforms an affine program P with break statement and without loops into a program C_P that fits the loop body of the canonical form in Figure 6a, could work as follows.

- For the base case where the program P is either a single affine assignment $\mathbf{x} := \mathbf{F}(\mathbf{x})$ or resp. the **break** statement, the transformed program C_P is simply **switch** {**case true** : $\mathbf{x} := \mathbf{F}(\mathbf{x})$; **skip**; } or resp. **switch** {**case true** : $\mathbf{x} := \mathbf{x}$; **break**; }, respectively.
- For a sequential composition $R = P; Q$, the algorithm recursively computes C_P and C_Q as in Figure 6b and Figure 6c respectively, and then compute C_R as in Figure 6d for which:
  - For each $1 \leq i \leq p$ such that $\delta_{P,i} =$ **break**, we have the branch $\mathbf{x} := \mathbf{F}_{P,i}(\mathbf{x})$; **break**; (i.e., the branch breaks already in the execution of P).
  - For each $1 \leq i \leq p$ and $1 \leq j \leq q$ such that $\delta_{P,i} =$ **skip**, we have the branch $\mathbf{x} := \mathbf{F}_{Q,j}(\mathbf{F}_{P,i}(\mathbf{x}))$; $\delta_{Q,j}$ under the branch condition $\phi_{P,i} \wedge (\phi_{Q,j}[\mathbf{F}_{P,i}(\mathbf{x})/\mathbf{x}])$ (i.e., the branch continues to the execution of Q).
- For a conditional branch $R =$ **if** b **then** P **else** Q, the algorithm recursively computes C_P and C_Q as in the previous case, and then compute C_R as in Figure 6e for which:
  - For each $1 \leq i \leq p$, we have the branch $\mathbf{x} = \mathbf{F}_{P,i}(\mathbf{x}); \delta_{P,i}$; with branch condition $\phi_{P,i} \wedge b$ (i.e., the branch conditions of P is conjuncted with the extra condition b).
  - For each $1 \leq j \leq q$, we have the branch $\mathbf{x} = \mathbf{F}_{Q,j}(\mathbf{x}); \delta_{Q,j}$; with branch condition $\phi_{Q,j} \wedge \neg b$ (i.e., the branch conditions of Q is conjuncted with the extra condition $\neg b$).

Note that although the transformation into our canonical form may cause exponential blow up in the number of conditional branches in the loop body, in practice a loop typically has a small number of conditional branches and further improvement can be carried out by removing invalid

branches (i.e., those whose branch condition is unsatisfiable). Moreover, such a canonical form is often necessary to derive precise disjunctive information for a while loop (see e.g. Xie et al. [83]).

Below we illustrate our algorithm to generate disjunctive linear invariants on unnested affine while loops. Informally, our algorithm applies the PDA pattern from Xie et al. [83] and follows Farkas' Lemma for linear invariant generation as Colón et al. [18], Liu et al. [57], Sankaranarayanan et al. [71], and further propose the improvement of invariant generation that is closely related to the PDA pattern and has not been considered in the existing approaches [18, 57, 71]. We fix an unnested affine while loop W. The workflow of our algorithm is demonstrated as follows (**Step B1** – **Step B3**).

Step B1. We first transform the loop W into a canonical form C_W w.r.t Figure 6a as stated previously.

Step B2. Then we apply the PDA pattern to transform the loop C_W into a LinTS. The transformation is in a straightforward fashion that every conditional branch (i.e., φ_i in Figure 6a) corresponds to a stand-alone location, and the guard of a transition is determined by the loop condition (i.e., G) and the branch conditions of the source and target locations of the transition. Formally, we have that the LinTS Γ_W derived from the loop W is given as follows:

- The set of locations is $\{\ell_1, \ldots, \ell_m, \ell_e\}$ where each ℓ_i $(1 \leq i \leq m)$ corresponds to the branch with branch condition φ_i and ℓ_e is the termination program counter of the loop.
- For each $1 \leq i, j \leq m$, we have the transition (where we denote $\mathbf{x}' := (x'_1, \ldots, x'_n)^{\mathrm{T}}$)

$$\tau_{ij} = (\ell_i, \ell_j, G \wedge \varphi_i \wedge \varphi_j[\mathbf{x}'/\mathbf{x}] \wedge \mathbf{x}' = \mathbf{F}_i(\mathbf{x}))$$

  that specifies the one-step jump from the location ℓ_i to the location ℓ_j, for which the guard condition is $G \wedge \varphi_i \wedge \varphi_j[\mathbf{x}'/\mathbf{x}] \wedge \mathbf{x}' = \mathbf{F}_i(\mathbf{x})$ since the transition needs to pass the loop guard G, satisfy the branch condition φ_i when staying in the location ℓ_i, have the affine update specified by $\mathbf{F}_i$ and fulfill the branch condition φ_j upon entering the location ℓ_j.
- For each $1 \leq i \leq m$, we have the transition

$$\tau'_i = (\ell_i, \ell_e, G \wedge \varphi_i \wedge (\neg G)[\mathbf{x}'/\mathbf{x}] \wedge \mathbf{x}' = \mathbf{F}_i(\mathbf{x}))$$

  that specifies the one-step jump from the location ℓ_i to the termination location ℓ_e for which the guard condition is a conjunction of the loop guard G (for staying in the loop at the current loop iteration), the branch condition φ_i (that the current execution of the loop body follows the location ℓ_i), the equality $\mathbf{x}' = \mathbf{F}_i(\mathbf{x})$ (for the affine update) and the negation of the loop guard (for jumping out of the loop).

After the transformation, we remove transitions with unsatisfiable guard condition to reduce the size of the derived LinTS. An example for the transformation has been given in Example 1.

Remark 2. *It is possible to derive more precise linear transition systems for affine while loops by considering modulus information of integer program values (such as the parity of program values).*

Step B3. After the transformation into a LinTS, we follow existing approaches [18, 57, 71] that generate linear invariants with Farkas' Lemma. In particular, we apply the recent approach [57] that has the most scalability. The major difference is that we consider an invariant propagation technique that takes advantage of a common feature in the PDA pattern to further improve the time efficiency. Besides, a slight difference is that we do not encode the constraints for the termination program location ℓ_e. This is because the invariant at ℓ_e can be derived by the invariant propagation technique that propagates the invariants from non-termination locations to the termination location.

In our invariant propagation, we explore a special structure in the derived LinTS that often arises in the PDA pattern, and propose a technique that applies to the special structure and allows one to generate invariants at only one location and obtain the invariants at other locations through a

propagation process. To illustrate the invariant propagation, we first identify the special structure of non-crossing linear transition systems.

Definition 2. *A LinTS* Γ *is* non-crossing *if a depth-first search (DFS) tree of the directed graph* $DG(\Gamma)$ *(i.e., its underlying directed graph) rooted at the initial location does not have cross edges. (Recall that a cross edge in a DFS tree is an edge whose destination location is a visited location in the DFS but not an ancestor of the source location of the edge).*

Non-crossing linear transition systems are common in the PDA pattern. For example, the case of multiphase invariants [75] is a special case of non-crossing linear transition systems where a location is never entered again once it is left. Moreover, the specific PDA patterns considered by Xie et al. [83] require the strict alternation between locations (which is necessary for their approach to obtain precise loop summaries), and hence is also a special case of non-crossing linear transition systems. Furthermore, any linear transition system that has one outgoing-transition for every location (which arises from deterministic mode change in while loops) is non-crossing.

Then we illustrate the main workflow of our invariant propagation technique. Consider a LinTS Γ transformed from an unnested affine while loop. Given a DFS tree T of $DG(\Gamma)$ rooted at the initial location ℓ^* that has the non-crossing property and a conjunctive linear invariant $\eta(\ell^*)$ at the location ℓ^* generated from the approach by Liu et al. [57], the invariant propagation works by repeatedly propagating the invariant $\eta(\ell^*)$ from the root to other locations in a breadth-first search (BFS) from the root ℓ^*. In the BFS, a single step of propagation from a location ℓ in the current BFS front with the invariant $\eta(\ell)$ (as a DNF PLP) computed from the prior BFS process to a location ℓ' in the next front, considers all transitions from ℓ to ℓ'; for each such transition $\tau = (\ell, \ell', \rho)$, our approach computes a DNF PLP as an invariant $I(\tau, \ell')$ for the LinTS $\Gamma[\ell', K_\tau := \{\sigma' \mid \exists \sigma.(\sigma \models \eta(\ell) \wedge \sigma, \sigma' \models \rho)\}]$ (see Page 6 for the definition of $\Gamma[-,-]$) again via the approach by Liu et al. [57] and disjuncts all these $I(\tau, \ell')$'s together to obtain $\eta(\ell')$. The invariant at the termination location ℓ_e is also obtained by performing a single propagation step from the non-termination locations.

The details of a single propagation in the BFS is as follows. Consider a location ℓ at the current BFS front with the computed PLP invariant $\eta(\ell) = \bigvee_{i=1}^{d} \Phi_i$ where each Φ_i is a linear assertion. Then for each transition $\tau = (\ell, \ell', \rho)$, we have that $I(\tau, \ell') = \bigvee_{i=1}^{d} I(\tau, \ell', i)$ where each $I(\tau, \ell', i)$ is a conjunctive linear invariant of the LinTS $\Gamma[\ell', K_{\tau,i} := \{\sigma' \mid \exists \sigma.(\sigma \models \Phi_i \wedge \sigma, \sigma' \models \rho)\}]$. Hence, our approach calculates $I(\tau, \ell')$ by computing for each $1 \leq i \leq d$ the conjunctive linear invariant $I(\tau, \ell', i)$ (over $\Gamma[\ell', K_{\tau,i}]$) by the approaches [57, 71]. To apply these approaches, we need to encode the set $K_{\tau,i}$ as a linear assertion $\Phi'_{\tau,i}$ without quantifiers that defines the set, and this can be accomplished by the projection of the polyhedron $\{(\sigma, \sigma') \mid \sigma \models \Phi_i \wedge \sigma, \sigma' \models \rho\}$ onto the dimensions of σ'. However, polyhedron projection is an operation with relatively high computation cost. Below we show that these $\Phi'_{\tau,i}$'s can be computed more efficiently by the resorting to the affine updates between $\mathbf{x}$ and $\mathbf{x}'$ from the original while loop.

Consider the task to project the polyhedron $H = \{(\sigma, \sigma') \mid \sigma \models \Phi \wedge \sigma, \sigma' \models \rho\}$ in the treatment of a transition $\tau = (\ell, \ell', \rho)$ stated above, where Φ is a linear assertion. Recall that the transition is derived in the way that the relationship between the variables from X and X' is given by some affine assignment $\mathbf{x} := \mathbf{A}\mathbf{x} + \mathbf{b}$ (i.e., $\mathbf{x}' = \mathbf{A}\mathbf{x} + \mathbf{b}$) under some conditional branch in the canonical form of Figure 6a. We consider two cases below.

- The first case is that the matrix $\mathbf{A}$ is invertible. In this case, we have that $\mathbf{x} = \mathbf{A}^{-1}\mathbf{x}' - \mathbf{A}^{-1}\mathbf{b}$, and we obtain a linear assertion Φ' over X' that defines the projected polyhedron directly as $(\Phi \wedge \rho)[(\mathbf{A}^{-1}\mathbf{x}' - \mathbf{A}^{-1}\mathbf{b})/\mathbf{x}]$. In this case, no polyhedron projection is needed.
- The second case is that the matrix A is not invertible. Then we solve the system of linear equations $\mathbf{A}\mathbf{x} = \mathbf{x}' - \mathbf{b}$ by the standard method of Gaussian Elimination in elementary linear algebra and obtains that $\mathbf{x} = \mathbf{u}(\mathbf{x}') + \sum_{i=1}^{k} a_k \cdot \mathbf{v}_i$ $(a_1, \ldots, a_k \in \mathbb{R})$ where (i) the vector $\mathbf{u}(\mathbf{x}')$

is a solution to the non-homogenous equation $\mathbf{Ax} = \mathbf{x}' - \mathbf{b}$ and can be expressed as an affine combination of the entries in $\mathbf{x}'$ (i.e., $\mathbf{u}(\mathbf{x}') = \mathbf{Cx}' + \mathbf{d}$ for some matrix $\mathbf{C}$ and vector $\mathbf{d}$) and (ii) $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are the basic solution of the homogeneous equation $\mathbf{Ax} = \mathbf{0}$ and are constant vectors not relying on $\mathbf{x}'$. The fresh variables $a_1, \ldots, a_k$ are the coefficients of the basic solution and can take any real value. As a consequence, the projection of the linear assertion $\sigma \models \Phi \wedge \sigma, \sigma' \models \rho$ (that defines the polyhedron H) onto the variables $\mathbf{x}'$ can be obtained as the projection of the linear assertion $(\Phi \wedge \rho)[(\mathbf{u}(\mathbf{x}') + \sum_{i=1}^{k} a_k \cdot \mathbf{v}_i)/\mathbf{x}]$ onto the variables $\mathbf{x}'$ (i.e., projecting away the dimensions of $a_1, \ldots, a_k$). Note that the number of the basic solution $a_1, \ldots, a_k$ is equal to $n - \text{rank}(A)$ where $\text{rank}(A)$ is the rank of the matrix A. This means that the number of variables to be projected away is smaller than n. It follows that in this case, it is possible to project away much less variables compared with the original projection method (that needs to project away all the n variables $x_1, \ldots, x_n$ in $\mathbf{x}$), and thus can further improve the time efficiency.

The advantage of incorporating invariant propagation lies at the observation that to generate the invariants at all the locations, previous approaches consider to solve them either as a whole [71] or separately [57] via the generator computation of polyhedral cones. Thus, all these approaches require to solve the invariants at all the locations with generator computation, an operation with relative high cost and possible exponential blow-up. Invariant propagation improves the time efficiency in that when the underlying LinTS has a non-crossing DFS tree, then it suffices to perform generator computation only in the computation of the invariants at the initial location and in the treatment of self-loops at other locations.

Note that non-crossing linear transition systems do not cover all cases of directed acyclic graphs, but this can be remedied by first computing the strongly-connected components (SCCs) of the underlying LinTS and then considering each SCC separately.

In summary, the workflow of our algorithm for generating linear invariants over an unnested affine while loop is as follows.

- First, our algorithm transforms an unnested affine while loop into the canonical form in Figure 6a and further transforms it into a linear transition system from the PDA pattern [83].
- Second, our algorithm applies the approach by Liu et al. [57] and our invariant propagation technique (if possible) to obtain linear invariants at the locations of the linear transition system. In the case that the linear transition system is non-crossing w.r.t the initial location, our algorithm applies the approach by Liu et al. [57] to obtain the linear invariant at the initial location and afterwards obtain the invariants at other locations through invariant propagation. Otherwise (i.e., the linear transition system is not non-crossing), our algorithm fully follows the original approach by Liu et al. [57] to generate the invariants at all the locations.

By an induction on the depth of the DFS tree, we can prove that the logical assertions generated from our invariant propagation are indeed invariants and are at least as tight as the invariants generated by the previous approaches [57, 71]. Due to space limitation, we relegate the detailed proofs to Appendix C.

4 DISJUNCTIVE LINEAR INVARIANT GENERATION FOR NESTED LOOPS

Recall that in the previous section, we proposed a novel approach for generating disjunctive linear invariants over unnested while loops via Farkas' Lemma, the PDA pattern and an invariant propagation technique. In this section, we extend this approach to nested affine while loops.

The main idea is as follows. Given a nested affine while loop W, our approach works by first recursively computing the loop summary $S_{W'}$ for each outermost nesting while loop W' in W (that

is a while loop which is in the loop body of W and does not lie in other nesting loops), and then apply the PDA pattern with the integration of these loop summaries $S_{W'}$. Below we fix a nested affine while loop W with variable set $X = \{x_1, \ldots, x_n\}$ and present the technical details.

The most involved part in our approach is the transformation of the loop W into its corresponding LinTS by the PDA pattern. Unlike the situation of unnested while loops, a direct recursive algorithm that transforms the loop W into a canonical form in Figure 6a as in the unnested case is not possible, since one needs to tackle the loop summaries from the outermost nesting while loops in W.

To address the problem above, our algorithm works with the *control flow graph* (CFG) H of the loop body of the loop W and considers the *execution paths* in this CFG. The CFG H is a directed graph whose vertices are the program counters of the loop body and whose edges describe the one-step jumps between these program counters. Except for the standard semantics for the jumps emitting from assignment statements and conditional branches, for a program counter that represents the entry point of an outermost nesting while loop, we have the special treatment that the jump at the program counter is directed to the termination program counter of the nesting while loop in the loop body of W (i.e., skipping the execution of the nesting while loop). An execution path in the CFG H is a directed path (of program locations) that ends in (i) either the termination program counter of the loop body of W without visiting a program counter that represents the **break** statement or (ii) a first **break** statement without visiting prior **break** statements. An example is as follows.

Example 3. *Consider the CFG of a specific example with two loop W and W' from* janne_complex *[9] in Figure 7. The execution path starts at the* Initial Condition $[x, y]$*, jumps to the next vertices along the edge whose condition is satisfied (e.g.,* True *is tautology, $x < 30$ is satisfied when variable x value is less than* 30*, etc.), and terminates in the* Exit *statement. One possible execution path for the outermost loop W is $E_{Outer} \rightarrow A_{IS} \rightarrow A_1 \rightarrow E_{Outer}$. The E_{Outer} means the outermost loop entry of W and E_{Inner} means the outermost-nesting-loop entry of W' in W. The A_1, A_2, A_3 represents the assignment statements in program and A_{IS} is a special assignment statement for the outermost-nesting-loop W' which could be obtained by computing loop summary for W'.* □

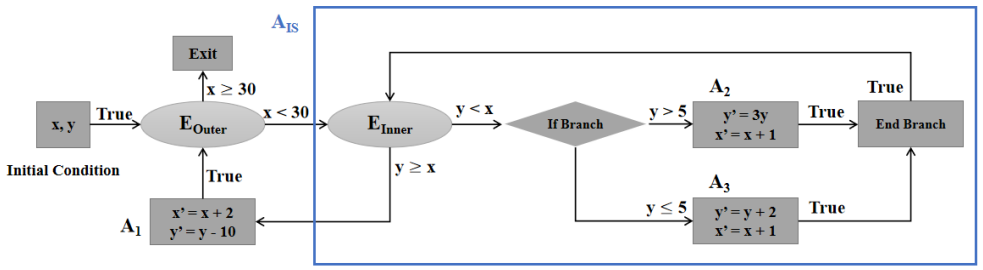


Fig. 7. The CFG of *janne_complex* [9]

Based on the CFG H and the execution paths, our approach constructs the LinTS for the whole loop W as follows. Since the output of an outermost nesting while loop W' in W cannot be exactly determined from the input to the loop W', we first have fresh variables $\overline{x}_{W',1}, \ldots, \overline{x}_{W',n'}$ to represent the output values of the variables $\overline{x}_{W',1}, \ldots, \overline{x}_{W',n}$ after the execution of the loop W'. With these fresh variables, we can then symbolically compute the values of the program variables at each program counter in an execution path. In detail, given an execution path $\omega = \iota_1, \ldots, \iota_k$ where each ι_i is a program counter of the loop body of the loop W, our approach computes the affine expressions $\alpha_{x,i}$ and PLPs β_i (for $x \in X$ and $1 \leq i \leq k$) over the program variables X (that represents their

initial values at the start of the loop body of W here) and the fresh variables (for the output of the outermost nesting loops). The intuition is that (i) each affine expression $\alpha_{x,i}$ represents the value of the variable x at the program counter ι_i along the execution path ω and (ii) each PLP β_i specifies the condition that the program counter ι_i is reached along the execution path ω. The computation is recursive on i as follows.

Denote the vectors $\alpha_i := (\alpha_{x_1,i}, \ldots, \alpha_{x_n,i})$ and $\overline{x}_{W'} = (\overline{x}_{W',1}, \ldots, \overline{x}_{W',n})$. For the base case when $i = 1$, we have $\alpha_1 = (x_1, \ldots, x_n)$ and $\beta_1 = \textbf{true}$ that specifies the initial setting at the start program counter ι_1 of the loop body of the original loop W. For the recursive case, suppose that our approach has computed the affine expressions in α_i and the PLP β_i. We classify four cases below:

- *Case 1:* The program counter ι_i is an affine assignment statement $\mathbf{x} := \mathbf{F}(\mathbf{x})$. Then we have that $\alpha_{i+1} = \alpha_i[\mathbf{F}(\mathbf{x})/\mathbf{x}]$ and $\beta_{i+1} := \beta_i$.
- *Case 2:* The program counter ι_i is a conditional branch with branch condition b and the next program counter ι_{i+1} follows its **then**-branch. Then the vector α_{i+1} is the same as α_i, and the PLP β_{i+1} is obtained as $\beta_{i+1} = \beta_i \wedge b$.
- *Case 3:* The program counter ι_i is a conditional branch with branch condition b and the next program counter ι_{i+1} follows its **else**-branch. The only difference between this case and the previous case is that β_{i+1} is obtained as $\beta_{i+1} := \beta_i \wedge \neg b$.
- *Case 4:* The program counter ι_i is the entry point of an outermost nesting while loop W' of W and ι_{i+1} is the successor program counter outside W' in the loop body of W. Then $\alpha_{i+1} := \overline{x}_{W'}$ and $\beta_{i+1} := S_{W'}(\alpha_i, \overline{x}_{W'})$. Note that in this case the loop summary $S_{W'}$ (see Page 6 for the definition of S) is recursively computed.

Example 4. *Continue with the execution path in Example 3. We show the evolution of $\alpha_{i,W}$ and $\beta_{i,W}$ with the initial setting $\alpha_{1,W} = [x, y], \beta_{1,W} = \textbf{true}$ in Figure 8 where W denotes the current loop.* □

$$\alpha_{1,W} = [x, y], \beta_{1,W} = \textbf{true} \xrightarrow{x < 30} \alpha_{2,W} = [x, y], \beta_{2,W} = \beta_{1,W} \wedge x < 30 \xrightarrow{A_{IS}}$$
$$\alpha_{3,W} = [\overline{x}_{W'}, \overline{y}_{W'}], \beta_{3,W} = \beta_{2,W} \wedge S_{W'}(\alpha_{2,W}, \alpha_{3,W}) \xrightarrow{A_1} \alpha_{4,W} = [\overline{x}_{W'} + 2, \overline{y}_{W'} - 10], \beta_{4,W} = \beta_{3,W}$$

Fig. 8. The evolution of $\alpha_{i,W}$ and $\beta_{i,W}$ for the execution path of W in Figure 7

After the α_i, β_i's are obtained for an execution path $\omega = \iota_1, \ldots, \iota_k$ from the recursive computation above, we let the PLP $\Psi_\omega := \bigwedge_{i \in I} \beta_i$ where the index set I is the set of all $1 \leq i \leq k$ such that the program counter ι_i corresponds to either a conditional branch or the entry point of an outermost nesting while loop, and the vector of affine expression $\alpha_\omega := \alpha_{k+1}$. Note that the PLP Ψ_ω is the condition that the execution of the loop body follows the execution path ω, and the affine expressions in the vector α_ω represents the values of the program variables after the execution of the loop body of W in terms of the initial values of the program variables and the fresh variables for the output of the outermost nesting while loops in W.

Our approach then constructs the LinTS for the original while loop W as follows:

- First, for each execution path ω of the loop body of W, we have a location ℓ_ω that represents this execution path.
- Second, for all locations $\ell_\omega, \ell_{\omega'}$ (that represent the execution paths ω, ω'), we have the transition $\tau_{\omega,\omega'} := (\ell_\omega, \ell_{\omega'}, \Psi_\omega \wedge \Psi'_{\omega'} \wedge \mathbf{x}' = \alpha_\omega)$ which means that if the execution path in the current iteration of the loop W is ω, then in the next iteration the execution path can be ω' with the guard condition $\Psi_\omega \wedge \Psi'_{\omega'} \wedge \mathbf{x}' = \alpha_\omega$ that comprises the conditions for the execution paths ω, ω' and the condition $\mathbf{x}' = \alpha_\omega$ for the next values of the program variables.

- Third, the initial condition θ in DNF is the disjunction of $\neg G$ and $G \wedge \Psi_\omega$, for all execution path ω. And we follow the standard technique (see e.g. Boutonnet and Halbwachs [9]) to include the input variables X_{in} and conjunct the linear assertion $\bigwedge_{x \in X} x = x_{\text{in}}$ into each disjunctive clause of the initial condition θ of the while loop W. Manually assumed initial conditions can also be conjuncted into each disjunctive clause of θ.

Again, we can remove invalid transitions by checking whether their guard condition is satisfiable or not.

Finally, we apply the approach [57] and our invariant propagation to the LinTS constructed above to obtain the loop summary as the invariant (over the variables in $X_{\text{in}} \cup X$) generated at the termination location ℓ_e, and rename each variable $x \in X$ to its output counterpart x_{out}. Recall that in the case of a non-crossing LinTS, our algorithm first applies the approach by Liu et al. [57] to obtain the invariants at the initial location, and then propagate the invariants to other locations; otherwise, our algorithm directly applies the approach [57].

5 IMPLEMENTATION AND EVALUATION

We implement our approach as a prototype tool that handles the linear disjunctive invariant generation over while loops in C programming language. The implementation includes a front-end that transforms C programs into the input form of our invariant generation solver (i.e., our back-end). The front-end, which is written in OCaml as well as C++ and based on Frama-C [33] program analysis platform, first transforms affine while loops in C into the canonical form as Figure 6a and then converts the canonical form into a linear transition system. The back-end is an extension of StInG [80] written in C++ and uses PPL 1.2 [8] for polyhedra manipulation (e.g., projection, generation computation, etc.). The back-end generates invariants at initial location by applying invariant-generation with Farkas' Lemma and uses invariant propagation method to generate invariants at other locations whenever applicable.

Notably, our back-end includes two additional features. The first one is the functionality to remove invalid transitions with unsatisfiable guard condition ρ. The second one is the treatment of the situation of the unsatisfiability in the application of Farkas' Lemma (see $-1 \geq 0$ at the bottom of Figure 3a and Figure 3b), which is however missing in the original tool StInG [80]. The former can simplify the LinTS to improve time efficiency and the later can increase accuracy. A key difficulty in the second one is that we obtain polyhedra rather than polyhedral cones, and thus cannot directly apply the generator computation. To address this difficulty, we show that it suffices to consider $\mu = 1$ in Figure 3b and include the generators of both the polytope and the polyhedral cone of the Minkowski decomposition of the polyhedron. As its correctness proof is somewhat technical, we relegate them to Appendix A and Appendix B.

Below we present the experimental evaluation. All the experimental results are obtained from a Linux (Ubuntu 20.04 LTS) with an 11th Intel Core i7 (3.20 GHz) CPU, 32 GB of memory.

We choose representative benchmarks related to disjunctive invariants and loop summary in the literature [5, 9, 43, 66, 75, 83] for evaluation. Our experimental results over these benchmarks are summarized in Table 1 – Table 3. In all the tables, "Our approach" means the experimental results by our approach, "Type" means what type of results we obtained, "Time" means the runtime measured in seconds, "v.s." means the accuracy of results compared against the original results in the literature. For the type of results, we have "Dis" means the result is an invariant (holding at the loop header) obtained by disjuncting all invariants at each location except ℓ_e, "Smry" means the result is a loop summary where the input variables carry the subscript 0 (e.g., x_0) and the output variables do not carry subscript (e.g., x), and "LR" means the result is an invariant at the termination location ℓ_e with a fixed input. "Detailed Results" means the detailed invariants or summaries for

Table 1. Experiment on Representative Benchmarks

Benchmark		Our Approach			
Name		Type	Time	v.s.	Detailed Result
Riley and Fedyukovich [66]	fig2 ★	Dis	0.02s	>	(z=0 ∧ 0 ≤ x ≤ 1000y-1 ∧ 1 ≤ y) ∨ (x-1000y=z ∧ x-999 ≤ 1000y ≤ x ∧ 1 ≤ 1000y) ∨ (z=1000 ∧ 1 ≤ y ∧ 1000y ≤ x-1000)
		Smry	0.03s	+	(x_0 ≤ x ∧ y=y_0 ∧ z=z_0) ∨ (x-1000y=z-z_0 ∧ y=y_0 ∧ x_0+1 ≤ 1000y ∧ 1000y ≤ x ≤ 1000y+999) ∨ (z=z_0+1000 ∧ y=y_0 ∧ x_0 ≤ 1000y-1 ∧ 1000y ≤ x-1000)
Ancourt et al. [5]	Gopan07 ★	Dis	<0.01s	+	(x=y ∧ 0 ≤ x ≤ 50) ∨ (x+y=102 ∧ 51 ≤ x ≤ 102)
		LR	<0.01s	>	x=102 ∧ y=-1
	Gulwani07 ★	Dis	0.01s	+	(y=50 ∧ 1 ≤ x ≤ 49) ∨ (x=y ∧ 50 ≤ x ≤ 99)
		LR	0.01s	>	x=y=100
	Halbwachs ★	Dis	0.01s	+	0 ≤ y ≤ x ≤ 100
		LR	0.01s	>	(101 ≤ x ≤ 102 ∧ 0 ≤ y ∧ y+2 ≤ x) ∨ (x=101 ∧ 1 ≤ y ≤ 101)
Sharma et al. [75]	POPL07 ★	Dis	<0.01s	>	(y=50 ∧ 0 ≤ x ≤ 49) ∨ (x=y ∧ 50 ≤ x ≤ 99)
		LR	<0.01s	>	x=y=100
	CAV06 ★	Dis	0.01s	+	(f=0 ∧ x=y ∧ 0 ≤ x ≤ 50) ∨ (f=0 ∧ x+y=102 ∧ 51 ≤ x ≤ 101) ∨ (f=0 ∧ y=0 ∧ x=102)
		LR	0.01s	+	f=1 ∧ x=102 ∧ y=-1
	ex1 ★	Dis	0.02s	+	(f=0 ∧ x=y ∧ 0 ≤ x ≤ 48) ∨ (f=0 ∧ x+y=98 ∧ 49 ≤ x ≤ 98) ∨ (f=0 ∧ y=-1 ∧ x=99)
		LR	0.02s	+	f=1 ∧ x=99 ∧ y=-2
	ex2 ★	Dis	0.02s	+	(0 ≤ x ≤ 24 ∧ x=y=z) ∨ (25 ≤ x ≤ 50 ∧ x=y ∧ 5x-100=z) ∨ (51 ≤ x ≤ 99 ∧ x+y=102 ∧ 5x-100=z)
		LR	0.02s	+	x=100 ∧ y=2 ∧ z=400
Xie et al. [83]	fig1a ‡★	Dis	0.01s	+	(n=100 ∧ 0 ≤ x ≤ 99 ∧ x=z-1) ∨ (n=100 ∧ 1 ≤ x ≤ 99 ∧ x=z)
		Smry	0.01s	>	(x=z=n=n_0 ∧ x_0 ≤ z_0-1 ∧ z_0 ≤ n-1) ∨ (x_0+1 ≤ x=n=n_0 ≤ z_0=z) ∨ (x=z=n=n_0 ∧ z_0 ≤ x_0 ≤ n-1)
	fig6a ★	Dis	0.02s	+	(n-1 ≥ m ∧ j ≥ 0 ∧ i ≥ 0 ∧ n-i ≥ 1 ∧ m-j ≥ 1) ∨ (m=j ∧ n ≥ i+1 ∧ i ≥ 0 ∧ n-1 ≥ m ∧ m ≥ 1)
		Smry	0.02s	=	i_0=j_0=0 ∧ m=m_0 ∧ i=n=n_0 ∧ j=0 ∧ 1 ≤ m ≤ n-1
	fig1c ★	Dis	<0.01s	+	1 ≤ j ≤ m-1 ∧ 0 ≤ k ∧ i ≤ m-1
		Smry	<0.01s	>	1 ≤ j ≤ m-1 ∧ m ≤ i ≤ 2m-2 ∧ 1 ≤ k ≤ m-i_0
	fig1f ★	Dis	0.01s	+	(s=1 ∧ x_1 = x_2 ∧ 0 ≤ x_1) ∨ (s=2 ∧ x_1 = x_2 + 1 ∧ 1 ≤ x_1) ∨ (s=3 ∧ x_1 = x_2 ∧ 1 ≤ x_1) ∨ (s=4 ∧ x_1 = x_2 ∧ 1 ≤ x_1)
		Smry	0.02s	>	(s=1 ∧ x_1-x_2 = x_{10}-x_{20} ∧ x_{10} ≤ x_1) ∨ (s=2 ∧ x_1-x_2-1 = x_{10}-x_{20} ∧ 1 ≤ x_1-x_{10}) ∨ (s=3 ∧ x_1-x_2 = x_{10}-x_{20} ∧ 1 ≤ x_1-x_{10}) ∨ (s=4 ∧ x_1-x_2 = x_{10}-x_{20} ∧ 1 ≤ x_1-x_{10})
Boutonnet and Halbwachs [9]	eudiv ‡★	Dis	0.01s	+	r ≥ b ≥ 1 ∧ a ≥ q+r ∧ q ≥ 0
		Smry	0.01s	>	a=a_0 ∧ b=b_0 ∧ r ≥ 0 ∧ b ≥ r+1 ∧ a+1 ≥ b+q+r ∧ q ≥ 1
	correct1 ‡★	Dis	<0.01s	+	s ≥ 0 ∧ t ≥ 0 ∧ x=o+e
		Smry	<0.01s	>	x-x_0=e-e_0 ∧ x_0=o+e_0 ∧ t ≥ 0 ∧ x-x_0-t+s+e_0 ≥ 1 ∧ x_0 ≥ x+s ∧ x_0+t ≥ e_0+x
	janne_complex ⊛	Dis	20.86s	+	(55x+11y ≤ 1686 ∧ x ≤ y ∧ 481x ≥ 241y) ∨ (y ≤ 5 ∧ 2x-y ≥ 14 ∧ x-y ≤ 12) ∨ (y ≤ 5 ∧ x-y ≤ 12 ∧ 65x-29y ≥ 420) ∨ (55x+11y-1686 ≤ 0 ∧ 1 ≤ x-y ≤ 12 ∧ y ≥ 6 ∧ 481x+4y ≥ 4842 ∧ 3x-y ≥ 22)
		Smry	34.34s	+	(x_0 ≤ 29 ∧ y_0 ≤ 5 ∧ y_0 ≤ x_0 - 1 ∧ x ≤ y + 12 ∧ -36x-12x_0+y-18y_0 ≥ -1811 ∧ -36x-61x_0+y-18y_0 ≥ -3036 ∧ -107x-52x_0-12y-78y_0+6639 ≥ 0 ∧ 3x-y ≥ 22 ∧ x ≥ 30 ∧ 2x-2x_0+y_0 ≥ 12 ∧ x-x_0 ≥ 4) ∨ (x_0 ≤ 29 ∧ y_0 ≥ x_0 ∧ 30 ≤ x ≤ 31 ∧ x ≤ y+12 ∧ 5x-5x_0+y-y_0 ≥ 0 ∧ 13x-13x_0-3y+3y_0 ≥ 0 ∧ 127x-155x_0-25y+25y_0+308 ≥ 0 ∧ 297x-297x_0-47y+47y_0-1064 ≥ 0)
	cnt_minver ⊛ ‡ ★	Dis	<0.01s	+	j ≤ 3i ≤ 2j ∧ j ≤ 3
		Smry	0.01s	+	(i_0 ≤ 2 ∧ j_0 ≤ 2 ∧ i=j=3) ∨ (i_0 ≤ 2 ∧ j_0 ≥ 3 ∧ i=3 ∧ j=j_0)
	cnt_fft1 ⊛★	Dis	0.10s	+	(n=8 ∧ m=15 ∧ k+2 ≤ j ∧ k ≤ 8 ∧ 9 ≤ j ≤ 2k ∧ 1 ≤ i ≤ 15) ∨ (n=8 ∧ m=15 ∧ 2 ≤ j ≤ 8 ∧ 2 ≤ i ≤ 15 ∧ j ≤ 2k ∧ k ≤ 8)
		Smry	0.10s	+	(n=n_0 ∧ m=m_0 ∧ i_0+1 ≤ i=m+1 ∧ j_0 ≥ n+1 ∧ k+1 ≤ j ≤ 2k ∧ 3k+1 ≤ j+n) ∨ (n=n_0 ∧ m=m_0 ∧ i=m+1 ≥ i_0+2 ∧ 2k ≥ j ≥ k+1 ∧ 3k+1 ≤ j+n ∧ j_0 ≤ n) ∨ (k=n=n_0 ∧ m=m_0 ∧ i=m+1 ≥ i_0+1 ∧ 2k ≥ j ∧ k ≥ j_0 ∧ k ≥ j)
Henry et al. [43]	fig1 ★	Dis	<0.01s	+	(2x=t ∧ p=0 ∧ 2x ≤ 99 ∧ 0 ≤ x) ∨ (2x=t+3 ∧ p=1 ∧ 2 ≤ x ≤ 51)

"Dis" or "Smry" or "LR" generated from our approach. For the accuracy comparison in the column "v.s.", we have "=" means that our result is equal to the original result, ">" means that our result is strictly stronger, and "+" means that no existing result is available. For the symbol in the column "Name", we have "⊛" means the affine nested loop, "‡" means that our result is strengthened by incremental method [10], "★" means that our result is obtained by invariant propagation.

First, Table 1 presents the experimental results for our approach with invariant propagation. Note that the runtime for all benchmarks is too less to be trivial and thus we only consider the comparison in accuracy. In Table 1, one can observe that our approach mostly generate invariants with better accuracy. In some multi-phase benchmarks, our approach could derive significantly tighter disjunctive invariants and summaries such as in Boutonnet and Halbwachs [9], Riley and Fedyukovich [66], Sharma et al. [75], Xie et al. [83], and generate precise LR results of program in disjunctive form such as in Ancourt et al. [5], Sharma et al. [75]. On benchmarks that require incremental invariant (or summary) generation [10], we run our approach twice for which the second run generates more invariants (or summaries) based on those obtained in the first run and obtain tighter invariants (or summaries) for benchmarks such as *fig1a* in Xie et al. [83] and *eudiv,*

correct1, cnt_minver in Boutonnet and Halbwachs [9]. Finally, our approach could also resolve nested loops with complex control flow such as *janne_complex, cnt_minver, cnt_fft1* in Boutonnet and Halbwachs [9].

Remark 3. *We are unable to have direct comparison with the very related work Boutonnet and Halbwachs [9], Henry et al. [43], Lin et al. [56], Riley and Fedyukovich [66], Xie et al. [83] due to the following reasons. First, the works Boutonnet and Halbwachs [9], Lin et al. [56], Xie et al. [83] neither publicize their implementation nor report the detailed invariants in some key benchmarks such as* janne_complex, cnt_minver, cnt_fft1. *Second, although the tool PAGAI [43] claims the functionality of disjunctive invariant generation, we find that this functionality could not work in the disjunctive-invariant-generation mode. Third, the tool by Riley and Fedyukovich [66] follows the smtlib format of the CHC solver, whereas the transformation of a program into the smtlib format is sophisticated and there is no existing tool that converts between the two complete different formats. In addition, their benchmarks are in smtlib format, so that we can not recover the loop information. We also note that a recent tool [69, 86] based on machine learning could only generate conjunctive invariants, and thus is orthogonal to our approach.*

Second, Table 2 presents the experimental comparison with the state-of-the-art software verification tool SeaHorn [73]. Since SeaHorn requires the user to provide a goal property, we feed SeaHorn non-trivial goal properties arising from the disjunctive feature of the benchmarks. In the table, the column "SeaHorn" means the results generated by SeaHorn, and the "Proof" column specifies whether the tool could verify the given assertion for which the symbol "F" (resp. "T") means the obtained results are unable (resp. able) to prove the goal property respectively. One can observe that SeaHorn fail on most of the benchmarks in disjunctive invariant generation even if these benchmarks are at a small scale. We find that the reasons include failure to handle **break**-statement such as *Gopan07*, incapability to handle non-initialized variables such as *fig1a, fig6a, fig1c, eudiv, correct1, janne_complex, cnt_minver, cnt_fft1* and incompetence to handle disjunction such as *Halbwachs*, etc. Unrolling of loops does not help as Seahorn fails on benchmarks that can be completely unrolled into finite iterations such as *fig2*, *CAV06*, *ex1*.

Table 2. Experiment for SeaHorn

Benchmark		SeaHorn	Our Approach
Name		Proof	Proof
Riley and Fedyukovich [66]	fig2★	F	T
Ancourt et al. [5]	Gopan07★, Halbwachs★	F	T
	Gulwani07★	T	T
Sharma et al. [75]	CAV06★, ex1★	F	T
	POPL07★, ex2★	T	T
Xie et al. [83]	fig1a‡★, fig1c★, fig6a★	F	T
	fig1f★	T	T
Boutonnet and Halbwachs [9]	eudiv‡★, correct1‡★, janne_complex⊛, cnt_minver⊛ ‡ ★, cnt_fft1⊛★	F	T
Henry et al. [43]	fig1★	T	T

Table 3. Experiment for Invariant Propagation

Benchmark				Our Approach		
				No PPG	PPG	
Name		Loc	Dim	Time (s)	Time (s)	Speedup
POPL07★ [75]	3p	3	9	<0.01	<0.01	1.00X
	4p	4	16	0.05	0.04	1.25X
	5p	5	25	0.33	0.05	6.60X
	6p	6	36	3.32	0.09	36.89X
	7p	7	49	35.40	0.21	168.57X
	8p	8	64	359.21	0.40	898.03X
	9p	9	81	2900.43	0.84	3452.89X

Finally, Table 3 demonstrates the improvement of speedup by our invariant propagation technique over a large benchmark. In Table 3, "r-p" means that r is a benchmark-inside number to show how many locations are there in the LinTS, "Loc" means the number of locations under LinTS, "Dim" means the number of unknown coefficients at all locations, "No PPG" means using our disjunctive linear invariant generation over each location under LinTS without invariant propagation (i.e., following the original approach in Liu et al. [57]), "PPG" means using our invariant propagation, "Time(s)" means the runtime measured in seconds, and "Speedup" means the ratio of time consumed by "No PPG" against "PPG". The experimental results in Table 3 show that our invariant propagation could substantially improve the time efficiency over a large non-crossing LinTS.

6 RELATED WORKS

Below we compare our approach with the most related existing approaches in the literature. We first have the comparison with the previous constraint-solving approaches for invariant generation. Note that our approach generates linear invariants via Farkas' Lemma, and hence is a constraint-solving approach for numerical invariant generation. The detailed comparison is as follows.

- On linear invariant generation, our approach follows the framework to apply Farkas' Lemma as proposed by Colón et al. [18], Liu et al. [57], Sankaranarayanan et al. [71], and extend the framework to disjunctive linear invariants and loop summary. Besides, we further propose techniques such as the invariant propagation and the integration of loop summary into nested loops so as to improve the time efficiency and the accuracy of the generated invariants. The recent result [47] also considers Farkas' Lemma, but focuses on conjunctive linear invariants over unnested affine while loops through the use of eigenvalues and roots of polynomial equations, and hence is orthogonal to our approach. Besides, other approaches on linear invariant generation include de Oliveira et al. [29], Gulwani et al. [39], Gupta and Rybalchenko [41]. The approach [39] solves the quadratic constraints from Farkas' Lemma by SAT solvers based on bit-vector modeling. The approach [29] uses eigenvectors to handle several restricted classes of conjunctive linear invariants. The tool InvGEN [41] generates conjunctive linear invariants by an integrated use of abstract interpretation and Farkas' Lemma. All these approaches proposes completely different approaches, and thus they are orthogonal to our approach.
- Since our approach targets linear invariant generation, it is incomparable with previous results on polynomial invariant generation [1, 15, 16, 20, 28, 44, 45, 49, 55, 68, 70, 85]. Note that, although linear functions are a special case of polynomials with degree 1, these approaches focus on how to handle general polynomials (i.e., the general case that the degree can be greater than 1) (through e.g. the Gröbner base) and hence are orthogonal to our approach. Moreover, most of these approaches consider only conjunctive polynomial invariants, and hence cannot handle disjunctive invariants.

It is also worth noting that the previous work [75] proposes a general framework for detecting multiphase disjunctive invariants that can be instantiated with constraint solving. Multiphase disjunctive invariants are a subclass of the PDA pattern in the sense that the multiphase feature is a special PDA in which a location cannot go back to itself once the location is left. Therefore, we consider a wider class of disjunctive invariants as compared with Sharma et al. [75].

Second, we compare our approach with previous approaches in abstract interpretation. Compared with the approaches that generate conjunctive linear invariants via polyhedral abstract domain [7, 24, 77], our approach targets the more general case of disjunctive linear invariants. There are also a bunch of abstract-interpretation approaches in disjunctive linear invariant generation, such as the work [38] that performs disjunctive partitioning by representing the contribution of each iteration with a separate abstract-domain element, the recent work [9] that distinguishes different disjunctive cases by different entries into the conditional branches w.r.t the input values, and the state-of-the-art tool PAGAI [43] that may infer disjunctive invariants as disjunctions of elements of the abstract domain via specific iteration algorithm. These approaches are based on abstract interpretation and heuristics different from the PDA pattern and Farkas' Lemma we follow, and hence are orthogonal to our approach.

Third, we compare our approach with existing approaches that follow other methodologies such as machine learning, logical inference and data-driven approaches. We first have a methodological comparison as follows. Unlike the method of constraint solving that can have an accuracy guarantee

for the generated invariants based on the established constraints, other methods such as machine-learning, logical-inference and data-driven approaches cannot give any accuracy guarantee on the generated numerical invariants, since they cannot guarantee the accuracy of the coefficients in numerical invariants. Moreover, machine learning and data-driven approaches themselves cannot guarantee that the generated logical assertions are indeed invariants, while logical inference approaches often require a goal property and manual efforts. Then, a key merit is that our approach can generate invariants *without* a given goal property to be proved, while several approaches (such as IC3 [78], CLN2INV [69], Riley and Fedyukovich [66]) and state-of-the-art program analysis platforms (such as CPAchecker [25], SeaHorn [73] and Ultimate Automizer [81]) requires a goal property and are only capable of generating invariants relevant to prove the property. Note that the invariant generation without a given goal property is a classical setting (see e.g. Colón et al. [18], Cousot and Halbwachs [24]), and has found applications in probabilistic program verification (see e.g. Chakarov and Sankaranarayanan [13], Wang et al. [82]) recently. Finally, it is worth noting that a heuristic approach also based on the PDA pattern is proposed in Lin et al. [56] recently, but this approach relies heavily on the heuristics of inductive variables (i.e., assignments must be in the form $x := x + c$ or $x := c * x$) together with the guessing of invariants, does not focus on nested loops, and requires a given goal property; in contrast, our approach can handle any affine assignment and the general PDA patterns with nested loops by Farkas' Lemma, and does not require a goal property.

Fourth, we compare our approach with the related approaches in loop summary. Compared with the approaches [22, 23] that are based on convex polyhedra abstract domain and can only generate conjunctive linear loop summaries, our approach is able to generate disjunctive loop summaries. Compared with the approach by Kranz and Simon [52] that applies Heyting completion [37] (to make an existing domain meet-distributive) on-demand and computes a summary of the function for each on-demand created predicate (represented via Herbrand terms), our approach is capable of generating linear inequality invariants with arbitrary coefficients, while their approach mainly uses an equality domain (as well as a pointer domain) to track equality relations of limited form between variables. Compared with the approach by Boutonnet and Halbwachs [9] that enhances abstract interpretation with disjunction from distinct entries into the conditional branches in the program by different initial inputs, our approach is orthogonal in the sense that we apply Farkas' Lemma and the PDA pattern by Xie et al. [83], which are completely different. Compared with (i) the PIPS tool [5, 46, 62] that employs heuristics to generate conjunctive linear loop summaries and (ii) the approach by Ancourt et al. [5] that generates conjunctive linear invariants by a simple heuristics that examines the stepwise incremental update of affine assignments, our approach follows a completely different methodology (Farkas' Lemma and the PDA pattern) and generates disjunctive linear loop summary. Compared with the approaches by Popeea and Chin [64, 65] that maintain a set of limited pre-fixed number of polyhedra for abstracting program states at each program point (to derive a disjunctive polyhedral analysis) under the framework of abstract interpretation, our approach is orthogonal to them and does not require the user to manually provide assertions. Compared with the approach [83] that proposes the PDA pattern, our approach solves the PDA pattern by Farkas' Lemma and invariant propagation so that both arbitrary affine assignments, general PDA patterns with nested loops and nondeterminism can be handled, while their approach relies on heuristics such as inductive variables ($x := x + c$ or $x := c * x$) and strict alternation between PDA states so that their approach can only handle affine assignments and PDA patterns in restricted forms and cannot handle nondeterminism and nested loops. Finally, our approach focuses on loop summary, and it would be an interesting future direction to investigate how our approach could be used for procedure summary [4, 40, 87].

7 CONCLUSION AND FUTURE WORK

In this work, we proposed a novel approach to generate linear disjunctive invariants and loop summaries via Farkas' Lemma, the disjunctive pattern of path dependency automata [83] and a novel invariant propagation technique. We implemented our approach as a prototype tool on the Frama-C platform. Experimental results show that our approach is capable of deriving substantially more accurate linear disjunctive invariants and summaries compared with existing approaches. There can be several future directions for this work. One further direction would be to derive more accurate linear transition systems by considering modulus information. Another future direction is to extend our approach to procedure summary. Finally, a more complete tool implementation is also an important future direction.

REFERENCES

[1] Assalé Adjé, Pierre-Loïc Garoche, and Victor Magron. 2015. Property-based Polynomial Invariant Generation Using Sums-of-Squares Optimization. In *SAS (LNCS, Vol. 9291)*. Springer, 235–251.

[2] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. 2012. Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In *CAV (LNCS, Vol. 7358)*. Springer, 672–678. https://doi.org/10.1007/978-3-642-31424-7_48

[3] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS (LNCS, Vol. 6337)*. Springer, 117–133. https://doi.org/10.1007/978-3-642-15769-1_8

[4] Frances E. Allen. 1974. Interprocedural Data Flow Analysis. In *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, Jack L. Rosenfeld (Ed.). North-Holland, 398–402.

[5] Corinne Ancourt, Fabien Coelho, and François Irigoin. 2010. A Modular Static Analysis Approach to Affine Loop Invariants Detection. *Electron. Notes Theor. Comput. Sci.* 267, 1 (2010), 3–16. https://doi.org/10.1016/j.entcs.2010.09.002

[6] Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. 2021. Polynomial reachability witnesses via Stellensätze. In *PLDI*. ACM, 772–787. https://doi.org/10.1145/3453483.3454076

[7] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. 2003. Precise Widening Operators for Convex Polyhedra. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 337–354. https://doi.org/10.1007/3-540-44898-5_19

[8] Roberto Bagnara, Elisa Ricci, Enea Zaffanella, and Patricia M. Hill. 2002. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In *SAS (Lecture Notes in Computer Science, Vol. 2477)*. Springer, 213–229. https://doi.org/10.1007/3-540-45789-5_17

[9] Rémy Boutonnet and Nicolas Halbwachs. 2019. Disjunctive Relational Abstract Interpretation for Interprocedural Program Analysis. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (LNCS, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 136–159. https://doi.org/10.1007/978-3-030-11245-5_7

[10] Aaron R. Bradley. 2012. Understanding IC3. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7317)*, Alessandro Cimatti and Roberto Sebastiani (Eds.). Springer, 1–14. https://doi.org/10.1007/978-3-642-31612-8_1

[11] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *CAV (LNCS, Vol. 3576)*. Springer, 491–504. https://doi.org/10.1007/11513988_48

[12] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[13] Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *CAV (LNCS, Vol. 8044)*. Springer, 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

[14] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2019. Non-polynomial Worst-Case Analysis of Recursive Programs. *ACM Trans. Program. Lang. Syst.* 41, 4 (2019), 20:1–20:52. https://doi.org/10.1145/3339984

[15] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *PLDI*. ACM, 672–687. https://doi.org/10.1145/3385412.3385969

[16] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *CAV (LNCS, Vol. 9206)*. Springer, 658–674. https://doi.org/10.1007/978-3-319-21690-4_44

[17] Yinghua Chen, Bican Xia, Lu Yang, Naijun Zhan, and Chaochen Zhou. 2007. Discovering Non-linear Ranking Functions by Solving Semi-algebraic Systems. In *ICTAC (LNCS, Vol. 4711)*. Springer, 34–49. https://doi.org/10.1007/978-3-540-75292-9_3

[18] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *CAV (LNCS, Vol. 2725)*. Springer, 420–432. https://doi.org/10.1007/978-3-540-45069-6_39

[19] Michael Colón and Henny Sipma. 2001. Synthesis of Linear Ranking Functions. In *TACAS (LNCS, Vol. 2031)*. Springer, 67–81. https://doi.org/10.1007/3-540-45319-9_6

[20] Patrick Cousot. 2005. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *VMCAI (LNCS, Vol. 3385)*. Springer, 1–24. https://doi.org/10.1007/978-3-540-30579-8_1

[21] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252. https://doi.org/10.1145/512950.512973

[22] Patrick Cousot and Radhia Cousot. 2001. Compositional Separate Modular Static Analysis of Programs by Abstract Interpretation. In *SSGRR*. 6–10.

[23] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (LNCS, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 159–178. https://doi.org/10.1007/3-540-45937-5_13

[24] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*. ACM Press, 84–96. https://doi.org/10.1145/512760.512770

[25] CPAchecker 2022. CPAchecker: The Configurable Software-Verification Platform. https://cpachecker.sosy-lab.org.

[26] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: dynamic symbolic execution for invariant inference. In *ICSE*. ACM, 281–290. https://doi.org/10.1145/1368088.1368127

[27] Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. 2016. Danger Invariants. In *FM (LNCS, Vol. 9995)*. 182–198. https://doi.org/10.1007/978-3-319-48989-6_12

[28] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2016. Polynomial Invariants by Linear Algebra. In *ATVA (LNCS, Vol. 9938)*. 479–494. https://doi.org/10.1007/978-3-319-46520-3_30

[29] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2017. Synthesizing Invariants by Solving Solvable Loops. In *ATVA (LNCS, Vol. 10482)*. Springer, 327–343. https://doi.org/10.1007/978-3-319-68167-2_22

[30] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *OOPSLA*. ACM, 443–456. https://doi.org/10.1145/2509136.2509511

[31] J. Farkas. 1894. A Fourier-féle mechanikai elv alkalmazásai (Hungarian). *Mathematikaiés Természettudományi Értesitö* 12 (1894), 457–472.

[32] Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *FMCAD*. IEEE, 57–64.

[33] Frama-C 2022. Frama-C Software Analyzers. https://frama-c.com/

[34] Ting Gan, Bican Xia, Bai Xue, Naijun Zhan, and Liyun Dai. 2020. Nonlinear Craig Interpolant Generation. In *CAV (LNCS, Vol. 12224)*. Springer, 415–438. https://doi.org/10.1007/978-3-030-53288-8_20

[35] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *CAV (LNCS, Vol. 8559)*. Springer, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

[36] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *POPL*. ACM, 499–512. https://doi.org/10.1145/2837614.2837664

[37] Roberto Giacobazzi and Francesca Scozzari. 1998. A Logical Model for Relational Abstract Domains. *ACM Trans. Program. Lang. Syst.* 20, 5 (1998), 1067–1109. https://doi.org/10.1145/293677.293680

[38] Denis Gopan and Thomas W. Reps. 2007. Guided Static Analysis. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings (LNCS, Vol. 4634)*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer, 349–365. https://doi.org/10.1007/978-3-540-74061-2_22

[39] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program analysis as constraint solving. In *PLDI*. ACM, 281–292. https://doi.org/10.1145/1375581.1375616

[40] Sumit Gulwani and Ashish Tiwari. 2007. Computing Procedure Summaries for Interprocedural Analysis. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, 253–267. https://doi.org/10.1007/978-3-540-71316-6_18

[41] Ashutosh Gupta and Andrey Rybalchenko. 2009. InvGen: An Efficient Invariant Generator. In *CAV (LNCS, Vol. 5643)*. Springer, 634–640. https://doi.org/10.1007/978-3-642-02658-4_48

[42] Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2020. Learning fast and precise numerical analysis. In *PLDI*. ACM, 1112–1127. https://doi.org/10.1145/3385412.3386016

[43] Julien Henry, David Monniaux, and Matthieu Moy. 2012. PAGAI: A Path Sensitive Static Analyser. *Electron. Notes Theor. Comput. Sci.* 289 (2012), 15–25. https://doi.org/10.1016/j.entcs.2012.11.003

[44] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. 2018. Polynomial Invariants for Affine Programs. In *LICS*. ACM, 530–539. https://doi.org/10.1145/3209108.3209142

[45] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2017. Automated Generation of Non-Linear Loop Invariants Utilizing Hypergeometric Sequences. In *ISSAC*. ACM, 221–228. https://doi.org/10.1145/3087604.3087623

[46] François Irigoin, Pierre Jouvelot, and Rémi Triolet. 1991. Semantical interprocedural parallelization: an overview of the PIPS project. In *Proceedings of the 5th international conference on Supercomputing, ICS 1991, Cologne, Germany, June 17-21, 1991*, Edward S. Davidson and Friedel Hossfeld (Eds.). ACM, 244–251. https://doi.org/10.1145/109025.109086

[47] Yucheng Ji, Hongfei Fu, Bin Fang, and Haibo Chen. 2022. Affine Loop Invariant Generation via Matrix Algebra. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 257–281. https://doi.org/10.1007/978-3-031-13185-1_13

[48] Hari Govind V. K., Sharon Shoham, and Arie Gurfinkel. 2022. Solving constrained Horn clauses modulo algebraic data types and recursive functions. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498722

[49] Deepak Kapur. 2005. Automatically Generating Loop Invariants Using Quantifier Elimination. In *Deduction and Applications (Dagstuhl Seminar Proceedings, Vol. 05431)*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2006/511

[50] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *PLDI*. ACM, 248–262. https://doi.org/10.1145/3062341.3062373

[51] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. 2018. Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.* 2, POPL (2018), 54:1–54:33. https://doi.org/10.1145/3158142

[52] Julian Kranz and Axel Simon. 2018. Modular Analysis of Executables Using On-Demand Heyting Completion. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 291–312. https://doi.org/10.1007/978-3-319-73721-8_14

[53] Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio. 2013. SMT-Based Array Invariant Generation. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7737)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, 169–188. https://doi.org/10.1007/978-3-642-35873-9_12

[54] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 788–801. https://doi.org/10.1145/3314221.3314634

[55] Wang Lin, Min Wu, Zhengfeng Yang, and Zhenbing Zeng. 2014. Proving total correctness and generating preconditions for loop programs via symbolic-numeric computation methods. *Frontiers Comput. Sci.* 8, 2 (2014), 192–202.

[56] Yingwen Lin, Yao Zhang, Sen Chen, Fu Song, Xiaofei Xie, Xiaohong Li, and Lintan Sun. 2021. Inferring Loop Invariants for Multi-Path Loops. In *International Symposium on Theoretical Aspects of Software Engineering, TASE 2021, Shanghai, China, August 25-27, 2021*. IEEE, 63–70. https://doi.org/10.1109/TASE52547.2021.00030

[57] Hongming Liu, Hongfei Fu, Zhiyong Yu, Jiaxin Song, and Guoqiang Li. 2022. Scalable Linear Invariant Generation with Farkas' Lemma. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 132 (oct 2022), 29 pages. https://doi.org/10.1145/3563295

[58] Zohar Manna and Amir Pnueli. 1995. *Temporal verification of reactive systems - safety*. Springer.

[59] Kenneth L. McMillan. 2008. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *TACAS (LNCS, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 413–427. https://doi.org/10.1007/978-3-540-78800-3_31

[60] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*. IEEE Computer Society, 683–693. https://doi.org/10.1109/ICSE.2012.6227149

[61] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI*. ACM, 614–630. https://doi.org/10.1145/2908080.2908118

[62] PIPS 2022. PIPS. https://pips4u.org

[63] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI (LNCS, Vol. 2937)*. Springer, 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

[64] Corneliu Popeea and Wei-Ngan Chin. 2006. Inferring Disjunctive Postconditions. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4435)*, Mitsu Okada and Ichiro Satoh (Eds.). Springer, 331–345. https://doi.org/10.1007/978-3-540-77505-8_26

[65] Corneliu Popeea and Wei-Ngan Chin. 2013. Dual analysis for proving safety and finding bugs. *Sci. Comput. Program.* 78, 4 (2013), 390–411. https://doi.org/10.1016/j.scico.2012.07.004

[66] Daniel Riley and Grigory Fedyukovich. 2022. Multi-Phase Invariant Synthesis. In *ESEC/FSE 2022*. To appear.

[67] Enric Rodríguez-Carbonell and Deepak Kapur. 2004. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In *SAS (LNCS, Vol. 3148)*. Springer, 280–295. https://doi.org/10.1007/978-3-540-27864-1_21

[68] Enric Rodríguez-Carbonell and Deepak Kapur. 2004. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *ISSAC*. ACM, 266–273. https://doi.org/10.1145/1005285.1005324

[69] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=HJlfuTEtvB

[70] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. 2004. Non-linear loop invariant generation using Gröbner bases. In *POPL*. ACM, 318–329. https://doi.org/10.1145/964001.964028

[71] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Constraint-Based Linear-Relations Analysis. In *SAS (LNCS, Vol. 3148)*. Springer, 53–68. https://doi.org/10.1007/978-3-540-27864-1_7

[72] Alexander Schrijver. 1999. *Theory of linear and integer programming*. Wiley.

[73] SeaHorn 2015. SeaHorn: A fully automated analysis framework for LLVM-based languages. http://seahorn.github.io.
[74] Rahul Sharma and Alex Aiken. 2016. From invariant checking to invariant inference using randomized search. *Formal Methods Syst. Des.* 48, 3 (2016), 235–256. https://doi.org/10.1007/s10703-016-0248-5
[75] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 703–719. https://doi.org/10.1007/978-3-642-22110-1_57
[76] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *ESOP (LNCS, Vol. 7792)*. Springer, 574–592. https://doi.org/10.1007/978-3-642-37036-6_31
[77] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59.
[78] Fabio Somenzi and Aaron R. Bradley. 2011. IC3: where monolithic and incremental meet. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, Per Bjesse and Anna Slobodová (Eds.). FMCAD Inc., 3–8. http://dl.acm.org/citation.cfm?id=2157657
[79] Saurabh Srivastava and Sumit Gulwani. 2009. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 223–234. https://doi.org/10.1145/1542476.1542501
[80] StInG 2006. StInG: Stanford Invariant Generator. http://theory.stanford.edu/~srirams/Software/sting.html.
[81] UltimateAutomizer 2021. UltimateAutomizer: A Software Model Checker. https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automizer.
[82] Jinyi Wang, Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2021. Quantitative analysis of assertion violations in probabilistic programs. In *PLDI*. ACM, 1171–1186. https://doi.org/10.1145/3453483.3454102
[83] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 61–72. https://doi.org/10.1145/2950290.2950340
[84] Rongchen Xu, Fei He, and Bow-Yaw Wang. 2020. Interval counterexamples for loop invariant learning. In *ESEC/FSE*. ACM, 111–122. https://doi.org/10.1145/3368089.3409752
[85] Lu Yang, Chaochen Zhou, Naijun Zhan, and Bican Xia. 2010. Recent advances in program verification through computer algebra. *Frontiers Comput. Sci. China* 4, 1 (2010), 1–16. https://doi.org/10.1007/s11704-009-0074-7
[86] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *PLDI*. ACM, 106–120. https://doi.org/10.1145/3385412.3385986
[87] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. Hybrid top-down and bottom-up interprocedural analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 249–258. https://doi.org/10.1145/2594291.2594328

A PROOF OF NO ACCURACY LOSS FOR μ=1

To prove that there is no accuracy loss while setting μ manually to 1, for convenience, we denote the consecution tabular with $-1 \geq 0$ as constraint consecution tabular and the consecution tabular without $-1 \geq 0$ as transition consecution tabular. Then we prove that constraint consecution tabular is equivalent whether $\mu = 1$ or $\mu = k, \forall k > 0$.

$$
\begin{array}{c|l}
k & c_{\ell,1}x_1 + \cdots + c_{\ell,n}x_n \quad + d_\ell \geq 0 \\
\lambda_0 & \quad 1 \geq 0 \\
\lambda_1 & a_{11}x_1 + \cdots + a_{1n}x_n + a'_{11}x'_1 + \cdots + a'_{1n}x'_n + b_1 \geq 0 \\
\vdots & \quad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \quad \vdots \\
\lambda_m & a_{m1}x_1 + \cdots + a_{mn}x_n + a'_{m1}x'_1 + \cdots + a'_{mn}x'_n + b_m \geq 0 \\
\hline
 & \quad -1 \geq 0
\end{array}
$$

(a) $\mu = k, \forall k > 0$

Fig. 9. Constraint consecution tabular

We scale the leftmost coefficient column k, λ_i's to be $1, \lambda'_i$'s by multiplying $\frac{1}{k}$, where $\lambda'_i = \frac{\lambda_i}{k}$. The coefficient of invariants after transformation is the same as the previous tabular.

$$
\begin{array}{c|l}
1 & c_{\ell,1}x_1 + \cdots + c_{\ell,n}x_n \quad + d_\ell \geq 0 \\
\lambda'_0 & \quad 1 \geq 0 \\
\lambda'_1 & a_{11}x_1 + \cdots + a_{1n}x_n + a'_{11}x'_1 + \cdots + a'_{1n}x'_n + b_1 \geq 0 \\
\vdots & \quad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \quad \vdots \\
\lambda'_m & a_{m1}x_1 + \cdots + a_{mn}x_n + a'_{m1}x'_1 + \cdots + a'_{mn}x'_n + b_m \geq 0 \\
\hline
 & \quad -\frac{1}{k} \geq 0
\end{array}
$$

Fig. 10. Transformed constraint consecution tabular

Consider all the constraint consecution tabular and choose the maximum k_{max} of their k's. Then, we scale λ_i's, $c_{l,i}$'s and d_ℓ by k_{max} and modify λ'_0 to be $\lambda''_0 = \lambda'_0 + \frac{k_{max}}{k} - 1$. Note that it's necessary to select the fixed k_{max} to scale $c_{l,i}$'s, so that we avoid affecting the solution in the transition consecution tabular as transition consecution tabular is always satisfied if we multiply c with fixed constant k_{max}.

$$
\begin{array}{c|l}
1 & k_{max} \cdot c_{\ell,1}x_1 + \cdots + k_{max} \cdot c_{\ell,n}x_n \quad + k_{max} \cdot d_\ell \geq 0 \\
\lambda''_0 & \quad 1 \geq 0 \\
\lambda'_1 & \quad a_{11}x_1 + \cdots + \quad a_{1n}x_n + a'_{11}x'_1 + \cdots + a'_{1n}x'_n + \quad b_1 \geq 0 \\
\vdots & \quad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \quad \vdots \\
\lambda'_m & \quad a_{m1}x_1 + \cdots + \quad a_{mn}x_n + a'_{m1}x'_1 + \cdots + a'_{mn}x'_n + \quad b_m \geq 0 \\
\hline
 & \quad -1 \geq 0
\end{array}
$$

Fig. 11. Equivalent constraint transformation tabular

Thus we prove there is no accuracy loss as we set $\mu = 1$ by means of coefficient scaling.

B PROOF OF CORRECTNESS OF SOLUTIONS TO INVARIANT SETS IN THE IMPLEMENTATION PART

In the implementation, we utilize decomposition theorem of polyhedra and decompose the solution set of invariant when $\mu = 1$ to be a polytope P and a polyhedral cone C. Similarly, we denote F as the solution set of invariants, which contains the coefficient of invariants at any locations and F' as the solution set of invariants when $\mu = 1$ in all the consecution tabular. Then the union of the polytope and polyhedral cone is chosen as our solution set of invariants $F^* = P \cup C$, where $F' = P + C$.

Lemma 1. ***Decomposition theorem of polyhedra.*** *A set P of vectors in Euclidean space is a polyhedron if and only if $P = Q + C$ for some polytope Q and some polyhedral cone C.*

Now, we are going to prove the correctness of F^*. I.e., the vectors in polytope and polyhedral cone are both the coefficient of invariants in different locations.

Consider the relation between F and F', we define that $F'' = \{k \cdot \boldsymbol{c} \mid \boldsymbol{c} \in F', k > 0\}$.

Proposition 1. $F = F''$

Proof. We first consider $F \subseteq F''$, which equivalent to prove $\forall \boldsymbol{c} \in F, \exists \boldsymbol{c_0} \in F', k \in R$ such that $k \cdot \boldsymbol{c_0} = \boldsymbol{c}$. We consider the transition consecution tabular. Note that, the consecution tabular is always satisfied if we scale the invariant $\eta(\ell)$ and $\eta(\ell')$ simultaneously.

Then consider the constraint consecution tabular. We have proved if we multiply $\boldsymbol{c_0} \in F'$ with k_{max}, we can find corresponding $\boldsymbol{c} = k_{max} * \boldsymbol{c_0}$ is the solution to constraint consecution tabular with $\mu = k, \forall k > 0$. (the definition of k_{max} and proof is given in Appendix A)

Thus, we prove that $\forall \boldsymbol{c} \in F$, there exists $\boldsymbol{c_0} \in F'$ and $k_{max} \in R$ such that $k_{max} * \boldsymbol{c_0} = \boldsymbol{c}$ and have $F \subseteq F''$.

Secondly, we prove $F'' \subseteq F$. From the definition of F'', if $\boldsymbol{c} \in F'$, we multiply $\frac{1}{k}$ to $\mu = 1$ in the constraint consecution tabular, and $k \cdot \boldsymbol{c}$ satisfy the transformed tabulars and other transition consecution tabulars. So $k \cdot \boldsymbol{c} \in F$, and we have $F'' \subseteq F$.

So $F = F''$. □

We utilize the decomposition theorem in F' and have $F' = P + C$, where P is a polytope and C is a polyhedral cone. From the properties of polytope and polyhedral cone, a polytope is a convex hull of finitely many vectors and a polyhedral cone is finitely generated by some vectors.

$$P = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \ldots, \boldsymbol{p_n}\} \tag{1}$$

$$C = \{\boldsymbol{g_1}, \boldsymbol{g_2}, \ldots, \boldsymbol{g_m}\} \tag{2}$$

Note that the addition in the theorem means Minkowski sum, Thus,

$$F' = P + C = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \ldots, \boldsymbol{p_n}; \boldsymbol{g_1}, \boldsymbol{g_2}, \ldots, \boldsymbol{g_m}\} \tag{3}$$

Where $\boldsymbol{p_i}$'s represents the vectors finitely generate the polytope P and g_i's represents the vectors finitely generate the polyhedral cone C. That means that $\forall v \in F', v = a_1\boldsymbol{p_1} + \cdots + a_n\boldsymbol{p_n} + b_1\boldsymbol{g_1} + \cdots + b_m\boldsymbol{g_m}$,where $\sum_i a_i = 1$ (from the definition of convex hull) and $a_i, b_i \geq 0, \forall i$. Consider $F = F'' = \{k \cdot \boldsymbol{c} \mid \exists k > 0, k \cdot \boldsymbol{c} \in F, \boldsymbol{c} \in F'\}$, it's concluded that $\forall \boldsymbol{v} \in F, \boldsymbol{v} = a'_1\boldsymbol{p_1} + \cdots + a'_n\boldsymbol{p_n} + b'_1\boldsymbol{g_1} + \cdots + b'_m\boldsymbol{g_m}$, where $a'_i = ka_i, k > 0$ and $b'_i = kb_i, k > 0$.

Thus, it's obvious that $\forall \boldsymbol{p} \in P$, we have $\boldsymbol{p} \in F$ as we can set $\boldsymbol{g_i} = 0, \forall i$. However, we can not use the similar method to prove $\forall \boldsymbol{c} \in C, \boldsymbol{c} \in F$, as the $\sum_i a_i$ equal to a non-zero number and $a_i \geq 0, \forall i$. So to prove the $P \cup C$ is also the solution set of invariants, we should consider the practical implications of invariant.

We have known that $F = \{\boldsymbol{v} \mid \boldsymbol{v} = a'_1\boldsymbol{p_1}+\cdots+a'_n\boldsymbol{p_n}+b'_1+_1+\cdots+b'_m\boldsymbol{g_m}, a'_i \geq 0, \boldsymbol{g_i} \geq 0 \forall i, \sum_i a'_i > 0\}$ corresponding to the solution of $\boldsymbol{v}^T x <= d_\ell$.

Destruct F to be

$$F = \{p + c \mid p = a'_1\boldsymbol{p_1} + \cdots + a'_n\boldsymbol{p_n}, c = \boldsymbol{g_1} + \cdots + b'_m\boldsymbol{g_m}, a'_i \geq 0, \boldsymbol{g_i} \geq 0 \forall i, \sum_i a'_i > 0\} \tag{4}$$

From the above conclusion, $P \subseteq F$, which means $\forall \boldsymbol{p} \in P$, $\boldsymbol{p}^T\boldsymbol{x} <= d_\ell$ is satisfied. Also, $\forall \boldsymbol{v} \in F, \boldsymbol{v} = \boldsymbol{p} + \boldsymbol{c}, \boldsymbol{p} \in P, \boldsymbol{c} \in C$, and $(\boldsymbol{p} + \boldsymbol{c})^T\boldsymbol{x} <= d_\ell$.

Consider $\forall \varepsilon > 0$, we have $(\varepsilon\boldsymbol{p} + \boldsymbol{c})^T\boldsymbol{x} <= d_\ell$. Thus, we finally conclude that $\lim_{\varepsilon\to 0}(\varepsilon\boldsymbol{p} + \boldsymbol{c})^T\boldsymbol{x} = \boldsymbol{c}^T\boldsymbol{x} <= d_\ell$, which means for all $\boldsymbol{c} \in C$, $\boldsymbol{c}$ is also a solution to invariants. Thus we prove $C \in F$, and $P \cup C \in F$.

So it's correct to directly use the union of the polytope and the polyhedral cone to represents the solution set of invariants.

C PROOF FOR CORRECTNESS AND ACCURACY FOR OUR INVARIANT PROPAGATION

Below we prove the theoretical properties that the linear assertions generated from our invariant propagation are indeed invariants, and are at least as tight as the invariants generated from the previous approaches [57, 71].

Proposition 2. *The linear assertions generated by the invariant propagation are invariants.*

Proof. Let Γ be a LinTS whose directed graph $DG(\Gamma)$ has a non-crossing DFS tree T. The proof is by induction on the BFS level of the tree T. The base step is that the linear assertion at the root (i.e., the initial location) is correct since it is generated by the approach [57]. The inductive step is to show that if the linear assertions generated at the nodes of the current level are invariants, then so are the linear assertions at the next level. The proof for the inductive step follows from the fact that any path of the LinTS Γ that visits a location ℓ' in the next BFS level should first visit some location ℓ (with the valuation σ guaranteed to satisfy the invariant $\eta(\ell)$) in the current BFS level, and then possibly repeatedly stays at the location ℓ'. (Note that here we use the fact that there is no crossing edge in the DFS tree T. This fact is captured by the initial condition $K_{\tau,i}$ for a transition (ℓ, ℓ', ρ) (that is obtained from the ith disjunctive clause Φ_i of the invariant $\eta(\ell)$) and the invariant $I(\tau, \ell', i)$ for the self-loop LinTS $\Gamma[\ell', K_{\tau,i}]$ in a single propagation step. □

Proposition 3. *The invariant propagation generates invariants at least as tight as the previous approaches [57, 71].*

Proof. The proof proceeds via an induction on the BFS level of the invariant propagation. For the base step, we have that the linear invariant generated at the root is generated directly from the previous approach [57]. Then the base step follows from the fact that the approach [57] has the same precision as the original approach [71]. For the inductive step, suppose the induction hypothesis that the invariant of every node at the current BFS level in the DFS tree implies the counterpart generated by the approach [71]. We prove that the implication holds for the next BFS level. The proof can be obtained by observing that each individual linear inequality (as a conjunctive inequality in a linear assertion) in the invariants generated by the approach [71] on a location ℓ' at the next BFS level satisfies the consecution condition derived from any transition $\tau = (\ell, \ell', \rho)$ to the location ℓ', so that each such inequality is implied by the initial condition $K_{\tau,i}$ and satisfies the possible consecution condition from the self-loop in $\Gamma[\ell', K_{\tau,i}]$. Since we apply the same approach [71] (i.e., solving the same constraints for the unknown coefficients from the consecution condition of the self-loop), the invariant $\eta(\ell')$ generated by our invariant propagation implies any individual linear inequality generated by the approach [71]. □