Affine Disjunctive Invariant Generation with Farkas' Lemma

Hongming Liu, Jingyu Ke, Hongfei Fu, Liqian Chen, Guoqiang Li

Affine 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
GUOQIANG LI, Shanghai Jiao Tong University, China

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

In this work, we propose a novel approach to generate affine disjunctive invariants via Farkas' Lemma. By observing that disjunctive invariants often arise from the conditional branches in the loop body, our approach first pushes all the conditional branches in a loop body to the top level so that every top-level branch appears at the entry point of the loop body, and treats each top-level branch as a standalone branch location. Then our approach constructs an affine transition system that describes the transitions between the branch locations, and solves the conjunctive affine invariants at each branch by existing approaches via Farkas' Lemma. The final disjunctive invariant is the disjunction of the conjunctive invariants generated over these branch locations. Furthermore, we explore the following improvements to our approach: (a) an invariant propagation technique on the affine transition system that first generates an invariant only at the initial branch location and then propagates this invariant to other branch locations in a breadth-first fashion to improve the time efficiency of invariant generation; (b) an extension of our approach to generate affine disjunctive loop summary, and (c) the use of loop summary to generate affine disjunctive invariants over nested affine while loops. Experimental results over a wide range of benchmarks demonstrate that our approach can (i) generate affine disjunctive invariants that are capable of proving safety properties of while loops beyond previous approaches and state-of-the-art software verifiers with even a considerable advantage in time efficiency, and (ii) derive substantially more accurate affine loop summaries than existing approaches. Moreover, the improvement of invariant propagation can indeed speed up the invariant generation, and that of using loop summary to handle nested loops can indeed generate tight disjunctive invariants for nested loops.

1 INTRODUCTION

Invariant generation is the classical problem that targets the automated generation of invariants which can be used to aid the verification of critical program properties. An invariant at a program location is an 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 assertion. Since invariants provide an over-approximation for reachable program states, they play a fundamental role in program verification and can be used for safety [Albarghouthi et al. 2012; Manna and Pnueli 1995; Padon et al. 2016], reachability [Alias et al. 2010; Asadi et al. 2021; Bradley et al. 2005;

*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; Guoqiang Li, Shanghai Jiao Tong University, Shanghai, China, li.g@sjtu.edu.cn.

Chen et al. 2007; Colón and Sipma 2001; David et al. 2016; Podelski and Rybalchenko 2004] and time-complexity [Chatterjee et al. 2019] 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 different types of program objects, invariant generation targets numerical values (e.g., integers or real numbers) [Bagnara et al. 2003; Boutonnet and Halbwachs 2019; Chatterjee et al. 2020; Colón et al. 2003; Rodríguez-Carbonell and Kapur 2004a; Singh et al. 2017], arrays [Larraz et al. 2013; Srivastava and Gulwani 2009], pointers [Calcagno et al. 2011; Le et al. 2019], algebraic data types [K. et al. 2022], etc. By the different methodologies in existing approaches, invariant generation can be solved by abstract interpretation [Boutonnet and Halbwachs 2019; Cousot and Cousot 1977; Cousot and Halbwachs 1978; Gopan and Reps 2007], constraint solving [Chatterjee et al. 2020; Colón et al. 2003; Cousot 2005; Gulwani et al. 2008], inference [Calcagno et al. 2011; Dillig et al. 2013; Donaldson et al. 2011; Gan et al. 2020; Garg et al. 2014; McMillan 2008; Sharma and Aiken 2016; Somenzi and Bradley 2011; Xu et al. 2020], recurrence analysis [Farzan and Kincaid 2015; Kincaid et al. 2017, 2018], machine learning [Garg et al. 2016; He et al. 2020; Ryan et al. 2020; Yao et al. 2020], data-driven approaches [Chen et al. 2015; Csallner et al. 2008; Le et al. 2019; Nguyen et al. 2012; Riley and Fedyukovich 2022; Sharma et al. 2013], etc. Most results in the literature consider a strengthened version of invariants, called *inductive invariants*, that requires the inductive condition that the invariant at a program location is preserved upon every 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 program execution goes back to the location).

An important criterion on the quality of invariants is the accuracy against the exact set of reachable program states. Invariants that have too much accuracy loss (i.e., including too many program states that actually are not reachable) may be not precise enough to verify a target program property, while invariants with better accuracy can verify more program properties. Thus, ensuring the accuracy of the generated invariants is an important subject 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 assertions. Compared with conjunctive invariants, disjunctive invariants are capable of capturing disjunctive features such as multiple phases and mode transitions in while loops, and thus can be substantially more accurate than conjunctive ones.

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.

We focus on affine disjunctive invariants over affine while loops. An affine while loop is a while loop in which every conditional branch and loop guard is specified by affine inequalities, and every assignment statement is in the form of an affine expression that specifies an affine update on the current program state. Moreover, we consider the method of constraint solving that usually leads to accurate invariants. A typical constraint-solving method is via Farkas' Lemma [Colón et al. 2003; Ji et al. 2022; Liu et al. 2022; Sankaranarayanan et al. 2004b]. that provides a complete characterization for affine inequalities. However, as Farkas' Lemma only concerns conjunction of affine inequalities, its application is limited to conjunctive affine invariant generation. The question on how to apply Farkas' Lemma to disjunctive affine invariant generation remains to be a challenge.

To address this challenge, we explore a succinct disjunctive pattern from the conditional branches in an unnested loop, show how this disjunctive pattern can be integrated with Farkas' Lemma, and further explore algorithmic improvements and extensions to loop summary and nested loop. Our detailed contributions are as follows.

- First, for an unnested while loop, we consider disjunctive invariants that arise from the conditional branches in the loop body. To utilize the disjunctive information from the conditional branches, our approach first pushes all the conditional branches in a loop body to the top level so that every top-level branch appears at the entry point of the loop body. The motivation behind the top-level branches is that each top-level branch refers to a branch location with a standalone conjunctive invariant for the location, and the final invariant is an overall disjunction of the conjunctive invariants at the top-level branches. Taking one step further, our approach constructs an affine transition system that describes the transitions between the current and the next branch locations in a loop iteration, and solves the conjunctive affine invariants over at the branch locations of the affine transition system by existing approaches via Farkas' Lemma.
- Second, under a mild non-crossing assumption for an affine transition system, we improve the constraint solving algorithm by a novel invariant propagation technique that first generates the invariant only at the initial branch location and then obtains the invariants at other branch locations by a breadth-first propagation from the initial branch location. The invariant propagation technique improves the overall time efficiency by having the involved invariant computation only at the initial branch location and obtaining the invariants at other locations by a lightweight propagation process.
- Third, we extend our approach to generate affine disjunctive loop summary of affine while loops. Loop summary is the classical problem of the automated derivation of the input-output relationship for a while loop. In our extension, we follow the standard paradigm (see e.g. Boutonnet and Halbwachs [2019]) that incorporates fresh variables to represent the initial values of the program variables in the loop and generate the invariants for both the original program variables and the fresh variables through our approach to derive the loop summary.
- Fourth, to derive the disjunctive affine invariants and loop summaries of a nested loop, we extend our approach to nested while loops by integrating the loop summaries of the inner loops into the construction of the affine transition system for the outer loop.
- Finally, we implement our approach as a prototype tool built upon the Clang Static Analyzer [Clang Static Analyzer 2022].

Experimental results over a wide range of benchmarks (including SV-COMP and WCET) demonstrate that our approach can prove safety properties that are related to various disjunctive features in affine loops and beyond previous approaches and state-of-the-art software verifiers, and derive substantially more accurate affine disjunctive loop summary than previous approaches. Moreover, our approach is more time efficient compared with previous approaches.

2 PRELIMINARIES

Below we recall the model of affine transition systems [Sankaranarayanan et al. 2004b] and affine invariant generation over such model via Farkas' Lemma. In our invariant generation algorithm, we use affine transition systems as the abstract model for programs with affine conditions and updates. We first present the necessary definitions for affine transition systems and invariants, and then the application of Farkas' Lemma to affine invariant generation. We relegate a detailed example to the next section.

2.1 Affine Transition Systems and Invariants

To present affine transition systems, we first define several basic concepts related to affine inequalities as follows. An *affine inequality* (resp. *affine 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. An *affine assertion* over V is a conjunction of affine inequalities and equalities over V. Moreover, a *propositional affine predicate* (PAP) over V is a propositional formula whose atomic propositions are affine inequalities and equalities over V. A PAP is in disjunctive (resp. conjunctive) normal form (DNF) (resp. CNF) if it is a finite disjunction of affine assertions (resp. a finite conjunction of finite disjunctions of affine assertions), respectively. Note that we only consider non-strict no-smaller-than operator $\geq$ for affine inequalities, and the no-greater-than inequalities $\alpha \leq \beta$ could be equivalently transformed into $-\alpha \geq -\beta$. Moreover, although an affine equality $\alpha = \beta$ could be equivalently expressed by the conjunction of the two inequalities $\alpha \leq \beta$ and $\alpha \geq \beta$, we handle each affine equality directly since one can apply algorithmic optimizations to affine equalities. Below we present the definition of affine transition systems.

Definition 1 (Affine Transition Systems [Sankaranarayanan et al. 2004b]). *An* affine transition system *(ATS) is a tuple* $\Gamma = \langle X, X', L, \mathcal{T}, \ell^*, \theta \rangle$ *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.*
- *$\mathcal{T}$ 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 PAP over $X \cup X'$.*
- *θ is a PAP in DNF over the variables X. Informally, each disjunctive clause of the PAP θ specifies an independent initial condition at the initial location ℓ^*.*

We define the directed graph $DG(\Gamma)$ of an ATS 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 an ATS, 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 an ATS and simply abbreviate "valuation over X" as "valuation" (i.e., omitting X). Given an ATS, 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 an affine 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 an ATS Γ, two valuations σ, σ' (over X) and an affine 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 affine 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 $\sigma \models \psi$.

The semantics of an ATS Γ is specified by the notion of paths. A *path* π of the ATS 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 ATS. Informally, an ATS 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 ATS 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 ATS is an affine 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 an affine 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 affine 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).

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

To automatically generate invariants, one often investigates a strengthened notion called *inductive invariants*. In this work, we present inductive affine invariants in the form of inductive affine assertion maps [Colón et al. 2003; Liu et al. 2022; Sankaranarayanan et al. 2004b] as follows. We say that an *affine assertion map* (AAM) over an ATS is a function η that maps every location ℓ of the ATS to an affine assertion $\eta(\ell)$ over the variables X. Then an AAM 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 affine assertion obtained by replacing every variable $x \in X$ in $\eta(\ell')$ with its next-value counterpart $x' \in X'$.

Informally, an AAM 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 an ATS, one could verify that every affine assertion in an inductive AAM is indeed an invariant. In the rest of the work, we focus on the automated synthesis of inductive AAMs, and the disjunctive affine invariants are obtained by taking a disjunction of relevant affine assertions in an AAM.

Sometimes we need to consider the ATS $\Gamma[\ell, K]$ derived from an ATS Γ, a location ℓ of Γ and a subset K of valuations. In detail, the ATS $\Gamma[\ell, K]$ 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 an ATS that describes the execution of a while loop, we denote by $X_{\text{in}} := \{x_{\text{in}} \mid x \in X\}$ a copy of input variables from X and $X_{\text{out}} := \{x_{\text{out}} \mid x \in X\}$ a copy of output variables. We write $\mathbf{x}_{\text{in}}$ (resp. $\mathbf{x}_{\text{out}}$) for the vector of input (resp. output) variables, respectively. With the designated termination location ℓ_e at the end of a while loop, a loop summary S is a logical formula $S(\mathbf{x}_{\text{in}}, \mathbf{x}_{\text{out}})$ with free variables $\mathbf{x}_{\text{in}}, \mathbf{x}_{\text{out}}$ 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 Affine Invariant Generation

Farkas' Lemma [Farkas 1894] is a classical theorem in the theory of affine inequalities and previous results [Colón et al. 2003; Liu et al. 2022; Sankaranarayanan et al. 2004b] have applied the theorem to affine invariant generation. In this work, we consider a variant form of Farkas' Lemma [Schrijver 1999, Corollary 7.1h] as follows.

Theorem 2.1 (Farkas' Lemma). *Consider an affine assertion φ over a set $V = \{x_1, \dots, x_n\}$ of real-valued variables as in Figure 1a. When φ is satisfiable (i.e., there is a valuation over V that satisfies φ), it implies an affine inequality ψ as in Figure 1b (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) $d = \lambda_0 + \sum_{i=1}^{m} \lambda_i \cdot b_i$ as in Figure 1c. Moreover, φ is unsatisfiable if and only if the inequality $-1 \ge 0$ (as ψ) can be derived from above.*

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

(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

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

(c) The Tabular Form for Farkas' Lemma

Fig. 1. The φ, ψ and Tabular Form for Farkas' Lemma [Colón et al. 2003; Sankaranarayanan et al. 2004b]

One direction of Farkas' Lemma is straightforward, as one easily sees that if we have a non-negative affine 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 affine inequalities.

Remark 1. *In the statement of Farkas' Lemma above, if we strengthen an affine 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 affine inequalities into equalities with the relaxation of non-negativity for their corresponding λ_j's.*

The application of Farkas' Lemma can be visualized by the tabular form in Figure 1c (taken from Colón et al. [2003]), where $\bowtie_1, \dots, \bowtie_m \in \{=, \ge\}$ and we multiply $\lambda_0, \lambda_1, \dots, \lambda_m$ with their inequalities in φ and sum up them together to get ψ. For $1 \le j \le m$, if $\bowtie_j$ is $\ge$, we require $\lambda_j \ge 0$, otherwise (i.e., $\bowtie_j$ is $=$) we do not impose restriction on λ_j.

To illustrate the application of Farkas' Lemma to invariant generation, we also recall several concepts from polyhedra theory. A subset P of $\mathbb{R}^n$ is a *polyhedron* if $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}\}$ for some real matrix $A \in \mathbb{R}^{m \times n}$ and real vector $\mathbf{b} \in \mathbb{R}^m$, where $\mathbf{x}$ is treated as a column vector and the comparison $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$ is defined in the coordinate-wise fashion. A polyhedron P is a *polyhedral cone* if $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A} \cdot \mathbf{x} \leq \mathbf{0}\}$ for some real matrix $A \in \mathbb{R}^{m \times n}$, where $\mathbf{0}$ is the m-dimensional zero column vector. It is well-known from the Farkas-Minkowski-Weyl Theorem [Schrijver 1999, Corollary 7.1a] that any polyhedral cone P can be represented as $P = \{\sum_{i=1}^{k} \lambda_i \cdot \mathbf{g}_i \mid \lambda_i \geq 0 \text{ for all } 1 \leq i \leq k\}$ for some real vectors $\mathbf{g}_1, \ldots, \mathbf{g}_k$, where such vectors $\mathbf{g}_i$'s are called a collection of *generators* for the polyhedral cone P.

The existing approaches [Colón et al. 2003; Liu et al. 2022; Sankaranarayanan et al. 2004b] apply Farkas' Lemma to (conjunctive) affine invariant generation. All these approaches follow the template-based paradigm as follows:

- Establish an affine template with unknown coefficients over the input ATS that represents the inductive AAM to be solved.
- Apply the initiation and consecution conditions to the template to obtain the constraints for an AAM.
- 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 AAM for the input ATS.

The technical details of the paradigm above are given as **Step A1** – **Step A4** below. We fix an input ATS with variable set $X = \{x_1, \ldots, x_n\}$.

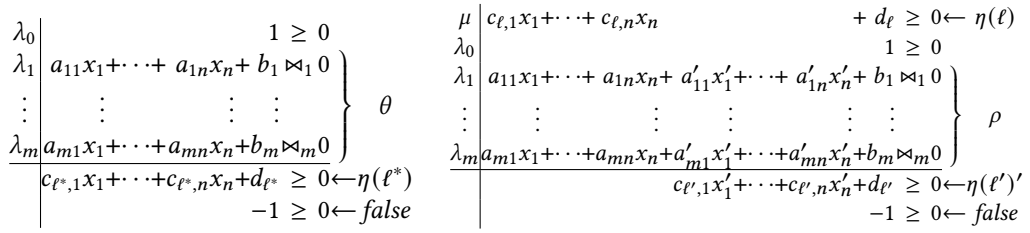
Step A1. In the first step, all the existing approaches establish a template for an inductive AAM. A template η involves an affine inequality $\eta(\ell) = c_{\ell,1} \cdot x_1 + \cdots + c_{\ell,n} \cdot x_n + d \geq 0$ at each location ℓ of the ATS with the unknown coefficients $c_{\ell,1}, \ldots, c_{\ell,n}, d$ to be resolved. Note that, although there is only one template at each location, the approaches can obtain a conjunctive affine invariant where one solution of the unknown coefficients corresponds to one conjunctive affine 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 affine 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 affine 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 1c to obtain Figure 2a which results in an affine 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 2b 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 affine since the fresh variable μ is multiplied to $\eta(\ell)$ in the tabular.

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

- By Colón et al. [2003], the non-affine constraints were solved through the complete but costly method of quantifier elimination;



(a) Initialization Tabular (b) Consecution Tabular

Fig. 2. Tabular for Initialization and Consecution [Colón et al. 2003; Sankaranarayanan et al. 2004b]

- By Sankaranarayanan et al. [2004b], the non-affine constraints were solved through (i) several reasonable heuristics to guess possible values for the key parameter μ in Figure 2b so as to remove the non-linearity and obtain an affine under-approximation of the original non-affine constraints, and (ii) the generator computation over polyhedral cones to obtain the invariants. A major heuristic there is to guess possible values for μ through some practical rules such 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. [2022], a substantial improvement on the scalability to the approach by Sankaranarayanan et al. [2004b] is proposed by generating affine 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.

3 OVERVIEW OF OUR APPROACH

In this section, we describe our our approach via a simple while loop with disjunctive feature. We first take a look at the example below.

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

(a) An affine while loop

```
x = 0;
y = 50;
while (x < 100){
  switch{
    case x > 49:
      x = x + 1;
      y = y + 1;
    case x ≤ 49:
      x = x + 1;
  }
}
```

(b) The transformed loop

Fig. 3. An affine while loop from Sharma et al. [2011] and its transformed loop

Example 1. *Consider an affine while loop in Figure 3a taken from Sharma et al. [2011] with integer-valued variables x, y. Before the loop, the values of the variables x, y are initialized to 0, 50, respectively. In each loop iteration, the value of x is incremented by one, and if this value exceeds 50, then the value of y is incremented by one. The loop has a disjunctive feature from the if branch in the loop body that*

when the value of x is no greater than 50*, then only the variable x is incremented; when the value of x reaches* 51*, then both the variables x, y are incremented (until the value of x reaches* 100*).* □

Our approach has three main steps. *The first step* to handle the loop is to transform the loop into an equivalent canonical form where all the conditional branches appear at the entry point (i.e., top level) of the loop body. Each top-level branch corresponds to a standalone conjunctive invariant at the branch, and the overall invariant is meant to be a disjunction of the conjunctive invariants at these top-level branches.

Example 2. *We push the if branch* ***if*** $(x > 50)$ *in Figure 3a to the front of the loop body, so that we have two top-level branches $x > 49$ and $x \leq 49$ in Figure 3b. Notice that if the value of x is no greater than* 49 *at the entry point of the original loop body, then the if branch is not executed and only the variable x is incremented. Otherwise (i.e., the value of x is greater than* 49*), both the variables x, y are incremented in the loop iteration. Thus, in Figure 3b we have a special* ***switch*** *statement at the top of the loop body, and two top-level branches $x > 49$ and $x \leq 49$ to distinguish between the cases of the increment of only x or both x, y. Our aim is to obtain independent invariants for the cases of $x > 49$ and $x \leq 49$, say η_1 for $x > 49$ and η_2 for $x \leq 49$, and the final invariant is the disjunction of $\eta_1 \wedge x > 49$ and $\eta_2 \wedge x \leq 49$ (i.e., $(\eta_1 \wedge x > 49) \vee (\eta_2 \wedge x \leq 49)$).* □

To obtain the invariants for the top-level branches, *our second step* is to construct an affine transition system that includes every top-level branch as a standalone location (we shall refer to such a location as a *branch location*) and every possible jump between the top-level branches (for the current and next loop iteration) as a transition. The details are given as follows.

Example 3. *To obtain an affine transition system that describes the jumps between the top-level branches in Figure 3b, we treat the two branches $x > 49$ and $x \leq 49$ as standalone branch locations, and use ℓ_1 and ℓ_2 to denote the branch locations, respectively. Moreover, we have a special location ℓ_e that represents the termination location of the loop. We further add transitions between the branch locations that respect their entry conditions in the current and next loop iteration. For example, the transition $\tau_5 : \langle \ell_2, \ell_1, \rho_5 \rangle$ specifies the jump from the branch location ℓ_2 to the branch location ℓ_1 with the guard condition ρ_5 specified by $x \leq 49,\ 50 \leq x' \leq 99,\ x' = x + 1,\ y' = y$, where (i) $x \leq 49$ and $50 \leq x' \leq 99$ are derived from the entry condition $x \leq 49$ of the branch location ℓ_2 (in the current loop iteration) and the counterpart $x > 49$ of ℓ_1 (in the next loop iteration), both conjuncted with the loop guard, and (ii) $x' = x + 1, y' = y$ specify the update to the variables x, y under the current branch location ℓ_2. In the same way, one can derive transitions for other jumps. The whole affine transition system is given in Figure 4, where ℓ_2 is the initial location. Note that the transitions τ_2, τ_6 are infeasible and hence can be removed.* □

After the affine transition system is constructed, *our third step* applies existing approaches [Liu et al. 2022; Sankaranarayanan et al. 2004b] via Farkas' Lemma to obtain the invariants at each branch location, and group these invariants disjunctively together to obtain the final invariant. Below we give a detailed illustration.

Example 4. *Consider to generate affine invariants over the ATS in Figure 4a. The approach [Sankaranarayanan et al. 2004b] 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** in the previous section). Then, it generates the constraints from the initialization and consecution conditions (**Step A2**) and simplifies the constraints by the Farkas' tabular in Figure 2 (**Step A3**). For initialization, the tabular in Figure 5a 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 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 Farkas' tabular to the transition τ_5 as in Figure 5b. The*

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

(a) The ATS

Fig. 4. The corresponding ATS for Figure 3b

fresh variables λ_j's are projected away and the fresh non-affine variable μ is eliminated by reasonable heuristics that guesses its value through either practical rules such as factorization or setting μ manually to 0 or 1. Other transitions are treated in a similar way.

λ_0			1	≥ 0	
λ_1	x			$= 0$	$\}$ θ
λ_2		y	-50	$= 0$	$\}$
	$c_{\ell_2,1}x + c_{\ell_2,2}y +$		d_{ℓ_2}	$\ge 0 \leftarrow \eta(\ell_2)$	

μ	$c_{\ell_2,1}x + c_{\ell_2,2}y$					$+ d_{\ell_2}$	$\ge 0 \leftarrow \eta(\ell_2)$	
λ_0						1	≥ 0	
λ_1	$-x$					$+ 49$	≥ 0	$\}$
λ_2				x'		$- 50$	≥ 0	$\}$
λ_3				$-x'$		$+ 99$	≥ 0	$\}$ ρ_5
λ_4	$-x$	$+$		x'		$- 1$	$= 0$	$\}$
λ_5		$-y$	$+$		y'		$= 0$	$\}$
				$c_{\ell_1,1}x' + c_{\ell_1,2}y' +$		d_{ℓ_1}	$\ge 0 \leftarrow \eta(\ell_1)'$	
						-1	$\ge 0 \leftarrow$ *false*	

(a) Initialization Farkas' Tabular for Figure 4a (b) Consecution Farkas' Tabular for τ_5 in Figure 4a

Fig. 5. Initialization and Consecution Farkas' Tabulars for Figure 4a

*The constraints obtained from the previous step constitutes a PAP Φ in CNF where each clause in the conjunction is the constraint derived from either the initialization or the consecution of a transition, and every disjunctive affine assertion in such a clause in the conjunction results from a distinct guessed value for the non-affine μ parameter in the tabular for consecution. In the last step (**Step A4**), the approach [Sankaranarayanan et al. 2004b] expands the PAP Φ equivalently into a DNF PAP (where each clause in the disjunction is an affine assertion that defines a polyhedral cone) and obtains the affine invariants by the generator computation of each polyhedral cone in the DNF PAP. For this example, one clause in the disjunction (treated as a polyhedral cone) from the DNF formula is shown in Figure 6a (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 6a, we obtain the corresponding generators and their invariants in Figure 6b, where in the left part each row specifies a generator with a type (a point, a ray 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.*

The affine invariants obtained from the generator computation can be further minimized by removing trivial invariants such as $0 \ge 0$ and redundant inequalities. After processing all the disjunctive clauses of the DNF and grouping all the generated invariants together, the final disjunctive invariant at the entry point of the loop body is $\eta(\ell_1) \vee \eta(\ell_2)$, which is $(x = y \wedge 50 \le x \le 99) \vee (y = 50 \wedge 0 \le x \le 49)$, and the invariant for the termination location ℓ_e is derived from the invariants at ℓ_1, ℓ_2 and the transitions to ℓ as $\eta(\ell_e) = (x = y = 100)$. The approach [Liu et al. 2022] improves the scalability by generating the invariants one location at a time that allows to detect redundant invariants more efficiently. □

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

(a) A 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 6a

Fig. 6. Example of a disjunctive clause and its generators and invariants

In the third step, we observe that simply to apply existing approaches in Farkas' Lemma incurs extra computation overhead since all these approaches require to compute the invariant at every location by an involved computation procedure. The overhead may be large if the number of branch locations is considerable. Thus, we propose an *invariant propagation* technique that only computes the invariant at the initial branch location (during which our approach does not compute the invariant at other branch locations) and have a breadth-first propagation of this invariant to other branch locations, when the affine transition system admits a depth-first search tree without cross edges (i.e., edges that go back to a non-ancestor node). The following example illustrates this idea over the ATS in Figure 4a.

Example 5. *Consider the affine transition system in Figure 4a. Its underlying directed graph is given in Figure 7 (here we ignore the termination location ℓ_e), for which we have a DFS tree that is composed of the solid tree edges and the dashed edges are back edges (i.e., edges that go back to an ancestor node). Notice that there is no cross edge in the DFS tree. Our invariant propagation technique utilizes the absence of cross edges.*

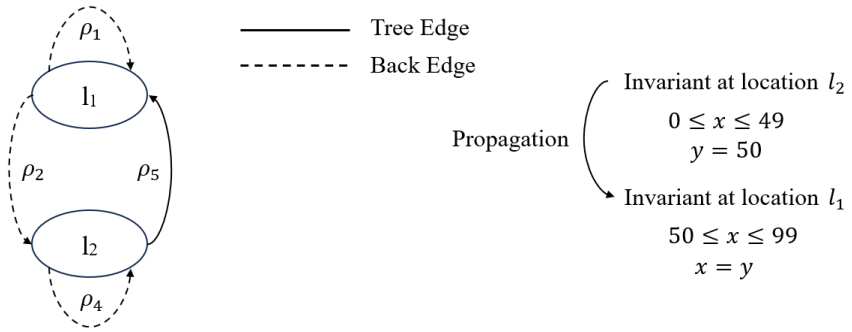


Fig. 7. Invariant Propagation through DFS tree

First, our approach computes the invariant at the initial branch location ℓ_2 to be $\eta(\ell_2) = (y = 50 \wedge 0 \leq x \leq 49)$. Then, our invariant propagation calculates a new initial condition $x = 50 \wedge y = 50$ for the branch location ℓ_1 by propagating the invariant $\eta(\ell_2)$ along the transition ρ_5. Next, with the new initial condition, our approach solves the invariant at the branch location ℓ_1 by considering only the self-loop transition ρ_1 to obtain $\eta(\ell_1) = (x = y \wedge 50 \leq x \leq 99)$. Note that the back edge from ℓ_1 to ℓ_2 does not affect the invariant at ℓ_1 (even if ρ_2 was feasible). Finally, the invariant at the termination

location ℓ_e is derived from $\eta(\ell_1)$ and $\eta(\ell_2)$. The key point of invariant propagation over this example is that the invariant at the branch location ℓ_1 is obtained by a propagation from that of ℓ_2, rather than a thorough invariant solving over the whole affine transition system. □

Loop Summaries and Nested Loops. Besides invariant generation over unnested affine while loops, our approach also generates affine disjunctive loop summary by introducing fresh input variables that represent the inputs of the programs and generating invariants over both the original program variables and the fresh input variables. Furthermore, our approach tackles nested affine while loops by computing the loop summaries of the inner loops and uses these summaries as the abstraction of the inner loops to construct the affine transition system for the outer loop.

4 DISJUNCTIVE AFFINE INVARIANT GENERATION FOR UNNESTED LOOPS

In this section, we present our approach for generating affine 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 ATS to be derived from the loop. We consider the canonical form of an unnested affine while loop as in Figure 8, where we have:

- The PAP 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 affine 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 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 major motivation behind Figure 8 is that we treat each top-level branch ϕ_i as a standalone branch location and the overall invariant is a disjunction of the invariants at these branch locations.

```
while G {
  switch
    case φ1 : x := F1(x) ; δ1 ;
      ⋮
    case φm : x := Fm(x) ; δm ;
}
```

Fig. 8. The canonical form of an unnested affine while loop

Any unnested affine while loop with break statement can be transformed into the canonical form in Figure 8 by recursively examining the substructures of the loop body of the loop. A detailed transformation is provided in Appendix A. 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.

Below we illustrate our algorithm to generate disjunctive affine invariants on unnested affine while loops. Informally, our algorithm applies the top-level branches and follows Farkas' Lemma for affine invariant generation as in Colón et al. [2003]; Liu et al. [2022]; Sankaranarayanan et al. [2004b], and further proposes the improvement of invariant generation that is closely related to the top-level branches and has not been considered in the existing approaches [Colón et al. 2003; Liu et al. 2022; Sankaranarayanan et al. 2004b]. Here we first consider 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 8 as stated previously. Taking Example 1 as a running example, the canonical form of the example is given in Example 2.

Step B2. Then we apply the top-level branches to transform the loop C_W into a ATS. The transformation is in a straightforward fashion that every top-level conditional branch (i.e., ϕ_i in Figure 8) corresponds to a stand-alone location, and the guard of a transition is determined by the loop condition (i.e., G) as well as the branch conditions of the source and target locations of the transition. Formally, we have that the ATS 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 location with branch condition ϕ_i and ℓ_e is the termination program counter of the loop.
- For each $1 \leq i \leq m$, if $\delta_i = \textbf{break}$, we have that transition (where we denote $\mathbf{x}' := (x'_1, \ldots, x'_n)^{\mathrm{T}}$)
$$\tau_i = (\ell_i, \ell_e, G \wedge \phi_i \wedge \mathbf{x}' = \mathbf{F}_i(\mathbf{x}))$$
that specifies the one-step jump from the branch location ℓ_i to the termination location ℓ_e, where 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) and $\mathbf{x}' = \mathbf{F}_i(\mathbf{x})$ (for the affine update).
- For each $1 \leq i, j \leq m$, where $\delta_i \neq \textbf{break}$, we have the transition
$$\tau_{ij} = (\ell_i, \ell_j, G \wedge \phi_i \wedge G[\mathbf{x}'/\mathbf{x}] \wedge \phi_j[\mathbf{x}'/\mathbf{x}] \wedge \mathbf{x}' = \mathbf{F}_i(\mathbf{x}))$$
that specifies the one-step jump from the branch location ℓ_i to the branch location ℓ_j, for which the guard condition is $G \wedge \phi_i \wedge G[\mathbf{x}'/\mathbf{x}] \wedge \phi_j[\mathbf{x}'/\mathbf{x}] \wedge \mathbf{x}' = \mathbf{F}_i(\mathbf{x})$ 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 loop guard $G[\mathbf{x}'/\mathbf{x}]$ and the branch condition ϕ_j upon entering the location ℓ_j.
- For each $1 \leq i \leq m$, where $\delta_i \neq \textbf{break}$, we have the transition
$$\tau'_i = (\ell_i, \ell_e, G \wedge \phi_i \wedge (\neg G)[\mathbf{x}'/\mathbf{x}] \wedge \mathbf{x}' = \mathbf{F}_i(\mathbf{x}))$$
for the one-step jump from the branch location ℓ_i to the termination location ℓ_e for which the guard condition is a conjunction of the loop guard G, the branch condition ϕ_i, the affine update $\mathbf{x}' = \mathbf{F}_i(\mathbf{x})$ 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 ATS. The transformation for the running example has been given in Example 3.

Step B3. After the transformation into an ATS, we follow existing approaches [Colón et al. 2003; Liu et al. 2022; Sankaranarayanan et al. 2004b] that generate affine invariants with Farkas' Lemma. In particular, we apply the recent approach [Liu et al. 2022] that has the most scalability (see

Example 4 for the running example). 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 from non-termination locations to the termination location. In the following, we further propose an invariant propagation technique that takes advantage of a common feature in the top-level branches to improve the time efficiency.

In our invariant propagation, we explore a special structure in the derived ATS that often arises in the top-level branches, 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 affine transition systems.

Definition 2. *An ATS Γ is* non-crossing *if there exists a depth-first search (DFS) tree of the directed graph $DG(\Gamma)$ (i.e., its underlying directed graph) rooted at the initial location that 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).*

An example of a non-crossing DFS tree is given in Example 5, while a simple example that violates the non-crossing property would be a complete directed graph. Non-crossing ATS's are common in the top-level branch form of an unnested while loop. For example, the case of multiphase invariants [Sharma et al. 2011] is a special case of non-crossing affine transition systems where a location is never entered again once it is left. The strict alternation between branch locations is also a special case of non-crossing affine transition systems. In general, any affine transition system that has one outgoing-transition for every location (which arises from deterministic mode change in while loops) is non-crossing, since in its DFS tree there is no cross edges.

We illustrate the main workflow of our invariant propagation technique. Consider an ATS 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 affine invariant $\eta(\ell^*)$ at the location ℓ^* generated from the approach by Liu et al. [2022], 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 that is from a location ℓ in the current BFS front with the invariant $\eta(\ell)$ (as a DNF PAP) 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 PAP as an invariant $I(\tau, \ell')$ for the ATS $\Gamma[\ell', K_\tau := \{\sigma' \mid \exists\sigma.(\sigma \models \eta(\ell) \wedge \sigma, \sigma' \models \rho)\}]$ (see Page 5 for the definition of $\Gamma[-,-]$) via the approach by Liu et al. [2022] and disjuncts all these $I(\tau, \ell')$'s together to obtain $\eta(\ell')$. Note that in such an ATS $\Gamma[\ell', K_\tau := \{\sigma' \mid \exists\sigma.(\sigma \models \eta(\ell) \wedge \sigma, \sigma' \models \rho)\}]$ we consider self-loop transitions at a location ℓ' since our approach needs to cover the case that when propagated to the location ℓ', the ATS (and the original program) may dwell at the branch location ℓ' for a finite unbounded number of steps. 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 PAP invariant $\eta(\ell) = \bigvee_{i=1}^{d} \Phi_i$ where each Φ_i is an affine 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 affine invariant of the ATS $\Gamma[\ell', K_{\tau,i} := \{\sigma' \mid \exists\sigma.(\sigma \models \Phi_i \wedge \sigma, \sigma' \models \rho)\}]$. Hence, our approach calculates $I(\tau, \ell')$ by computing for each $1 \leq i \leq d$ the conjunctive affine invariant $I(\tau, \ell', i)$ (over $\Gamma[\ell', K_{\tau,i}]$) by the approaches [Liu et al. 2022].

Example 6. *A preliminary example of invariant propagation for our running example has been given in Example 5, where we have the DFS tree and the breath-first propagation from the branch*

location ℓ_2 to ℓ_1. We give more details for the single propagation step from ℓ_2 to ℓ_1. For $\Phi := \eta(\ell_2) = (y = 50 \wedge 0 \leq x \leq 49)$ and transition $\tau = (\ell_2, \ell_1, \rho_5)$, our approach computes $K_\tau := \{\sigma' \mid \exists \sigma.(\sigma \models \Phi \wedge \sigma, \sigma' \models \rho_5) = \{(x, y) \mid (x = 50 \wedge y = 50)\}$, and further derives the invariant for ℓ_1 from the ATS $\Gamma[\ell_1, K_\tau]$ (that comprises only the location ℓ_1 and the self-loop transition ρ_1 at ℓ_1). □

To instantiate a single propagation step, we need to encode the set $K_{\tau,i}$ as an affine 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, polyhedral 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 an affine 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 8. 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 an affine 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 polyhedral projection is needed.
- The second case is that the matrix A is not invertible. Then we solve the system of affine equations $\mathbf{A}\mathbf{x} = \mathbf{x}' - \mathbf{b}$ by the standard method of Gaussian Elimination in elementary affine 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 $\mathbf{A}\mathbf{x} = \mathbf{x}' - \mathbf{b}$ and can be expressed as an affine combination of the entries in $\mathbf{x}'$ (i.e., $\mathbf{u}(\mathbf{x}') = \mathbf{C}\mathbf{x}' + \mathbf{d}$ 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 $\mathbf{A}\mathbf{x} = \mathbf{0}$ 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 affine 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 affine 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 $n - \mathrm{rank}(A)$ where $\mathrm{rank}(A)$ 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 [Sankaranarayanan et al. 2004b] or separately [Liu et al. 2022] 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 ATS 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 affine transition systems do not cover all cases of directed acyclic graphs, but this can be partially remedied by first computing the strongly-connected components (SCCs) of the underlying ATS and then considering each SCC separately.

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

- First, our algorithm transforms an unnested affine while loop into the canonical form in Figure 8 and further transforms it into an affine transition system.
- Second, our algorithm applies the approach by Liu et al. [2022] and our invariant propagation technique (if possible) to obtain affine invariants at the branch locations of the affine transition system. In the case that the affine transition system is non-crossing w.r.t the initial location, our algorithm applies the approach by Liu et al. [2022] to obtain the affine invariant at the initial location and afterwards derive the invariants at other locations through invariant propagation. Otherwise (i.e., the affine transition system is not non-crossing), our algorithm follows the original approach by Liu et al. [2022] to generate the invariants at all the locations.

By an induction on the depth of the DFS tree, we can prove that the assertions generated from our invariant propagation are indeed invariants and are at least as tight as the invariants generated by the previous approaches [Liu et al. 2022; Sankaranarayanan et al. 2004b]. Due to space limitation, we relegate the detailed proofs to Appendix D.

5 DISJUNCTIVE AFFINE INVARIANT GENERATION FOR NESTED LOOPS

Recall that in the previous section, we proposed a novel approach for generating disjunctive affine invariants over unnested while loops via Farkas' Lemma, top-level branches 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 inner while loop W' in W (from the innermost to the outermost), and then tackling the main loop body via the top-level branches and the loop summaries $S_{W'}$ of the inner loops. 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 main loop W into its corresponding ATS by the top-level branches. Unlike the situation of unnested while loops, a direct recursive algorithm that transforms the loop W into a canonical form in Figure 8 as in the unnested case is not possible, since one needs to tackle the loop summaries from the inner 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 of the jumps emitting from assignment statements and conditional branches, for a program counter that represents the entry point of an inner while loop that is not nested in other inner loops, we have the special treatment that the jump at the program counter is directed to the termination program counter of this inner loop in the loop body of W (i.e., skipping the execution of this inner loop). An *execution path* in the CFG H is a directed path of program counters 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 7. *Consider the janne_complex program from Boutonnet and Halbwachs [2019] in Figure 9. The CFG of the program is given in Figure 10 where the nodes correspond to the program counters, the directed edges with guards specifies the jumps and their conditions, and the affine assignments are given in the program counters A_1, A_2, A_3.*

We denote by W the outer loop with entry point E_{Outer}, and by W' the inner loop with entry point E_{inner}. 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*

```
while (x < 30 ){
    while (y < x ){
        if  (y > 5)  y = y * 3;
        else  y = y + 2;
        if  (y>=10  &&  y<=12)  x = x + 10;
        else  x = x + 1;
    }
    x = x + 2;  y = y - 10;
}
```

Fig. 9. The janne_complex program

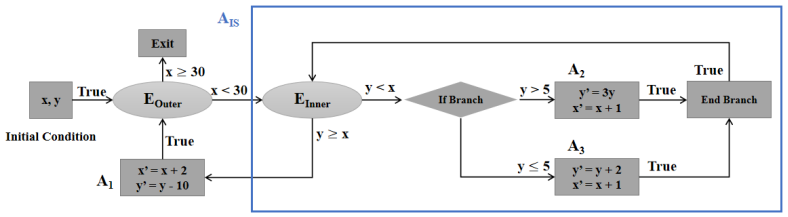


Fig. 10. The CFG of *janne_complex* [Boutonnet and Halbwachs 2019]

less than 30*, etc.), and terminates in the* Exit *statement. The only execution path for the loop body of* W *is* $A_{IS} \rightarrow A_1$*, for which we abstract the whole inner loop by* A_{IS}*.* □

Based on the CFG H and the execution paths, our approach constructs the ATS for the outer loop W as follows. Since the output of an inner while loop W' in W cannot be exactly determined from the input to the loop W', we first have fresh output 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 inner loop W'. These output variables are used to express the loop summaries of these inner loops.

Then, to get the numerical information from execution paths, we 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 PAPs β_i (for $x \in X$ and $1 \leq i \leq k$) over the program variables in X (for which they represent their initial values at the start of the loop body of W here) and the fresh output variables. 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 PAP 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 $\beta_1 = \mathbf{true}$ 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 PAP 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 PAP 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 inner 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 $\beta_{i+1} := S_{W'}(\alpha_i, \overline{x}_{W'})$. Here we use the ouput variables to express the loop summary. Note that the loop summary $S_{W'}$ (see Page 6 for the definition of S) is recursively computed.

Example 8. *Continue with the execution path in Example 7. The evolution of α_i and β_i with the*

$$\alpha_1 = [x, y], \beta_1 = \textbf{true} \quad \xrightarrow{x < 30} \quad \alpha_2 = [x, y], \beta_2 = \beta_1 \wedge x < 30 \quad \xrightarrow{A_{IS}}$$
$$\alpha_3 = [\overline{x}_{W'}, \overline{y}_{W'}], \beta_3 = \beta_2 \wedge S_{W'}(\alpha_2, \alpha_3) \quad \xrightarrow{A_1} \quad \alpha_4 = [\overline{x}_{W'} + 2, \overline{y}_{W'} - 10], \beta_4 = \beta_3$$

Fig. 11. The evolution of α_i and β_i for the execution path of W in Figure 10

initial setting $\alpha_1 = [x, y], \beta_1 = \textbf{true}$ is given in Figure 11. □

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 PAP $\Psi_\omega := \bigwedge_{i \in I} \beta_i$ 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 inner while loop, and the vector of affine expression $\alpha_\omega := \alpha_{k+1}$. Note that the PAP 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 represent the values of the program variables after the execution path ω 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 inner while loops in W.

Finally, our approach constructs the ATS for the loop W and we only present the main points:

- First, for each execution path ω of the loop body of W, we have a standalone location ℓ_ω for this execution path. Recall that we abstract the inner loops, so that the execution paths can be finitely enumerated.
- Second, for all locations $\ell_\omega, \ell_{\omega'}$ (from 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, we enumerate all possible initial locations l_ω, along with their corresponding initial conditions $\theta = G \wedge \Psi_\omega$. To derive loop summary, we follow the standard technique (see e.g. Boutonnet and Halbwachs [2019]) to include the input variables X_{in} and conjunct the affine assertion $\bigwedge_{x \in X} x = x_{\text{in}}$ into each disjunctive clause of the initial condition θ. Manually specified initial conditions can also be conjuncted into θ.

A detailed process that handles **break** statement is similar to the unnested situation. Again, we can remove invalid transitions by checking whether their guard condition is satisfiable or not.

Finally, we apply the approach [Liu et al. 2022] and our invariant propagation to the ATS constructed above to obtain the loop summary as an 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 x_{out}.

Example 9. *Consider the janne_complex program in Figure 9. By integrating the loop summary of the inner loop, our approach constructs an affine transition system that corresponds to a while loop of 22 top-level branches. For the lack of space, we relegate the detailed branches to Figure 16 in Appendix F.* □

At the end of the illustration of our algorithms, we discuss possible extensions as follows.

Remark 2 (Extensions). *Our approach can be extended in the following ways. To obtain a more precise top-level branch representation, one extension is by (i) distinguishing even/odd integer values of program variables and (ii) detecting hidden termination phases via the approach in [Ben-Amram and Genaim 2017]. To handle machine integers, another extension is by having a piecewise disjunctive treatment for the cases of overflow and non-overflow. Finally, our approach could be extended to floating point numbers by considering piecewise affine approximations [Miné 2004, 2006].*

6 IMPLEMENTATION AND EVALUATION

We implement our approach as a prototype tool based on the Clang Static Analyzer [Clang Static Analyzer 2022]. 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 first transforms affine while loops in C into the canonical form as in Figure 8 and then converts the canonical form into an affine transition system. The back-end is an extension of StInG [StInG 2006] written in C++ and uses PPL 1.2 [Bagnara et al. 2002] for polyhedra manipulation (e.g., projection, generator 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 2a and Figure 2b), which is however missing in the original tool StInG [StInG 2006]. The former can simplify the ATS 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 2b 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 B and Appendix C.

Below we present the experimental evaluation. We compare our approach with (i) previous related approaches on disjunctive invariant generation, (ii) relevant approaches in loop summary and (iii) state-of-the-art software verifiers including SeaHorn [SeaHorn 2015], CPAChecker [CPAchecker 2022], Veriabs [Darke et al. 2021] (the champion of the reachability track in SV-COMP 2023) and the recent recurrence analysis tool from Wang and Lin [2023]. All the experimental results are obtained from a Linux (Ubuntu 20.04 LTS) with an 11th Gen Intel Core i7 (3.20 GHz) CPU, 32 GB of memory. We choose representative benchmarks related to affine disjunctive invariants and loop summary from the literature [Ancourt et al. 2010; Boutonnet and Halbwachs 2019; Henry et al. 2012; Riley and Fedyukovich 2022; Sharma et al. 2011; Xie et al. 2016] and also SV-COMP, WCET benchmark sets for evaluation. Our experimental results are summarized in Table 1 – Table 4.

In all the tables, "Our approach" means the results by our approach, "Type" means what type of results we obtained, "Time" means the runtime measured in seconds, "v.s." means the accuracy compared against the previous results. 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

Table 1. Experimental Results on Invariant Generation

Benchmark		Our Approach			
Name		Type	Time	v.s.	Detailed Result
Riley and Fedyukovich [2022]	fig2 ★	Dis	0.02s	>	(z=0 ∧ 0 ≤ x ≤ 1000y-1 ∧ 1 ≤ y) ∨ (x-1000y=z ∧ x-999 ≤ 1000y ≤ x ∧ 1 ≤ y) ∨ (z=1000 ∧ 1 ≤ y ∧ 1000y ≤ x-1000)
Ancourt et al. [2010]	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. [2011]	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. [2016]	fig1a ★	Dis	0.01s	+	(n=100 ∧ 0 ≤ x ≤ 99 ∧ x=z-1) ∨ (n=100 ∧ 1 ≤ x ≤ 99 ∧ x=z)
	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)
	fig1c ★	Dis	<0.01s	+	1 ≤ j ≤ m-1 ∧ 0 ≤ k ∧ i ≤ m-1
	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)
Boutonnet and Halbwachs [2019]	eudiv ‡★	Dis	0.01s	+	r ≥ b ≥ 1 ∧ a ≥ q+r ∧ q ≥ 0
	correct1 ★	Dis	<0.01s	+	s ≥ 0 ∧ t ≥ 0 ∧ x=o+e
	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)
	minver ⊛ ‡ ★	Dis	<0.01s	+	j ≤ 3i ≤ 2j ∧ j ≤ 3
	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)
Henry et al. [2012]	fig1 ★	Dis	<0.01s	+	(2x=t ∧ p=0 ∧ 2x ≤ 99 ∧ 0 ≤ x) ∨ (2x=t+3 ∧ p=1 ∧ 2 ≤ x ≤ 51)

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 determined fixed-input. "Detailed Results" means detailed invariants or summaries for "Dis" or "Smry" or "LR" generated from our approach. For the accuracy 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 an affine nested loop, "‡" means that our result is strengthened by incremental method [Bradley 2012] which is a strategy to strengthen an invariant using previously generated invariants step-by-step, "★" means that our result is obtained by invariant propagation.

First, Table 1 presents the experimental results on our approach with invariant propagation. Note that the runtime for all benchmarks are mostly very short (within 0.2s) and thus we only consider the comparison in accuracy. In Table 1, one can observe that our approach mostly generates invariants with better accuracy. In detail, our approach could derive significantly tighter disjunctive invariants for benchmarks in Riley and Fedyukovich [2022]; Sharma et al. [2011], and generate precise LR results of program in disjunctive form for benchmarks in Ancourt et al. [2010]; Sharma et al. [2011]. On several benchmarks (such as *eudiv, minver* in Boutonnet and Halbwachs [2019]) that require incremental method, we run our approach twice for which the second run generates more invariants based on those obtained in the first run and obtain tighter invariants. Finally, our approach could also resolve nested loops with complex control flow such as *janne_complex, minver, fft1* in Boutonnet and Halbwachs [2019]. We relegate the detailed invariants in the original papers to Appendix E.

Table 2. Experimental Results on Loop Summary

Benchmark		Our Approach			
Name		Type	Time	v.s.	Detailed Result
Riley and Fedyukovich [2022]	fig2 ★	Smry	0.03s	+	$(x_0 \le x \wedge y=y_0 \wedge z=z_0) \vee$ $(x\text{-}1000y=z\text{-}z_0 \wedge y=y_0 \wedge x_0+1 \le 1000y$ $\wedge\ 1000y \le x \le 1000y+999) \vee$ $(z=z_0+1000 \wedge y=y_0$ $\wedge\ x_0 \le 1000y\text{-}1 \wedge 1000y \le x\text{-}1000)$
Xie et al. [2016]	fig1a ★ ‡		0.01s	>	$(x=z=n=n_0 \wedge x_0 \le z_0\text{-}1 \wedge z_0 \le n\text{-}1) \vee$ $(x_0+1 \le x=n=n_0 \le z_0=z) \vee$ $(x=z=n=n_0 \wedge z_0 \le x_0 \le n\text{-}1)$
	fig6a ★		0.02s	=	$i_0=j_0=0 \wedge m=m_0 \wedge i=n=n_0 \wedge j=0 \wedge 1 \le m \le n\text{-}1$
	fig1c ★		< 0.01s	>	$1 \le j \le m\text{-}1 \wedge m \le i \le 2m\text{-}2 \wedge 1 \le k \le m\text{-}i_0$
	fig1f ★		0.02s	>	$(s=1 \wedge x_1\text{-}x_2 = x_{10}\text{-}x_{20} \wedge x_{10} \le x_1) \vee$ $(s=2 \wedge x_1\text{-}x_2\text{-}1 = x_{10}\text{-}x_{20} \wedge 1 \le x_1\text{-}x_{10}) \vee$ $(s=3 \wedge x_1\text{-}x_2 = x_{10}\text{-}x_{20} \wedge 1 \le x_1\text{-}x_{10}) \vee$ $(s=4 \wedge x_1\text{-}x_2 = x_{10}\text{-}x_{20} \wedge 1 \le x_1\text{-}x_{10})$
Boutonnet and Halbwachs [2019]	eudiv ★ ‡		0.01s	=	$a=a_0 \wedge b=b_0 \wedge r \ge 0 \wedge b \ge r+1 \wedge a+1 \ge b+q+r \wedge q \ge 1$
	correct1 ★ ‡		< 0.01s	+	$(x\text{-}x_0=e\text{-}e_0 \wedge x_0=o+e_0 \wedge t \ge 0 \wedge x\text{-}x_0\text{-}t+s+e_0 \ge 1$ $\wedge\ x_0 \ge x+s \wedge x_0+t \ge e_0+x)$
	janne_complex ⊛		34.34s	+	$(x_0 \le 29 \wedge y_0 \le 5 \wedge y_0 \le x_0 - 1 \wedge x \le y + 12$ $\wedge\ \text{-}36x\text{-}12x_0+y\text{-}18y_0 \ge \text{-}1811 \wedge \text{-}36x\text{-}61x_0+y\text{-}18y_0 \ge \text{-}3036$ $\wedge\ \text{-}107x\text{-}52x_0\text{-}12y\text{-}78y_0+6639 \ge 0 \wedge 3x\text{-}y \ge 22$ $\wedge\ x \ge 30 \wedge 2x\text{-}2x_0+y_0 \ge 12 \wedge x\text{-}x_0 \ge 4) \vee$ $(x_0 \le 29 \wedge y_0 \ge x_0 \wedge 30 \le x \le 31 \wedge x \le y+12$ $\wedge\ 5x\text{-}5x_0+y\text{-}y_0 \ge 0 \wedge 13x\text{-}13x_0\text{-}3y+3y_0 \ge 0$ $\wedge\ 127x\text{-}155x_0\text{-}25y+25y_0+308 \ge 0$ $\wedge\ 297x\text{-}297x_0\text{-}47y+47y_0\text{-}1064 \ge 0)$
	minver ★ ⊛		0.01s	+	$(i_0 \le 2 \wedge j_0 \le 2 \wedge i=j=3) \vee$ $(i_0 \le 2 \wedge j_0 \ge 3 \wedge i=3 \wedge j=j_0)$
	fft1 ★ ⊛		0.10s	+	$(n=n_0 \wedge m=m_0 \wedge i_0+1 \le i=m+1 \wedge j_0 \ge n+1$ $\wedge\ k+1 \le j \le 2k \wedge 3k+1 \le j+n) \vee$ $(n=n_0 \wedge m=m_0 \wedge i=m+1 \ge i_0+2$ $\wedge\ 2k \ge j \ge k+1 \wedge 3k+1 \le j+n \wedge j_0 \le n) \vee$ $(k=n=n_0 \wedge m=m_0 \wedge i=m+1 \ge i_0+1$ $\wedge\ 2k \ge j \wedge k \ge j_0 \wedge k \ge j)$
WCET[Gustafsson et al. 2010]	cnt_cover ★		< 0.01s	+	$c=c_0+10 \wedge cnt=cnt_0+10$
	cnt_minver ★ ⊛		0.05s	+	$(i_0=2 \wedge j_0 \le 2 \wedge i=j=3 \wedge cnt_1=1) \vee$ $(i_0 \le 1 \wedge j_0 \le 2 \wedge i=j=3) \vee$ $(i_0 \le 2 \wedge j \ge 3 \wedge i=3 \wedge j=j_0 \wedge i_0+cnt_2=3)$
	cnt_fft1 ★ ⊛		0.20s	+	$(n=n_0 \wedge m=m_0 \wedge i_0+cnt_1=i=m+1 \wedge j_0 \ge n+1$ $\wedge\ k+1 \le j \le 2k \wedge 3k+1 \le j+n \wedge i\text{-}i_0 \ge cnt_2 \ge 1) \vee$ $(n=n_0 \wedge m=m_0 \wedge i=m+1=i_0+cnt_1 \wedge 2k \ge j \ge k+1$ $\wedge\ 3k+1 \le j+n \wedge j_0 \le n \wedge i\text{-}i_0\text{-}1 \ge cnt_2 \ge 1) \vee$ $(k=n=n_0 \wedge m=m_0 \wedge i=m+1=i_0+cnt_1 \wedge 2k \ge j$ $\wedge\ k \ge j_0 \wedge k \ge j \wedge i\text{-}i_0\text{-}1 \ge cnt_2 \ge 0)$
SPEED[Gulwani et al. 2009]	cnt_SimpleSingle ★		< 0.01s	+	$(x=n_0 \wedge cnt_1 = cnt_2 + cnt_3 \wedge x = x_0 + cnt_2 + cnt_3$ $\wedge\ x = n \wedge cnt_2 \ge 1 \wedge cnt_3 \ge 0) \vee$ $(x=n_0 \wedge cnt_1 = cnt_2 + cnt_3 \wedge x = x_0 + cnt_2 + cnt_3$ $\wedge\ x = n \wedge cnt_2 \ge 0 \wedge cnt_3 \ge 1)$
	cnt_SimpleSingle2 ★		0.10s	+	$(cnt_1 = cnt_2 \wedge m = m_0 \wedge x = x_0 + cnt_2 \wedge cnt3 = 0$ $\wedge\ y = y_0 + cnt_2 \wedge x = n \wedge x = n_0 \wedge cnt_2 \ge 1 \wedge y \ge m\ ) \vee$ $(y = m \wedge y = m_0 \wedge x = x_0 + cnt_1 \wedge x = x_0 + cnt_2 + cnt_3$ $\wedge\ x = x_0 + y - y_0 \wedge x_0 = n - cnt_2$ $\wedge\ x_0 = n_0 - cnt_2 \wedge cnt_2 \ge 1 \wedge x \ge x_0 + cnt_2 + 1) \vee$ $(y = m_0 \wedge cnt_2 = 0 \wedge x = x_0 + cnt_1 \wedge x = x_0 + cnt_3$ $\wedge\ x = x_0 + y - y_0 \wedge y = m \wedge n = n_0 \wedge x \ge n + 1 \wedge x \ge x_0 + 1\ )$
	cnt_SimpleMultiple ★		0.05s	+	$(y = m_0 \wedge y = m \wedge x = x_0 + cnt_1 - cnt_2 \wedge cnt_1 = cnt_2 + cnt_3$ $\wedge\ y = y_0 + cnt_2 \wedge x = n \wedge x = n_0 \wedge cnt_1 \ge cnt_2 + 1 \wedge cnt_2 \ge 1) \vee$ $(x = x_0 + cnt_1 \wedge cnt_2 = 0 \wedge m = m_0 \wedge x = x_0 + cnt_3$ $\wedge\ y = y_0 \wedge x = n \wedge x = n_0 \wedge x \ge x_0 + 1 \wedge y \ge m)$
	cnt_NestedMultiple ★ ⊛		0.03s	+	$(y=m_0 \wedge cnt_2=1 \wedge x=x_0+1 \wedge cnt_3=0$ $\wedge\ cnt_1=1 \wedge x=n \wedge x=n_0 \wedge y=m \wedge y \ge y_0+1) \vee$ $(y=m_0 \wedge cnt_2=1 \wedge x=x_0+cnt_1 \wedge cnt_1=cnt_3+1$ $\wedge\ y=m \wedge x=n \wedge x=n_0 \wedge y \ge y_0+1 \wedge cnt_1 \ge 2) \vee$ $(x=x_0+cnt_1 \wedge cnt_2=0 \wedge m=m_0 \wedge x=x_0+cnt_3$ $\wedge\ y=y_0 \wedge x=n \wedge x=n_0 \wedge x\ge x_0+1 \wedge y \ge m)$

Table 3. Experiment for SeaHorn, CPAChecker, VeriAbs and OOPSLA23

Benchmark		Our Approach		SeaHorn		CPAChecker		VeriAbs		OOPSLA23	
Name		Proof	Time (s)	Proof	Time (s)	Proof	Time (s)	Proof	Time (s)	Proof	Time (s)
Riley and Fedyukovich [2022]	fig2★	T	0.02	F	> 36000	F	3	F	27	F	1
Ancourt et al. [2010]	Gopan07★		0.01	F	> 36000	T	17	T	30	F	1
	Halbwachs★		0.01	F	> 36000	T	30	T	33	F	1
	Gulwani07★		0.01	T	1	T	16	T	18	T	2
Sharma et al. [2011]	CAV06★		< 0.01	F	> 36000	T	12	T	35	F	1
	ex1★		0.01	F	> 36000	T	14	T	35	F	1
	POPL07★		0.02	T	1	T	11	T	19	T	2
	ex2★		0.02	T	2	T	13	T	20	T	2
Xie et al. [2016]	fig1a★		0.01	F	1	F	10	F	21	F	4
	fig1c★		< 0.01	F	1	F	10	F	27	F	1
	fig6a★		0.02	F	1	F	11	F	874	F	1
	fig1f★		0.01	T	1	F	445	F	513	T	3
Boutonnet and Halbwachs [2019]	eudiv★‡		0.01	F	1	F	10	F	18	F	1
	janne_complex⊛		28.24	F	1	F	9	F	16	F	1
	minver★ ‡ ⊛		< 0.01	F	1	F	9	F	898	F	2
	fft1★⊛		0.08	F	1	F	9	F	1	F	1
	correct1★		0.01	F	1	T	10	F	19	F	2
Henry et al. [2012]	fig1★		0.01	T	1	T	17	T	18	F	2
SV-COMP [2023]	benchmark44_disjunctive.c★		0.01	F	1	F	1	F	32	F	1
	count_by_nondet.c★		0.01	F	> 36000	F	1	F	901	F	3
	mono-crafted_6.c★		0.01	F	> 36000	F	1	T	236	T	2
	mono-crafted_9.c★		0.01	F	> 36000	F	1	T	270	T	2
	mono-crafted_13.c★		0.01	F	> 36000	F	1	T	209	T	2
	Mono4_1.c★		0.01	F	> 36000	F	1	F	403	T	2
	Mono5_1.c★		0.01	F	> 36000	F	1	F	405	T	3
	Mono6_1.c★		0.02	F	> 36000	F	1	F	403	T	2
	gcnr2008.c★		0.02	F	1	F	1	F	62	F	1
	gr2006.c★		0.02	F	> 36000	T	17	T	85	F	1
	benchmark07_linear.c★		0.01	T	1	T	10	F	30	F	1
	benchmark21_disjunctive.c★		0.01	T	1	T	11	T	31	F	1
	benchmark32_linear.c★		0.01	T	1	T	8	T	17	F	1
	benchmark51_polynomial.c★		0.01	T	1	T	10	T	17	F	1
	afnp2014.c★		0.01	T	1	T	66	T	22	T	2
	eq1.c★		0.01	T	1	T	10	F	223	F	1
	gj2007.c★		0.01	T	1	T	11	T	51	T	2
	nested_5.c★⊛		0.03	T	2	T	8	T	23578	F	1
	terminator_02-2.c★		0.02	T	1	T	9	T	19	F	1
	nested_6.c★⊛		0.02	T	1	T	9	T	15	F	1
	sum01_bug02.c★		0.01	T	1	T	9	F	1	F	1
	sum01_bug02_sum01_bug02_base.case.c★		0.01	T	1	T	9	F	1	F	1
	nested_delay_notd2.c★		0.02	T	1	F	1	F	208	F	1
	benchmark06_conjunctive.c★		0.01	T	2	F	1	F	853	F	1
	benchmark31_disjunctive.c★		0.01	T	2	F	1	F	30	F	1
	benchmark45_disjunctive.c★		0.01	T	2	F	1	F	28	F	1
	benchmark46_disjunctive.c★		0.01	T	2	F	1	F	29	F	1
	benchmark47_linear.c★		0.01	T	1	F	1	F	33	F	1
	bhmr2007.c★		0.02	T	1	F	1	F	899	F	1
	cggmp2005_variant.c★		0.01	T	1	F	1	T	193	T	2
	ddlm2013.c★		0.02	T	1	F	1	F	901	T	2
	half.c★		0.01	T	1	F	1	F	811	T	2

Second, Table 2 presents the experimental results on affine disjunctive loop summary. We first compare our generated loop summaries with existing results in Xie et al. [2016] and Boutonnet and Halbwachs [2019] (for eudiv, correct1), and find that our approach mostly generate more accurate loop summaries. Then we test our approach on WCET benchmarks in Gustafsson et al. [2010] related to affine loop summary and adapt Speed benchmarks in Gulwani et al. [2009] to affine runtime behaviour by fixing the number of loop iterations in either the outer or the inner loop. In these benchmarks, we use a special variable cnt to represent the number of loop iterations of an outer/inner loop. The results for these benchmarks were previously not reported, and our results show that our approach generates precise affine disjunctive loop summaries for these benchmarks.

Third, Table 3 presents the comparison with the state-of-the-art software verifiers SeaHorn [SeaHorn 2015], CPAChecker [CPAchecker 2022], Veriabs [Darke et al. 2021] and the tool from Wang and Lin [2023] (the column "OOPSLA23" in the table) . We first have the comparison over the benchmarks in Table 1 (i.e., the benchmarks except for "SV-COMP" in Table 3). For this part of benchmarks, since both SeaHorn and CPAChecker require the user to provide a goal property, we feed them simple goal properties such as the equality between variables and constants (e.g., $x = y$, $x = 100$, etc.) arising

from the disjunctive feature of the benchmarks. Then we choose representative benchmarks with disjunctive feature from the categories *loop-new, loop-lit, loop-crafted-1, loops, loop-invariants, loop-zilu, loop-simple* of SV-COMP and compare the results between our approach and Seahorn/CPAChecker. These benchmarks from SV-COMP covers typical disjunctive features including *multi-phase loop, loop with if-else or if-else-break, loop with nondeterminism-branch, loop with switch-case, loop under non-initialized variables, mode transition, nested loops*. We keep the original assertions for the benchmarks from SV-COMP. In the table, the columns "SeaHorn"/"CPAChecker"/"Veriabs"/"OOPSLA23" mean the results generated by SeaHorn/CPAChecker/Veriabs/OOPSLA23, 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 incapable (resp. capable) of checking the assertions respectively. We set a time-out of 10 hours in this table. One can observe that these tools fail on most of the benchmarks even if these benchmarks are at a small scale, while our approach succeeds in checking the assertions in all the benchmarks and is substantially more time efficient. We find that the reasons behind these tools include failure to handle **break**-statement such as *Gopan07*, incapability to handle non-initialized variables such as *fig1a, fig6a, fig1c, eudiv, correct1, janne_complex, minver, fft1*, incompetence to handle disjunction such as *Halbwachs*, insufficient to handle nested loops with complex control flow such as *janne_complex, minver, fft1*, etc.

Table 4. Experiment for Invariant Propagation

Benchmark				Our Approach		
				No PPG	PPG	
Name		Loc	Dim	Time (s)	Time (s)	Speedup
POPL07★ [Sharma et al. 2011]	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 4 demonstrates the improvement of speedup by our invariant propagation technique. In Table 4, "*r*-p" means that *r* is a benchmark-inside number to show how many locations are there in the ATS, "Loc" means the number of locations under ATS, "Dim" means the number of unknown coefficients at all locations, "No PPG" means using our disjunctive affine invariant generation over each location under ATS without invariant propagation (i.e., following the original approach in Liu et al. [2022]), "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 4 show that our invariant propagation could substantially improve the time efficiency over large benchmarks.

Remark 3 (Other Related Approaches). *We are unable to have direct comparison with the very related work Boutonnet and Halbwachs [2019]; Henry et al. [2012]; Lin et al. [2021]; Riley and Fedyukovich [2022]; Xie et al. [2016] due to the following reasons. First, the works Boutonnet and Halbwachs [2019]; Lin et al. [2021]; Xie et al. [2016] neither publicize their implementation nor report the detailed invariants in some key benchmarks such as* janne_complex, minver, fft1. *Second, although the tool PAGAI [Henry et al. 2012] claims the functionality of disjunctive invariant generation, we find that this functionality could not work in the disjunctive-invariant-generation mode. Third, the tool in Riley and Fedyukovich [2022] accepts only the smtlib format of the CHC solver and has a preprocessing on the original CHC input, making the recovery of the original loop information difficult. We have tried the submodules of SeaHorn [SeaHorn 2015] and Eldarica [Eldarica 2022] to transform*

several simple examples (e.g., Gopan07 *and* POPL07*) in this paper into their CHC format, but this tool does not terminate on the CHC inputs of these simple examples. We also note that machine learning approaches [Ryan et al. 2020; Si et al. 2018; Yao et al. 2020] could also generate disjunctive invariants, but we found robustness problem that a slight deviation in a simple program (without changing the branch structure in the loop) can cause these approaches non-terminating. Our approach is based on constraint solving and therefore does not have this robustness issue.* □

7 RELATED WORKS

Below we compare our approach with the most related approaches in the literature. We first have the comparison with the constraint-solving approaches.

- Our approach uses the framework to apply Farkas' Lemma as proposed in Colón et al. [2003]; Liu et al. [2022]; Sankaranarayanan et al. [2004b] and extend the framework to disjunctive affine invariants and loop summary, for which our basic contribution is the construction of an affine transition system that reflects the disjunctive feature from the conditional branches in a loop. Furthermore, we propose invariant propagation to improve the time efficiency, and the use of loop summary to handle nested loops. The recent result [Ji et al. 2022] also considers Farkas' Lemma, but focuses on conjunctive affine 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 affine invariant generation include de Oliveira et al. [2017]; Gulwani et al. [2008]; Gupta and Rybalchenko [2009]. The approach [Gulwani et al. 2008] solves the quadratic constraints derived from Farkas' Lemma by SAT solvers and bit-vector modeling. The approach [de Oliveira et al. 2017] uses eigenvectors to handle several restricted classes of conjunctive affine invariants. The tool InvGEN [Gupta and Rybalchenko 2009] generates conjunctive affine invariants by an integrated use of abstract interpretation and Farkas' Lemma. These approaches propose completely different techniques, and thus are orthogonal to our approach.
- Since our approach targets affine invariant generation, it is incomparable with previous results on polynomial invariant generation [Adjé et al. 2015; Chatterjee et al. 2020; Chen et al. 2015; Cousot 2005; de Oliveira et al. 2016; Hrushovski et al. 2018; Humenberger et al. 2017; Kapur 2005; Lin et al. 2014; Rodríguez-Carbonell and Kapur 2004b; Sankaranarayanan et al. 2004a; Yang et al. 2010]. Moreover, most of these approaches consider only conjunctive polynomial invariants, and hence do not consider disjunction.

It is also worth noting that the previous work [Sharma et al. 2011] proposes a general framework for detecting multiphase disjunctive invariants that can be instantiated with constraint solving. Multiphase disjunctive invariants are a special case of our disjunctive pattern (that considers standalone conjunctive invariants at top-level branches) in the sense that each phase in a multiphase feature is directly captured by its phase condition as a top-level branch in our approach. Therefore, we consider a wider class of disjunctive invariants as compared with Sharma et al. [2011].

Second, we compare our approach with the results [Lin et al. 2021; Xie et al. 2016] that consider a similar disjunctive pattern to ours. These approaches propose *path dependency automata* that consider different execution paths of the loop body w.r.t whether each conditional branch in the loop body is entered or not, treat each execution path as a standalone mode, and have transitions between these modes. However, an indispensible ingredient of path dependency automata is the exact estimation of the number of loop iterations sojourning in each mode, and hence is limited to inductive variables (i.e., assignments must be in the form $x := x + c$ or $x := c * x$) and strict alternation between different modes. Our approach directly constructs affine transition systems between different top-level branch locations, and hence do not have such limitation. Moreover, we

extend our approach to handle nested loops via loop summary, while these approaches could not have an adequate support for nested loops.

Third, we compare our approach with abstract interpretation. Compared with the approaches that generate conjunctive affine invariants via polyhedral abstract domain [Bagnara et al. 2003; Cousot and Halbwachs 1978; Singh et al. 2017], our approach targets the more general case of disjunctive affine invariants. There are also a bunch of abstract-interpretation approaches in disjunctive affine invariant generation, such as the work [Gopan and Reps 2007] that performs disjunctive partitioning by representing the contribution of each iteration with a separate abstract-domain element, the recent work [Boutonnet and Halbwachs 2019] 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 [Henry et al. 2012] 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 our disjunctive pattern and techniques, and hence are orthogonal to our approach.

Fourth, we compare our approach with recurrence analysis [Farzan and Kincaid 2015; Kincaid et al. 2017, 2018]. Recurrence analysis usually relies on the existence of a closed form solution. For example, the very recent most related recurrence analysis approach [Wang and Lin 2023] (that also targets disjunctive invariants and loop summary) requires the (ultimate) strict alternation between top-level conditional branches to ensure the existence of a closed form solution, so that the applicability of this approach is limited and does not include nested loops. Our approach is not limited by the absence of a closed form solution.

Fifth, we compare our approach with other methods such as machine learning, inference and data-driven approaches. Unlike constraint solving that can have an accuracy guarantee for the generated invariants based on the constraints, these methods cannot have an accuracy guarantee. Furthermore, machine learning and data-driven approaches themselves cannot guarantee that the generated assertions are indeed invariants. Moreover, our approach can generate invariants *without* the need of a goal property, while these approaches usually requires a goal property. Note that the invariant generation without a given goal property is a classical setting (see e.g. Colón et al. [2003]; Cousot and Halbwachs [1978]), and has applications in loop summary and probabilistic program verification (see e.g. Chakarov and Sankaranarayanan [2013]; Wang et al. [2021]).

Finally, we compare our approach with the related approaches on loop summary. Compared with the approaches [Cousot and Cousot 2001, 2002] that are based on convex polyhedra abstract domain and can only generate conjunctive affine loop summaries, our approach is able to generate disjunctive loop summaries. Compared with the approach by Kranz and Simon [2018] that applies Heyting completion [Giacobazzi and Scozzari 1998] (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 affine 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 [2019] 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 top-level branches, which are completely different. Compared with (i) the PIPS tool [Ancourt et al. 2010; Irigoin et al. 1991; PIPS 2022] that employs heuristics to generate conjunctive affine loop summaries and (ii) the approach by Ancourt et al. [2010] that generates conjunctive affine invariants by a simple heuristics that examines the stepwise incremental update of affine assignments, our approach follows a completely different methodology and generates disjunctive affine loop summary. Compared with the approaches by Popeea and Chin [2006, 2013] 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. It would also be an interesting future direction to investigate how our approach could be used for procedure summary [Allen 1974; Gulwani and Tiwari 2007; Zhang et al. 2014].

8 CONCLUSION AND FUTURE WORK

In this work, we proposed a novel approach to generate affine disjunctive invariants and loop summaries via Farkas' Lemma. Experimental results show that our approach is capable of deriving substantially more accurate affine disjunctive invariants and loop summaries against existing approaches. One future direction would be to consider extensions in Remark 2. Another interesting direction is to extend our approach to procedure summary.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

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.

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

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

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.

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

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

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

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

Amir M. Ben-Amram and Samir Genaim. 2017. On Multiphase-Linear Ranking Functions. In *CAV (LNCS, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 601–620. https://doi.org/10.1007/978-3-319-63390-9_32

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

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

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

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

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

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

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

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

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

Clang Static Analyzer 2022. Clang Static Analyzer: A source code analysis tool that finds bugs in C, C++, and Objective-C programs. https://clang-analyzer.llvm.org/.

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

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

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

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

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

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

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

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

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

Priyanka Darke, Sakshi Agrawal, and R Venkatesh. 2021. VeriAbs: A tool for scalable verification by abstraction (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27*. Springer, 458–462.

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

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

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

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

Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software Verification Using k-Induction. In *SAS (LNCS, Vol. 6887)*, Eran Yahav (Ed.). Springer, 351–368. https://doi.org/10.1007/978-3-642-23702-7_26

Eldarica 2022. Eldarica: A model checker for Horn clauses, Numerical Transition Systems, and software programs. https://github.com/uuverifiers/eldarica.

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

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

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

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

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

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

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

Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 127–139. https://doi.org/10.1145/1480881.1480898

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

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

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

Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks – Past, Present and Future, Björn Lisper (Ed.). OCG, Brussels, Belgium, 137–147.

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

Antoine Miné. 2004. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2986)*, David A. Schmidt (Ed.). Springer, 3–17. https://doi.org/10.1007/978-3-540-24725-8_2

Antoine Miné. 2006. Symbolic Methods to Enhance the Precision of Numerical Abstract Domains. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3855)*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). Springer, 348–363. https://doi.org/10.1007/11609773_23

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

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

PIPS 2022. PIPS. https://pips4u.org

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

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

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

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

Enric Rodríguez-Carbonell and Deepak Kapur. 2004a. 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

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

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

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

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

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

SeaHorn 2015. SeaHorn: A fully automated analysis framework for LLVM-based languages. http://seahorn.github.io.

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

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

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

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 7762–7773. https://proceedings.neurips.cc/paper/2018/hash/65b1e92c585fd4c2159d5f33b5030ff2-Abstract.html

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.

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

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

StInG 2006. StInG: Stanford Invariant Generator. http://theory.stanford.edu/~srirams/Software/sting.html.

SV-COMP 2023. Software Verification Competition. https://sv-comp.sosy-lab.org.

Chenglin Wang and Fangzhen Lin. 2023. Solving Conditional Linear Recurrences for Program Verification: The Periodic Case. In *OOPSLA*. ACM. to appear.

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

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

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

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

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

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 PROCESS OF TRANSFORMATION TO CANONICAL FORM

Here, we provide a detailed demonstration of how to transform an affine program P into its canonical form, as shown in Figure 8 and Figure 12:

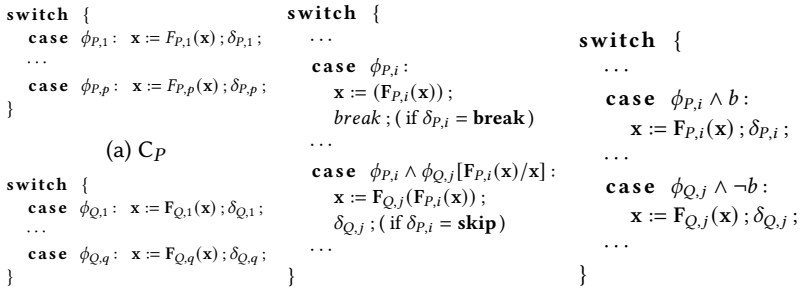


Fig. 12. The canonical form of transformation (TF) for P, Q

- 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 12a and Figure 12b respectively, and then compute C_R as in Figure 12c 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 already breaks 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 12d 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 $\neg b$).

B 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 $\mu = k, \forall k > 0$.

We scale the leftmost coefficient column k, λ_i's to be 1, λ_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.

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}.

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

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

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

Fig. 13. Constraint consecution tabular

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

Fig. 14. Transformed constraint consecution tabular

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

Fig. 15. Equivalent constraint transformation tabular

C 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.

D PROOF FOR CORRECTNESS AND ACCURACY FOR OUR INVARIANT PROPAGATION

Below we prove the theoretical properties that the affine assertions generated from our invariant propagation are indeed invariants, and are at least as tight as the invariants generated from the previous approaches [Liu et al. 2022; Sankaranarayanan et al. 2004b].

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

Proof. Let Γ be an ATS 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 affine assertion at the root (i.e., the initial location) is correct since it is generated by the approach [Liu et al. 2022]. The inductive step is to show that if the affine assertions generated at the nodes of the current level are invariants, then so are the affine assertions at the next level. The proof for the inductive step follows from the fact that any path of the ATS 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 ATS $\Gamma[\ell', K_{\tau,i}]$ in a single propagation step. □

Proposition 3. *The invariant propagation generates invariants at least as tight as the previous approaches [Liu et al. 2022; Sankaranarayanan et al. 2004b].*

Proof. The proof proceeds via an induction on the BFS level of the invariant propagation. For the base step, we have that the affine invariant generated at the root is generated directly from the previous approach [Liu et al. 2022]. Then the base step follows from the fact that the approach [Liu et al. 2022] has the same precision as the original approach [Sankaranarayanan et al. 2004b]. 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 [Sankaranarayanan et al. 2004b]. We prove that the implication holds for the next BFS level. The proof can be obtained by observing that each individual affine inequality (as a conjunctive inequality in an affine assertion) in the invariants generated by the approach [Sankaranarayanan et al. 2004b] 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 [Sankaranarayanan et al. 2004b] (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 affine inequality generated by the approach [Sankaranarayanan et al. 2004b]. □

E FULL EXPERIMENTAL RESULTS ON INVARIANT GENERATION AND LOOP SUMMARY COMPARED WITH ORIGINAL RESULTS

Table 5. Full Experimental Results on Invariant Generation for Table 1

Benchmark		Comparison				
Name		Type	Time	v.s.	Original Result	Our Result
Riley and Fedyukovich [2022]	fig2 ★	Dis	0.02s	>	(y>0 y>x/1000 $\implies$ z=0) ∧ (y>0 y=x/1000 $\implies$ z=x-1000y) ∧ (y>0 y<x/1000 $\implies$ z=1000)	(z=0 ∧ 0 ≤ x ≤ 1000y-1 ∧ 1 ≤ y) ∨ (x-1000y=z ∧ x-999 ≤ 1000y ≤ x ∧ 1 ≤ y) ∨ (z=1000 ∧ 1 ≤ y ∧ 1000y ≤ x-1000)
Ancourt et al. [2010]	Gopan07 ★	LR	<0.01s	>	x=102	x=102 ∧ y=-1
	Gulwani07 ★	LR	0.01s	>	y=100	x=y=100
	Halbwachs ★	LR	0.01s	>	2≤x+y ∧ y≤x ∧ x+y≤202	(101 ≤ x ≤ 102 ∧ 0 ≤ y ∧ y+2 ≤ x) ∨ (x=101 ∧ 1 ≤ y ≤ 101)
Sharma et al. [2011]	POPL07 ★	Dis	<0.01s	>	(y=50 ∧ x ≤ 50) ∨ (x=y ∧ 50 ≤ x ≤ 100)	(y=50 ∧ 0 ≤ x ≤ 49) ∨ (x=y ∧ 50 ≤ x ≤ 99)
		LR	<0.01s	>	y=100	x=y=100

Table 6. Full Experimental Results on Loop Summary for Table 2

Benchmark		Comparison				
Name		Type	Time	v.s.	Original Result	Our Result
Xie et al. [2016]	fig1a ★ ‡	Smry	0.01s	>	($x_0 \geq n_0 \wedge x=x_0 \wedge z=z_0$) ∨ ($x_0 < n_0 \leq z_0 \wedge x=n_0 \wedge z=z_0$) ∨ ($x_0 < n_0 \wedge z_0 < n_0 \wedge x=z=n_0$)	(x=z=n=n_0 ∧ $x_0 \leq z_0$-1 ∧ $z_0 \leq$ n-1) ∨ (x_0+1 ≤ x=n=$n_0 \leq z_0$=z) ∨ (x=z=n=$n_0 \wedge z_0 \leq x_0 \leq$ n-1)
	fig1c ★		< 0.01s	>	i < m	$1 \leq j \leq$ m-1 ∧ m ≤ i ≤ 2m-2 ∧ $1 \leq k \leq$ m-i_0
	fig1f ★		0.02s	>	$x_{10} \geq 0 \wedge x_{20} \geq 0$	(s=1 ∧ x_1-x_2 = x_{10}-x_{20} ∧ $x_{10} \leq x_1$) ∨ (s=2 ∧ x_1-x_2-1 = x_{10}-x_{20} ∧ $1 \leq x_1$-x_{10}) ∨ (s=3 ∧ x_1-x_2 = x_{10}-x_{20} ∧ $1 \leq x_1$-x_{10}) ∨ (s=4 ∧ x_1-x_2 = x_{10}-x_{20} ∧ $1 \leq x_1$-x_{10})

F FULL INNER CAESS FOR FIGURE 9

```
while(x < 30){
 switch
  case :
   ;Skip;
  case x <= 29, 6 <= y, y <= x − 1 :
   x′ <= 29, y′ >= 6, y′ − x′ <= −1, 36x′ − y′ <= 36x − 3y + 18, 3x′ − y′ >= 22, x′ >= x + 3, x′ − y′ <= 12 ; Skip ;
  case x <= 29, 6 <= y, y <= x − 1 :
   x′ <= 29, y′ <= x′ − 1, y′ <= 5, 36x − 3y + 18 >= 36x′ − y′, 3x′ − y′ >= 22, x′ − x >= 3, x′ <= y′ + 12 ; Skip ;
  case x <= 29, 6 <= y, y <= x − 1 :
   x′ <= 29, y′ <= x′ − 1, y′ <= 5, 36x − 3y + 18 >= 36x′ − y′, 3x′ − y′ >= 22, x′ − x >= 3, x′ <= y′ + 12 ; Skip ;
  case x <= 29, 6 <= y, y <= x − 1 :
   x′ <= 29, y′ >= x′, 36x − 3y + 18 >= 36x′ − y′, 3x′ − y′ >= 22, x′ − x >= 3, x′ <= y′ + 12 ; Skip ;
  case x <= 29, 6 <= y, y <= x − 1 :
   x′ >= 30, 36x − 3y + 18 >= 36x′ − y′, 3x′ − y′ >= 22, x′ − x >= 3, x′ <= y′ + 12 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ <= 29, y′ <= x′ − 1, y′ <= 5, x′ <= y′ + 12, 36x − 18y + 141 >= 36x′ − y′ − 82, 6x − 3y + 6 >= 6x′, y′ − 22, 2x′ − 2x + y − 12 >= 0, x′ − x >= 4, 3x′ − y′ >= 22 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ <= 29, 6 <= y′, y′ <= x′ − 1, x′ <= y′ + 12, 36x − 18y + 141 >= 36x′ − y′ − 82, 6x − 3y + 6 >= 6x′ − y′ − 22, 2x′ − 2x + y − 12 >= 0, x′ − x >= 4, 3x′ − y′ >= 22 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ <= 29, y′ <= x′ − 1, y′ <= 5, x′ <= y′ + 12, 36x − 18y + 141 >= 36x′ − y′ − 82, 6x − 3y + 6 >= 6x′ − y′ − 22, 2x′ − 2x + y − 12 >= 0, x′ − x >= 4, 3x′ − y′ >= 22 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ <= 29, y′ >= x′, x′ <= y′ + 12, 36x − 18y + 141 >= 36x′ − y′ − 82, 6x − 3y + 6 >= 6x′ − y′ − 22, 2x′ − 2x + y − 12 >= 0, x′ − x >= 4, 3x′ − y′ >= 22 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ >= 30, x′ <= y′ + 12, 36x − 18y + 141 >= 36x′ − y′ − 82, 6x − 3y + 6 >= 6x′ − y′ − 22, 2x′ − 2x + y − 12 >= 0, x′ − x >= 4, 3x′ − y′ >= 22 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ <= 29, y′ <= x′ − 1, y′ <= 5, x′ − 2x + y = 0, x′ = y′, y + 2 <= x′, x′ <= 7 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ <= 29, 6 <= y′, y′ <= x′ − 1, x′ − 2x + y = 0, x′ = y′, y + 2 <= x′, x′ <= 7 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ <= 29, y′ <= x′ − 1, y′ <= 5, x′ − 2x + y = 0, x′ = y′, y + 2 <= x′, x′ <= 7 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ <= 29, y′ >= x′, x′ − 2x + y = 0, x′ = y′, y + 2 <= x′, x′ <= 7 ; Skip ;
  case x <= 29, y <= x − 1, y <= 5 :
   x′ >= 30, x′ − 2x + y = 0, x′ = y′, y + 2 <= x′, x′ <= 7 ; Skip ;
  case x <= 29, y >= x :
   x′ <= 29, y′ >= x′, x′ = x + 2, y′ = y − 10 ; Skip ;
  case x <= 29, y >= x :
   x′ <= 29, 6 <= y′, y′ <= x′ − 1, x′ = x + 2, y′ = y − 10 ; Skip ;
  case x <= 29, y >= x :
   x′ <= 29, y′ <= x′ − 1, y′ <= 5, x′ = x + 2, y′ = y − 10 ; Skip ;
  case x <= 29, y >= x :
   x′ <= 29, y′ <= x′ − 1, y′ <= 5, x′ = x + 2, y′ = y − 10 ; Skip ;
  case x <= 29, y >= x :
   x′ <= 29, y′ <= x′ − 1, y′ <= 5, x′ = x + 2, y′ = y − 10 ; Skip ;
  case x <= 29, y >= x :
   x′ >= 30, x′ = x + 2, y′ = y − 10 ; Skip ;
}
```

Fig. 16. The full janne_complex program after converted