Affine Disjunctive Invariant Generation with Farkas' Lemma

Jingyu Ke$^{1\,[0009-0008-6848-3105]}$, Hongfei Fu$^{1\boxtimes[0000-0002-7947-3446]}$, Hongming Liu$^{1\,[0000-0001-8987-596X]}$, Zhouyue Sun$^{1\,[0009-0009-8863-5548]}$, Liqian Chen$^{2\,[0000-0001-8084-8009]}$, and Guoqiang Li$^{1\,[0000-0001-9005-7112]}$

[1] Shanghai Jiao Tong University, China
`{Windocotber, jt002845, hm-liu, sunzhouyue, li.g}@sjtu.edu.cn`
[2] National University of Defense Technology, China
`lqchen@nudt.edu.cn`

Abstract. In the verification of loop programs, disjunctive invariants are essential to capture complex loop dynamics such as phase and mode changes. In this work, we develop a novel approach for the automated generation of affine disjunctive invariants for affine while loops via Farkas' Lemma, a fundamental theorem on linear inequalities. Our main contributions are two-fold. First, we combine Farkas' Lemma with a succinct control flow transformation to derive disjunctive invariants from the conditional branches in the loop. Second, we propose an invariant propagation technique that minimizes the invariant computation effort by propagating previously solved invariants to yet unsolved locations as much as possible. Furthermore, we resolve the infeasibility checking in the application of Farkas' Lemma which has not been addressed previously, and extend our approach to nested loops via loop summary. Experimental evaluation over more than 100 affine while loops (mostly from SV-COMP 2023) demonstrates that our approach is promising to generate tight linear invariants over affine programs.

1 Introduction

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 [56,62,2], reachability [20,10,3,63,16,29,4] and time-complexity [14] analysis. Invariant generation targets the automated generation of invariants which can be used to aid the verification of critical program properties.

Automated approaches for invariant generation have been studied for decades and there have been an abundance of literature along this line of research. From different program objects, invariant generation targets numerical values (e.g., integers or real numbers) [19,15,65,76,5,9], arrays [51,78], pointers [52,12], algebraic data types [46], etc. By different methodologies, invariant generation can be solved by abstract interpretation [22,24,9,37], constraint solving [21,19,39,15], inference [30,12,71,35,86,34,77,57,31], recurrence analysis [33,49,50], machine learning [36,41,88,67], data-driven approaches [52,73,58,26,17,64], 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 execution back and forth to the location.

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 capture disjunctive features such as multiple phases and mode transitions in loops. Although extensive research has been conducted on conjunctive invariant generation, verification of programs with complex disjunctive loops still demands a more precise and scalable approach, rather than merely generating conjunctive invariants at the loop entry point to summarize the entire loop.

Moreover, most of the existing disjunctive invariant analyses rely on specific program patterns, such as alternating loop paths [72] or periodic regular loops [85,54], which cannot be effectively generalized to real-world programs with arbitrary execution traces. Methods based on abstract interpretation or symbolic execution often fail to converge, or converge to low-precision invariants, when dealing with loops that are deeply nested or have large constant iteration counts. To address this, we propose a modular approach capable of handling complex loops and efficiently mitigating the computational explosion caused by the interleaving of paths in disjunctive loops.

We use constraint solving to generate real-valued affine disjunctive invariants. A typical constraint solving method is via Farkas' Lemma [19,69,45,55] that provides a complete characterization for affine invariants. However, the application of Farkas' Lemma is mostly limited to the conjunction of affine inequalities. The question on how to leverage Farkas' Lemma to affine disjunctive invariants remains to be a challenge. In this paper, we focus on the generation of affine disjunctive invariants in affine loops. An affine loop is a while loop where all conditional and assignment statements are in the form of linear expressions.

Our contributions. First, we introduce a novel control flow transformation that extracts loop paths (from entry to exit) as standalone locations in a transition system and establishes transitions between them. Second, to alleviate the exponential computational overhead introduced by the control flow transformation, we propose an invariant propagation technique that propagates already-computed invariants to locations whose invariants yet need to be computed as much as possible. Third, we fully resolve the infeasible situation in the application of Farkas' Lemma [69,55] and extend our approach to nested loops through loop summary. Fourth, we implement our approach as a prototype DInvG[‡]. Experimental evaluation with various state-of-the-art verification tools using over 100 benchmarks from SV-COMP 2023 [80] and [9], shows that our approach is both tight (in

[‡]The tool implementation is available on GitHub: `https://github.com/WindOctober/DInvG`.

the accuracy of the generated invariant) and is time efficient for real-valued disjunctive affine invariant generation.

The remainder of this paper is structured as follows. Sec. 2 revisits the fundamental definitions of affine transition systems and invariants, providing a variant of *Farkas' Lemma* as well as the basic definitions of polyhedra. Sec. 3 offers an overview of how our tool DInvG transforms programs and solves for disjunctive invariants. In Sec. 4, we formalize the definition of control flow transformation and extract the corresponding affine transition systems, present the pseudocode for invariant propagation as well as optimizations for nested loop and infeasible traces, and prove that the disjunctive invariants generated are inductive. Sec. 5 demonstrates the efficiency and precision advantages of DInvG compared to several state-of-the-art tools and conducts an ablation study on invariant propagation. Sec. 6 compares our method with related verification approaches, elaborating on the conceptual and implementation differences between invariant propagation and control flow transformation. A full version of this paper can be found at [48].

2 Preliminaries

Below we revisit affine transition systems [69] and their associated invariants, elucidate Farkas' Lemma, and outline fundamental principles from polyhedra theory. It is important to note that, within the scope of this paper, we treat linear and affine concepts equivalently.

2.1 Affine Transition Systems and Invariants

An *affine inequality* over a set $V = \{x_1, \ldots, x_n\}$ of real-valued variables is of the form $a_1x_1 + \cdots + a_nx_n + b \geq 0$, where a_i's and b are real coefficients. An *affine assertion* over V is a conjunction of affine inequalities over V.

An affine transition system possesses a finite number of locations as well as real-valued variables, and specifies transitions between locations with affine guards and affine updates on the values of the variables.

Definition 1 (Affine Transition Systems [69]). *An* affine transition system *(ATS) is a tuple $\Gamma = \langle X, X', L, \mathcal{T}, \ell^*, \theta \rangle$:*

- *X is a finite set of real-valued variables and $X' = \{x' \mid x \in X\}$ is the set of primed variables.*
- *L is a finite set of* locations *and $\ell^* \in L$ is the initial location.*
- *$\mathcal{T}$ is a finite set of* transitions *where each transition τ is a triple $\langle \ell, \ell', \rho \rangle$ from location ℓ to location ℓ' with the guard affine assertion ρ over $X \cup X'$.*
- *θ is a disjunction of affine assertions over X that specifies the initial condition at ℓ^*.*

The directed graph $DG(\Gamma)$ of the ATS Γ is defined as the graph where the vertices are the locations of Γ and there is an edge (ℓ, ℓ') if and only if there is a transition $\langle \ell, \ell', \rho \rangle$ with source location ℓ and target location ℓ'.

The intuition of an ATS $\mathit{\Gamma} = \langle X, X', L, \mathfrak{T}, \ell^*, \theta\rangle$ is as follows. Each variable $x \in X$ represents the current value of the variable and each primed variable $x' \in X'$ represents the next value of its unprimed variable $x \in X$ after one step of transition. The transition $\langle \ell, \ell', \rho\rangle$ specifies the jump from the current location ℓ to the next location ℓ' with the guard condition ρ specifying the condition to enable the transition. The guard condition involves both the current values (represented by X) and the next values (by X'), so that it can specify the relationship between the current and next values.

Below we describe the semantics of an ATS. 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 $\sigma(x) \in \mathbb{R}$. We mostly 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.

Given an affine assertion φ and a valuation σ over a variable set V, we write $\sigma \models \varphi$ to mean that σ satisfies φ, i.e., φ is true when one substitutes the corresponding values $\sigma(x)$ into all the variables x in φ. Given an ATS $\mathit{\Gamma}$, two valuations σ, σ' 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 $x' \in X'$ with $\sigma'(x)$ 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 case of disjunction of affine assertions is similar.

The semantics of an ATS $\mathit{\Gamma}$ is given by its paths. A *path* π of the ATS $\mathit{\Gamma}$ 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 $0 \le j \le k-1$, 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 $(\ell_k, \sigma_k) = (\ell, \sigma)$. 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 $0 \le i \le k$, it holds that $\ell_i = \ell$ implies $\sigma_i \models \varphi$. An invariant φ is *affine* if φ is an affine assertion over the variable set X, and is *disjunctively affine* if φ is a disjunction of affine assertions.

In invariant generation, one often investigates a strengthened version of invariants called *inductive invariants*. In this work, we present affine inductive invariants in the form of inductive affine assertion maps [19,69,55] as follows.

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. An AAM η is called *inductive* if the following holds:

- (**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 $x' \in X'$.

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.

2.2 Farkas' Lemma and Polyhedra

Farkas' Lemma [32] is a classical theorem in the theory of affine inequalities and previous results [19,69,55] have applied the theorem to affine invariant generation. In these results, the form of Farkas' Lemma follows [70, Corollary 7.1h].

Theorem 1 (Farkas' Lemma). *Consider an affine assertion φ over a set $V = \{x_1, \ldots, x_n\}$ 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 $\lambda_0, \lambda_1, \ldots, \lambda_m$ 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.*

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

(a) φ in Farkas' Lemma

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

(b) ψ in Farkas' Lemma

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

(c) The Tabular Form for Farkas' Lemma

Fig. 1: The φ, ψ and Tabular Form for Farkas' Lemma [19,69]

Farkas' Lemma simplifies the inclusion of a polyhedron inside a halfspace into the satisfiability of a system of affine inequalities. We refer to the case of unsatisfiable φ with $\psi := -1 \ge 0$ in the statement of Theorem 1 as *infeasible implication.* The application of Farkas' Lemma can be visualized by the tabular form in Figure 1c (taken from [19]), and we multiply $\lambda_0, \lambda_1, \ldots, \lambda_m$ with their inequalities in φ and sum up them together to get ψ. For $1 \le j \le m$, we require $\lambda_j \ge 0$.

A subset P of $\mathbb{R}^n$ is a *polyhedron* if $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A} \cdot \mathbf{x} \le \mathbf{b}\}$ 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 $\mathbf{A} \cdot \mathbf{x} \le \mathbf{b}$ is defined in the coordinate-wise fashion. A polyhedron P is a *polyhedral cone* if $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\cdot\mathbf{x} \le \mathbf{0}\}$ 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 [70, Corollary 7.1a] that any polyhedral cone P can be represented as $P = \{\sum_{i=1}^{k} \lambda_i \cdot \mathbf{g}_i \mid \lambda_i \ge 0 \text{ for all } 1 \le i \le k\}$ 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.

```
int x = 0, y = 50;
--------------------------------------------
                       while ( x < 100 ){
while ( x < 100 ){       case x > 49 :
  x = x + 1;               x = x + 1;
  if ( x > 50 )            y = y + 1;
    y = y + 1;           case x ≤ 49 :
}                          x = x + 1;
                       }
       P1                      P2
```

(a) Source P_1 and Its Transformation P_2.

$$X = \{x, y\},\ L = \{\ell_1, \ell_2^*\},\ \mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\},$$
$$\theta : x = 0 \wedge y = 50,\ \tau_1 : \langle \ell_1, \ell_1, \rho_1 \rangle,$$
$$\tau_2 : \langle \ell_1, \ell_2, \rho_2 \rangle,\ \tau_3 : \langle \ell_2, \ell_2, \rho_3 \rangle,\ \tau_4 : \langle \ell_2, \ell_1, \rho_4 \rangle,$$

$$\rho_1 : \left[\begin{array}{r} 50 \le x \le 99 \\ 50 \le x' \le 99 \\ x' = x + 1 \\ y' = y + 1 \end{array}\right], \rho_2 : \left[\begin{array}{r} 50 \le x \le 99 \\ x' \le 49 \\ x' = x + 1 \\ y' = y + 1 \end{array}\right]$$

$$\rho_3 : \left[\begin{array}{r} x \le 49 \\ x' \le 49 \\ x' = x + 1 \\ y' = y \end{array}\right], \rho_4 : \left[\begin{array}{r} x \le 49 \\ 50 \le x' \le 99 \\ x' = x + 1 \\ y' = y \end{array}\right]$$

(b) The ATS Corresponding to P_2

Fig. 2: An example from [72] and its transformed form and corresponding ATS

3 An Overview of Our Approach

Consider the affine program P_1 in Figure 2a. Our approach has three parts, namely control flow transformation, invariant computation and invariant propagation.

Control Flow Transformation. For the non-nested loop P_1, we extract each execution path from the loop entry to the exit and transform it into the form of loop P_2. Each **case** statement corresponds to a possible path in the original loop with a path condition ϕ of taking the path specified by the conjunction of a conditional formula ϕ_c and an assignment formula ϕ_a. For example, in the first case of P_2 that corresponds to the case of entering the if-branch in P_1, we have ϕ_a is $(x' = x + 1 \wedge y' = y + 1)$, ϕ_c is $x > 49$, and ϕ is $\phi_a \wedge \phi_c$.

Then, an ATS in Figure 2b is directly derived from the transformed program. In this example, each **case** statement is considered as an independent location in the ATS, e.g., the location ℓ_1 stands for the first **case** in P_2. The transitions between the locations are derived from the jumps between different paths, where each transition guard ρ for a transition τ, can be obtained through the formula:

$$\rho := \phi_c \wedge \phi_c'[x'/x] \wedge \phi_a \wedge G \wedge G[x'/x]$$

where, ϕ_a and ϕ_c represent the assignment and conditional formulas at the transition's start location, ϕ_c' represents the conditional formula at the transition's end location, and G is the loop guard. After the ATS is constructed, we apply the approaches [69,55] in Farkas' Lemma combined with the technique of invariant propagation to obtain invariants at all locations, and group them disjunctively together to obtain result invariants for original loop. Note that the construction of the ATS is the key connection to apply Farkas' Lemma.

Invariant Computation. We first establish affine invariant templates at each location in the ATS by setting:

$$\eta(\ell_i) := c_{\ell_i,1}x + c_{\ell_i,2}y + d_{\ell_i} \ge 0\ , \forall i \in \{1, 2\}$$

where, $c_{\ell_i,1}, c_{\ell_i,2}, d_{\ell_i}$ are unknown coefficients to be resolved. Then, we generate the constraints from the initialization as well as consecution conditions via the

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

(a) Initialization Tabular

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

(b) Consecution Tabular

Fig. 3: Tabular for Initialization and Consecution [55]

Farkas' tabular in Figure 1c and derive the initialization tabular and consecution tabular as shown in Figure 3. Setting $\varphi = \theta$ and $\psi = \eta(\ell^*)$ in Theorem 1, the initialization of $\theta \models \eta(\ell^*)$ results in linear constraints, while the consecution condition $\eta(\ell) \wedge \rho \models \eta(\ell')'$ by setting $\varphi = \eta(\ell) \wedge \rho$ and $\psi = \eta(\ell')'$ results in quadratic constraints since we have a fresh λ, which we denote as μ, multiplied by $\eta(\ell)$.

In this context, we adopt a unified notation by employing $\eta(\ell)$ to represent both affine expressions and affine inequalities interchangeably throughout the manuscript. To resolve the quadratic constraints from the consecution, the previous approach [69] has considered heuristics to guess the value of the multiplier μ through either practical rules such as factorization or setting μ manually to 0 or 1. These settings relax the original consecution definition into two stronger forms:

– **Local Consecution**: For transition $\tau : \langle \ell_i, \ell_j, \rho \rangle$, $\rho \models \eta(\ell_j)' \geq 0$,
– **Incremental Consecution**: For transition $\tau : \langle \ell_i, \ell_j, \rho \rangle$, $\rho \models \eta(\ell_j)' \geq \eta(\ell_i)$.

Then, we collect the derived constraints to constitute a formula Φ in CNF, which further expands into a DNF Φ'. Note that each disjunctive clause in the DNF Φ' is an affine assertion defining a polyhedral cone, and we solve for the unknown coefficients of the templates by computing the generators of these polyhedral cones. We present a polyhedral cone in the DNF Φ' of the ATS in Figure 4b, along with corresponding generators, as shown in Figure 4a, where

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

(a) A clause in the DNF

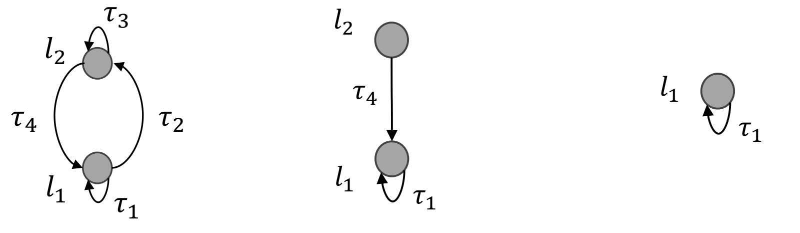
type	c_{11}	c_{12}	d_1	c_{21}	c_{22}	d_2	$\eta(\ell_1)$	$\eta(\ell_2)$
point	0	0	0	0	0	0	$0 \geq 0$	$0 \geq 0$
line	1	-1	0	0	-1	50	$x - y = 0$	$-y + 50 = 0$
ray	0	0	49	1	0	0	$49 \geq 0$	$x \geq 0$
ray	0	0	1	0	0	0	$1 \geq 0$	$0 \geq 0$
ray	1	0	-50	0	0	0	$x - 50 \geq 0$	$0 \geq 0$
ray	0	0	1	0	0	1	$1 \geq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

(b) generators and their invariants

Fig. 4: Example of a clause in the DNF with its generators and invariants

"point" means a single vector, "ray" means a vector that can be scaled by an arbitrary positive value, and "line" means a vector that can be scaled by any positive or negative value. When putting the generators back to invariants, we obtain the invariants shown in the right part of Figure 4b. The resulting disjunctive invariant is the disjunction of invariants at all locations over an ATS Γ, corresponding to the invariants required on different loop paths.

Invariant Propagation. In the preceding exposition, the computation of conjunctive affine invariants adheres to existing approaches [69,55]. However, the strategies employed to resolve invariants at each location across the entire ATS result in a significant decline in computational efficiency. To address this limitation, we introduce a propagation technique predicated on existing invariant computation results:



(a) $DG(\Gamma)$ (b) $DG(\Gamma)$ After Elimination (c) $DG(\Gamma)$ After Propagation

Fig. 5: Procedure of Invariant Propagation for example in Figure 2

Consider the affine transition system Γ in Figure 2b. Its underlying directed graph $DG(\Gamma)$ is given in Figure 5a. We first compute the invariant $\eta(\ell_2) := y = 50 \wedge 0 \leq x \leq 49$ at the initial location ℓ_2 as above. Then, we can eliminate all transitions pointing to ℓ_2 (the correctness of which is demonstrated in the following section), obtaining the graph in Figure 5b. Notably, there exists a topological order, where ℓ_2 precedes ℓ_1. Thus, we propagate the invariants at ℓ_2 along the transition τ_4's transition guard ρ to establish the initial condition θ of the ATS over ℓ_1 in Figure 5c, which is derived from removing ℓ_2 after propagation. Finally, by solving the invariants for the simplified ATS instead of the original ATS, we obtain the affine invariant $\eta(\ell_1) = (x = y \wedge 50 \leq x \leq 99)$. Note that our invariant propagation is different from abstract interpretation, see Section 6 for details.

Moreover, $\eta(\ell_1)$ corresponds to the invariant within the loop, specifically representing the invariant at the first **case**. For the exit state $x = 100$, the disjunctive invariant generated within the loop, in conjunction with the loop exit condition $\neg G$, derives the program state $x = 100$ outside the loop through stepwise deduction.

4 Algorithmic Details in Our Approach

Below we present our approach for generating affine disjunctive invariants over affine programs. We first illustrate our control flow transformation for non-nested loops, then our invariant propagation to reduce invariant computation, and finally the resolution of the infeasible implication and the extension to nested loops.

4.1 Control Flow Transformation

We fix the set of program variables in a loop as $X = \{x_1, \ldots, x_n\}$ and identify it as the set of variables in the ATS to be derived from the loop. We consider the canonical form of a non-nested affine while loop as in Figure 6 similar to [45], where we have:

- The column vector $\mathbf{x} = (x_1, \ldots, x_n)^{\mathrm{T}}$ represents the vector of program variables, and G is a disjunction of affine assertions that serves as the loop condition.
- 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 matrix (resp. n-dimensional column vector) that specifies the affine update under the function $\mathbf{F}_i$ in the conditional branch ϕ_i. 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 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 has no effect or the **break** statement that exits.

```
while $(G)$ {
    case $\phi_1$:   $\mathbf{x} := \mathbf{F}_1(\mathbf{x})\,;\delta_1\,;$
              ⋮
    case $\phi_m$:   $\mathbf{x} := \mathbf{F}_m(\mathbf{x})\,;\delta_m\,;$
}
```

Fig. 6: The canonical form of a non-nested affine while loop

Any non-nested affine while loop with a break statement can be transformed into the canonical form in Figure 6 by recursively examining the substructures of the loop body. A detailed recursive transformation process is provided in our extended version [48]. Note that although the transformation into our canonical form may cause exponential growth 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, such as τ_2 in Figure 2).

Moreover, such a canonical form is often necessary to capture precise disjunctive information in a while loop. Each case corresponds not merely to an individual loop path but also encapsulates the set of states at the entry of a loop. By finely partitioning the incoming program states according to branching conditions and formulating constraints among these states, we endeavor to precisely characterize the dynamics of internal state transitions within loops exhibiting multi-phase behavior. Below we demonstrate our control flow transformation that transforms the canonical form into an ATS.

Formally, the ATS $\mathit{\Gamma}_W$ for a loop W in our canonical form is given as follows:

- The set of locations is $\{\ell_1, \dots, \ell_m, \ell_e\}$, where each ℓ_i $(1 \le i \le m)$ corresponds to i-th case in the canonical form and ℓ_e is the termination location of the loop.
- For each $1 \le i \le m$ such that $\delta_i = \textbf{break}$, we have the transition (we denote $\mathbf{x}' := (x'_1, \dots, x'_n)^{\mathrm{T}}$)

$$\tau_i = (\ell_i, \ell_e, G \wedge \phi_i \wedge \mathbf{x}' = \mathbf{F}_i(\mathbf{x}))$$

  that specifies the one-step jump from ℓ_i to the termination location ℓ_e.
- For each $1 \le i, j \le m$ such that $\delta_i \neq \textbf{break}$, 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 jump from ℓ_i in the current loop iteration to ℓ_j in the next loop iteration.
- For each $1 \le i \le m$ such that $\delta_i \neq \textbf{break}$, 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 jump from ℓ_i to the termination location ℓ_e.

After the transformation, we remove transitions with an unsatisfiable guard condition to reduce the size of the derived ATS. The transformation for our running example has been given in Figure 2. The transformed ATS $\mathit{\Gamma}_W$ enables us to apply existing approaches [69,55] to generate invariants at the locations of the ATS $\mathit{\Gamma}_W$. Finally, recall that the overall disjunctive invariant for the ATS $\mathit{\Gamma}_W$ is the disjunction of the invariants at all the locations.

As for the previous approaches [69,55] that only set locations at the initial location of the loop, our control flow transformation has different locations corresponding to different paths of the original loop, thereby achieving fine-grained piecewise invariants. Moreover, contrary to the granular program translations employed in traditional software model checking, which is similar to the control flow transformation, our motivation is mainly to integrate this transformation with Farkas' Lemma. By analyzing the transition patterns among internal loop paths, we aim to more effectively capture the phase-specific characteristics of multi-stage programs.

As demonstrated by our experiments, considering transitions between any pair of paths (ℓ_i, ℓ_j), rather than partitioning the loop into more complex path regular expressions (using regular expressions to abstract loop behaviors over multiple iterations) allows us to maintain a balance between the precision of invariant generated and the efficiency of constraint solving. To further improve the efficiency, we have designed the invariant propagation algorithm shown below for these directed graphs constructed from paths.

4.2 Invariant Propagation

In the computation of invariants, previous approaches [69,55] require to generate the invariants at all the locations of an ATS. As invariant computation is usually expensive, it is important to explore optimizations that avoid redundant computations. In this section, we propose a novel invariant propagation technique that is applicable to any directed graph of an ATS and achieves maximal efficiency in specific graph structures such as directed cycles. Below we demonstrate the procedure of invariant propagation via Algorithm 1.

The algorithm consists of the following steps: First, we initialize the assertion map and use the classical Tarjan's algorithm [81] to compute a list of strongly connected components (SCCs) in the directed graph $DG(\Gamma)$ (lines 1-2). Depending on whether the graph is decomposable, i.e., whether the size of the SCC list is more than one, we consider two cases:

(i) For a directed graph that can be decomposed into multiple SCCs, we start from the entry SCC and traverse the list of SCCs in breadth-first order, computing invariants for each SCC recursively and integrating them into the final assertion map η (lines 3-19).

(ii) For a directed graph that is a single SCC, we compute the initial invariants at the starting location using the previous method [55], then eliminate the start location ℓ^*, traverse each edge originating from ℓ^* , propagate the invariants to the remaining sub-graph, and disjunctively merge the returned inductive assertion mappings to produce the final disjunctive invariant (lines 20-26).

We formally define the specific functions involved in the algorithm as follows:

1. **Merge**(η_1, η_2) (line 16 and line 24). We extend η to a mapping from the set of locations L to disjunction of affine assertions, specifically representing affine inequalities in DNF. The Merge function is thereby defined as a new mapping such that, for any location $\ell \in L$:

$$\text{Merge}(\eta_1, \eta_2)(\ell) = \eta_1(\ell) \vee \eta_2(\ell)$$

Algorithm 1 *InvProp*(Γ,$DG(\Gamma)$,ℓ^*)

```
Require: $\Gamma$ — ATS, $DG(\Gamma)$ — directed graph of $\Gamma$, $\ell^*$ — initial location of $\Gamma$.
Ensure: $\eta$ — an inductive assertion map for $\Gamma$.
 1: Init assertion map $\eta$ for $\Gamma$ .
 2: SCCs $\leftarrow$ Tarjan($DG(\Gamma), \Gamma$)            ▷ Find all SCCs in the directed graph
 3: if Len(SCCs) $\neq 1$ then
 4:    id $\leftarrow$ FindSCC($\ell^*$, SCCs)                 ▷ Find the SCC containing $\ell^*$
 5:    stack.push(id, $\ell^*$)
 6:    while ¬stack.isEmpty() do
 7:       (cur, $\ell_s$) $\leftarrow$ stack.pop()          ▷ $\ell_s$ is the initial location of current SCC
 8:       $\Gamma_s \leftarrow$ SCCs[cur]                 ▷ cur is the index of current traversed SCC
 9:       $\eta_s \leftarrow$ InvProp($\Gamma_s, DG(\Gamma_s), \ell_s$)                 ▷ Process single SCC
10:       for each transition $\tau$ directed from $\ell_s$ to $\ell_t$ do
11:          next $\leftarrow$ FindSCC($\ell_t$, SCCs)
12:          if next $\neq$ cur then
13:             stack.push(next, $\ell_t$)               ▷ Traverse SCCs in BFS order
14:          end if
15:       end for
16:       $\eta \leftarrow$ Merge($\eta, \eta_s$)                ▷ Combine assertion maps disjunctively
17:    end while
18:    return $\eta$
19: end if
20: $\eta \leftarrow$ InitInv($\Gamma, \ell^*$)              ▷ Compute invariant only in initial location
21: $\Gamma_s \leftarrow$ Project($\Gamma, \ell^*$)                 ▷ Derive sub-ATS $\Gamma_s$ by removing $l^*$
22: for each transition $\tau$ directed from $\ell^*$ to $\ell_t$ do
23:    $\eta_s \leftarrow$ InvProp($\Gamma_s, DG(\Gamma_s), \ell_t$)
24:    $\eta \leftarrow$ Merge($\eta, \eta_s$)
25: end for
26: return $\eta$
```

2. **Project**(Γ, ℓ^*) (line 21). Considering the directed graph DG(Γ) corresponding to the ATS Γ, we remove all edges associated with the node ℓ^*, as well as the node itself. The derived ATS corresponding to the resulting sub-graph is denoted by Project(Γ, ℓ^*).

Note that in Algorithm 1, we omitted the initial condition θ and the propagation effect along the transitions. At each point of our algorithm (line 10 and line 22) that tackles a transition $\langle \ell_s, \ell_t, \rho \rangle$, the propagation effect is computed as the post image of the conjunction of the invariant on ℓ_s and the guard ρ via polyhedral projection onto the primed variables X' and serves as the initial condition θ of the new ATS including ℓ_t.

Example 1. Recall the example in Section 3, specifically Figure 5. Here, Γ is an indivisible SCC. After computing the invariant $\eta(\ell_2)$ of the ATS Γ at the initial location ℓ_2, we consider all transitions (i.e. $\{\tau_4\}$) starting from the initial location ℓ_2, as depicted in the figure. Then, we propagate the invariant through the transition τ_4 to ℓ_1. After project to obtain the remaining sub-ATS Γ_{sub}, composed of ℓ_1 and its self-loop transition, we recursively compute this indivisible SCC to obtain the complete inductive assertion map. □

Our invariant propagation technique applies to all ATS. The main advantage to incorporate this technique is that it allows the generation of invariants only at the initial locations of (sub-)SCCs, thus avoiding the generation of the invariants at all locations as adopted in [69,55]. In the case that the directed graph of the input ATS is a cycle, our invariant propagation reaches the highest efficiency that generates the invariant only at the initial location of the cycle and derives invariants at other locations of the cycle by propagation, since the cycle has an explicit topological order after the removal of the initial location. This advantage becomes more prominent in loops with a non-neglectable amount of conditional branches. The soundness of the invariant propagation is given in the following theorem.

Theorem 2. *The assertion maps generated by the invariant propagation algorithm are inductive.*

Proof. We prove by induction on the number k of locations in the input ATS Γ that the assertion map obtained by our invariant propagation algorithm for the ATS Γ is inductive.

We first consider the base case, i.e., $k = 1$. In this case, $DG(\Gamma)$ has only one location, which is obviously indivisible. Here, the function **InitInv**(), previously mentioned as applying Farkas' Lemma for conjunctive invariant computation, is called. Therefore, the resulting assertion map is inductive, and its correctness is guaranteed by the prior results.

Assuming that the case when the size of Γ equals k holds, we prove that it holds for Γ of size $k + 1$. For an ATS Γ with $k + 1$ locations, if it is divisible, it can be decomposed into several sub-SCCs Γ_{sub} with sizes less than or equal to k. After the call to function **InvProp**() at Line 8, we obtain an inductive assertion mapping by the inductive condition. The **Merge**() function does not affect the inductive condition of the combined mapping. On the other hand, if it is indivisible, then our approach computes the invariant at its initial location and, after projecting away the initial location ℓ^*, obtains a sub-ATS Γ_{sub} of size k. Similarly, the recursive call to invariant propagation at Line 20 and merging the returned results always yields an inductive assertion map by the inductive condition. □

4.3 Other Optimizations

Loop Summary. To address more general control flow, such as nested loops, we use the standard method of loop summary to express the input-output relationship of the inner loops (while adding fresh variables for input values) to handle nested loops, as described in our extended version [48].

Infeasible Implication. In the previous results [69,55], the infeasible implication is not handled in their prototype. Recall the infeasible implication corresponding to $\eta(\ell) \wedge \rho \models -1 \geq 0$ illustrated in Figure 3b. To fully address this issue, we can simply set $\mu = 1$ in Figure 3b so that the nonlinear multiplier μ is eliminated. The correctness is given by the following theorem.

Theorem 3. *Let Γ be an ATS. For any AAM η that fulfills the initial and consecution conditions derived from the ATS Γ with the original constraints for the infeasible implication as in each consecution tabular of Figure 3b (aimed at $-1 >= 0$) with each μ in an infeasible implication instantiated as k for some $k > 0$, it is equivalent to setting all μ's to 1 while preserving the constraints of infeasible implication.*

We present our proof in our extended version [48]. The main idea of the proof is that, for the infeasible implication case, by scaling each λ_i other than μ, the consecution tabular used to generate polyhedra is transformed into an equivalent tabular with scaled lambda variables λ'_i. so that it suffices to choose the multiplier μ to be 1.

5 Experimental Evaluation

In this section, we present the evaluation of the implementation (referred to as DInvG) of our approach to generate disjunctive affine invariants. We focus on the following two questions (**RQ1** and **RQ2**).

- **RQ1:** How competitive is DInvG when compared with other approaches?
- **RQ2:** How effective does invariant propagation enhance our approach?

5.1 Experimental Setup

Implementation. We implement our approach (including the algorithmic techniques in Section 4) as a prototype DInvG, dividing the implementation into front-end and back-end. The front-end utilizes Clang Static Analyzer [18] to extract and transform C programs, processing programs into the format required by the back-end. The back-end is an extension of StInG [79] written in C++ and uses PPL 1.2 [6] for polyhedra manipulation (e.g., projection, generator computation, etc.), which generates invariants and propagate them to obtain a disjuntive invariant as the loop invariant.

Environment. All experiments are conducted on a machine equipped with a 12th-generation Intel(R) Core(TM) i7-12800HX CPU, 16 cores, 2304 MHz, 9.5GB RAM, running Ubuntu 20.04 (LTS). Following the competition settings of SV-COMP, for studies **RQ1** and **RQ2**, we impose a time limit of 900s.

Benchmarks. We have a total of 114 affine programs, 38.6% of which have disjunctive features, sourced from: 1) 105 benchmarks from the SV-COMP, ReachSafety-Loop track. We excluded those with arrays, pointers, and other non-numeric features, those with modulus, division, polynomial, and other non-linear operations. 2) 9 benchmarks from the recent paper [9], which include complex nested loops and examples with disjunctive features.

Methodology. In **RQ1**, we compare DInvG utilizing invariant propagation techniques with several state-of-the-art software verifiers:

- Veriabs [28] is a state-of-the-art software verifier that is an integration of various strategies such as fuzz testing, k-induction, loop shrinking, loop pruning, full-program induction, explicit state model checking and other invariant generation techniques, which is capable to deal with programs with disjunctive features.
- CPAChecker [25] is a well-developed software verifier that is based on bounded model checking and interpolation and has a comprehensive ability to verify various kinds of properties.
- OOPSLA23 [82] is a recent recurrence analysis tool that handles only loops with the ultimate strict alternation pattern that eventually the loop will alternate between different modes periodically and performs good on such class of programs, which thus excels in the verification of disjunctive programs with alternating modes.
- DIG [59] is an invariant generation tool considering disjunctive features in programs and utilizes front-end CIVL [74] to obtain symbolic execution traces. It employs dynamic analysis along with efficient algorithms from algebra and geometry to solve numerical invariant templates, thereby generating numerical invariants at any position within a program, which is capable of extensively handling the programs with array, nonlinear, linear and disjunctive features.
- IKOS with *Polyset* domain from PPLite [11,8] is a classic abstract interpretation framework with various interface supports. The *Polyset* abstract domain is an efficient implementation of the powerset of polyhedra and serves as an alternative to the trace partitioning strategy implemented in Astree [23].

In **RQ2**, we focus on comparing the impact of the invariant propagation technique on the time efficiency. By contrasting the tool's performance when calculating invariants for each location individually against using invariant propagation, we analyze the role of invariant propagation.

5.2 Tool Comparison (RQ1)

Our work primarily focuses on the generation of disjunctive invariants, whereas tools like CPAChecker and Veriabs are specifically designed as bug finders for verifying assertions. However, by integrating the PPL library [6] and Z3 [89], we use the generated invariants to verify the correctness of assertions and demonstrate the precision of the invariants generated by DInvG.

The complete comparison results of DInvG with other tools are presented in Table 1. In the table, *Source* indicates the source category of the benchmark. The term *#Ver.* represents the number of examples correctly verified by the

Benchmark		DInvG			Veriabs			CPAChecker		
Source	#Num	#Ver.	#Unk.	Time (s)	#Ver.	#Unk.	Time (s)	#Ver.	#Unk.	Time (s)
loop-invariants	5	4	1	0.47	5	0	153.31	4	1	1001.49
loop-new	2	2	0	0.11	0	2	959.74	0	2	1807.20
loop-invgen	5	4	1	0.41	5	0	160.51	0	5	4518.19
loops-crafted-1	25	20	5	4.84	25	0	4010.55	0	25	22607.55
loop-simple	2	1	1	2	1	1	944.69	1	1	919.03
loop-zilu	26	26	0	0.77	25	1	1064.60	26	0	307.18
loops	18	15	3	3.26	17	1	536.33	17	1	1123.35
loop-lit	10	10	0	22.22	10	0	280.87	5	5	5655.33
loop-acceleration	10	9	1	0.32	9	1	493.13	9	1	1030.78
loop-crafted	2	2	0	0.09	2	0	49.59	2	0	27.97
[9]	9	8	1	2.12	9	0	286.24	4	5	4576.98
Total	**114**	101	13	**34.65**	108	6	8939.56	68	46	43573.42

Benchmark		OOPSLA23			DIG			IKOS + PPLite		
Source	#Num	#Ver.	#Unk.	Time (s)	#Ver.	#Unk.	Time (s)	#Ver.	#Unk.	Time (s)
loop-invariants	5	1	4	14.09	0	5	2344.12	3	2	0.88
loop-new	2	0	2	5.83	0	2	241.68	1	1	988.06
loop-invgen	5	4	1	14.78	1	4	264.34	5	0	1.03
loops-crafted-1	25	22	3	82.03	10	15	6030.28	0	25	8270.92
loop-simple	2	0	2	5.82	0	2	493.24	2	0	10.01
loop-zilu	26	0	26	68.24	19	7	6878.38	17	9	1827.31
loops	18	4	14	48.36	3	15	5241.99	7	11	909.04
loop-lit	10	7	3	29.59	3	7	2024.29	5	5	3993.86
loop-acceleration	10	8	1	27.48	3	7	2263.40	6	4	1.66
loop-crafted	2	2	0	5.63	0	2	503.76	2	0	0.33
[9]	9	6	3	28.98	1	8	497.92	8	1	2.04
Total	**114**	55	59	330.83	40	74	26783.40	56	58	16005.15

Table 1: Comparisons Over 114 Benchmarks

verifier, and *#Unk.* (unknown) mainly arises from the following situations: a) The front-end fails to parse correctly, resulting in program crashes. b) Returns **Unknown**. c) Timeouts. For the benchmarks from [9], which do not contain assertions to be verified, we modify the invariants generated by our DInvG as assertions and test them over the other tools to obtain results.

From the table, it is evident that DInvG typically requires less than 0.3 seconds on average for verification, and its overall verification accuracy is very close to that of the SV-COMP 2023 Reachability track winner Veriabs, while significantly outperforming Veriabs in terms of time efficiency by 10X to 1000X. This is mainly because Veriabs employs a rich strategy to assist verification, granting it a stronger verification capability but also requiring more time for most examples. CPAChecker experienced a broad range of timeouts in examples with complex loops that could not be verified within a finite unfolding of loops. This is due to the intrinsic limitations of its bounded model checking approach, and its loop unwinding strategy also results in verification times on the dataset that significantly exceed those of other tools.

Despite the fact that the tool from [82] has the second fewest number of verified benchmarks, it outperforms other tools in examples suitable for recurrence analysis. For DIG, we employ it to generate loop invariants and post conditions,

Benchmark		DInvG					
		No PPG			PPG		
Source	#Num	#Ver.	#Unk.	Time (s)	#Ver.	#Unk.	Time (s)
SV-COMP	105	91	14	1825.53	93	12	32.53
paper	9	8	1	10.76	8	1	2.12

Table 2: Experiment for Invariant Propagation

and use Z3 [89] prover to verify the assertion. Nevertheless, the frontend of DIG necessitates CIVL's reliance on extracting symbolic execution paths from the program. When processing loops, it similarly depends on loop unrolling, and if it cannot fully unroll loops within a small bound, it determines that locations after the loop are unreachable. Consequently, it exhibits issues analogous to those of CPAChecker. Additionally, for some randomly assigned variables in SV-COMP, DIG lacks a suitable modeling. We have already reported several bugs via issues on GitHub. As a classical framework for abstract interpretation, IKOS with PPLite did not deliver optimal verification outcomes on the dataset. In some straightforward nested loops and more extensive loop iterations, it either failed to converge to a fixed point, or the precision of the invariants obtained upon convergence was insufficient to verify assertions, thereby causing timeouts or unknown in certain instances.

In summary, we conclude that DInvG significantly outperforms other tools such as Veriabs in time efficiency for affine numerical programs, while its verification capability is not inferior to the SV-COMP winner Veriabs. We also conducted an in-depth analysis of the cases where our DInvG returns **Unknown**. The primary reasons for the issues include: a) the absence of type range constraints at the front end, b) reliance on modular arithmetic, c) the need for more complex loop generalizations, d) exceeding the computational precision of the PPL library, and e) exponential arithmetic that surpasses the modeling capabilities of linear templates. 7-8 of these cases could be further solved by optimizing implementations. In the verifiable cases, the preliminary implementation of DInvG has already far surpassed existing methods in efficiency.

5.3 Ablation Study in Invariant Propagation (RQ2)

In this section, we conduct an ablation study to evaluate the performance of the invariant propagation technique within DInvG. In Table 2, we present the overall results, where we can clearly observe that the use of invariant propagation leads to a 5X-50X improvement in time efficiency.

More specifically, through the scatter plot in Figure 7, we compared the time performance of individual examples before and after the application of invariant propagation techniques. In some cases, invariant propagation led to significant efficiency improvements (10X-1000X). This is due to the fact that for more complex programs, the size of the ATS Γ is larger, and applying invariant

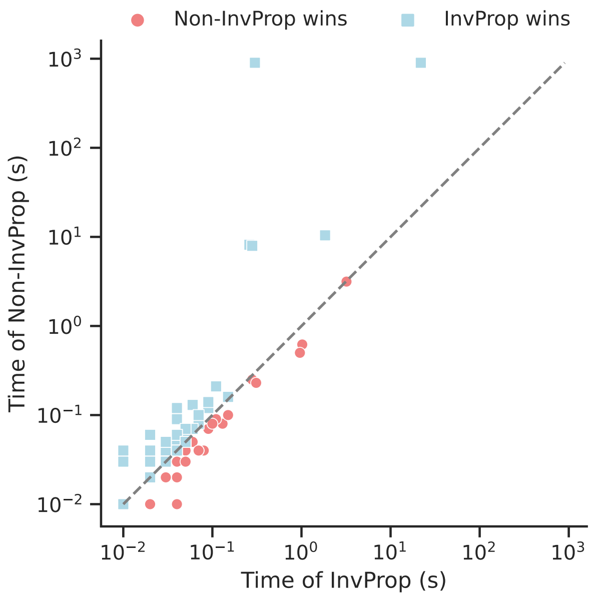


Fig. 7: Comparison for Invariant Propagation

propagation techniques on this basis can maximize performance optimization. Since the tool itself performs efficiently in most examples, the optimization brought by this technique is not apparent in those cases in the graph where the time is below 0.1 seconds. As the propagation itself, including the projection of sub-ATS, incurs a certain time cost, which dilutes the time optimization brought about by invariant propagation.

In conclusion, invariant propagation significantly enhances the tool's scalability and yields superior optimization results for complex examples. This also reveals that, within our constraint-solving methodology, the cost of computing invariants at any given location constitutes the principal computational bottleneck. By reducing the number of locations that need to be computed and leveraging prior results to avoid redundant polyhedral operations, we can effectively enhance efficiency.

5.4 Caveat to Correctness

This section elucidates configurations that may induce subtle deviations from real-world programs or alternative models during the empirical evaluation of our tool.

- In our current experimental setup, we have not accounted for the behavior of machine integers during overflow conditions. Consequently, our verification process is confined to affine programs that do not encounter overflow errors.
- Within the context of control flow transformations, we introduce uncertainty into conditional statements by adhering to the SV-COMP guidelines. This

is achieved by replacing branch conditions with functions that return random Boolean values, thereby emulating the semantics of non-deterministic branches. Nonetheless, we have yet to effectively model uncertainty in variable coefficients, specifically affine inequalities with coefficients represented as intervals.

6 Related Works

Our methodology enhances conjunctive affine invariants by integrating optimizations from prior research [19,69,55,45] and utilizing control flow transformation techniques to extend them to disjunctive forms. A principal contribution of this paper is the mitigation of computational inefficiencies resulting from the exponential state space expansion associated with disjunctive extensions, achieved through invariant propagation. Consequently, this approach distinguishes our work from existing studies. The work [39] generates disjunctive invariants by predefining disjunctive templates, heuristically selecting physical cut points (while we select abstract locations from loop paths) and transforming the quadratic constraints from Farkas' Lemma into SAT solving. Other approaches for conjunctive affine invariant generation include [61,40]. These approaches propose completely different techniques, and thus are orthogonal to our approach.

Polynomial invariant generation [47,87,15,43,66,21,1,53,17,60,44,68] has been widely investigated. Most of these approaches consider conjunctive polynomial invariants only. Compared with conjunctive polynomial invariants, disjunctive affine invariants capture the precise feature of phase and mode changes in affine loops, and therefore are more precise.

The works [85,54] are based on path dependency automata, requiring precise estimates of the number of iterations in loops, which limits their analysis to programs with regular alternation and inductive variables (computable general terms). The work [72] studies the detection of multiphase disjunctive invariants. Multiphase invariants are a special case of our control flow transformation since each phase in a multiphase loop cannot go back to previous phases, while in our control flow transformation, locations can go back and forth via transitions. Thus, we have a wider class of disjunctive invariants as compared with [72].

Our control flow transformation is related to control flow refinement [7,38,27,75] in the literature. These approaches mostly focus on representing the control flow of multiple loop iterations as regular expressions and refine these regular expressions by various approaches such as abstract domains, simulation relation and even invariant generation to reduce infeasible paths. Our control flow transformation considers the loop body within a single loop iteration, and is dedicated to the application of Farkas' Lemma. Thus, our control flow transformation has a different focus compared with these results. Moreover, the use of Farkas' Lemma can circumvent the issue that finer control flow may not always lead to finer analysis in control flow refinement [27].

Our invariant propagation is related to abstract interpretation [24,5,76,37,9,42]. The main difference is that it propagates the *already-computed* invariants (via

Farkas' Lemma) to yet not computed locations as much as possible to minimize the invariant generation computation, while abstract interpretation usually requires an involved fixed-point iteration to *compute* invariants.

Recurrence analysis [33,49,50] works well over programs with specific structure that ensures closed form solutions. For example, the most related recurrence analysis approach [82] (that also targets disjunctive invariants) solves the exact invariant over the class of loops with (ultimate) strict alternation between different modes. Compared with recurrence analysis, our approach does not require specific program structure to ensure closed form solution, but is less precise over programs that can be solved exactly by recurrence analysis.

Finally, 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 several approaches (such as IC3 [77], CLN2INV [67], [64]) usually requires a goal property. Note that the invariant generation without a given goal property is a classical setting (see e.g. [19,24]), and has applications in loop summary and probabilistic program verification (see e.g. [13,83]).

LLM-based invariant generation methods [84] performs poorly on certain complex programs exhibiting disjunctive features. Those large-scale models have been unable to precisely comprehend the disjunctive properties inherent in these programs, and the invariants they produce often necessitate iterative interaction with Frama-C until an invariant that can be successfully verified by Frama-C is generated.

7 Conclusion

In this work, we propose a novel approach to generate affine disjunctive invariants over affine loops. Our novelty lies in combining a control flow transformation to extract the interleaving relationships between loop paths and employing Farkas' Lemma to solve the disjunctive invariants of loops. Additionally, we apply invariant propagation techniques to mitigate the computational costs of exponential explosion. A thorough resolution of the infeasible implication in the application of Farkas' Lemma and an extension to nested loops through loop summary are proposed as optimizations for practical program verification. Experimental results show that our approach is competitive with state-of-the-art software verifiers in affine disjunctive invariant generation over affine loops.

Acknowledgments. We thank anonymous reviewers for constructive comments. This work is supported by the National Key R&D Program of China (No. 2022YFA1005101) and the National Natural Science Foundation of China (Nos. 61872232, 62172271).

References

1. Adjé, A., Garoche, P., Magron, V.: Property-based polynomial invariant generation using sums-of-squares optimization. In: SAS. LNCS, vol. 9291, pp. 235–251. Springer, [S.l.] (2015)
2. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A framework for abstraction- and interpolation-based software verification. In: CAV. LNCS, vol. 7358, pp. 672–678. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_48, https://doi.org/10.1007/978-3-642-31424-7_48
3. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: SAS. LNCS, vol. 6337, pp. 117–133. Springer (2010). https://doi.org/10.1007/978-3-642-15769-1_8, https://doi.org/10.1007/978-3-642-15769-1_8
4. Asadi, A., Chatterjee, K., Fu, H., Goharshady, A.K., Mahdavi, M.: Polynomial reachability witnesses via stellensätze. In: PLDI. pp. 772–787. ACM (2021). https://doi.org/10.1145/3453483.3454076, https://doi.org/10.1145/3453483.3454076
5. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise widening operators for convex polyhedra. In: Cousot, R. (ed.) Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2694, pp. 337–354. Springer (2003). https://doi.org/10.1007/3-540-44898-5_19, https://doi.org/10.1007/3-540-44898-5_19
6. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the parma polyhedra library. In: SAS. Lecture Notes in Computer Science, vol. 2477, pp. 213–229. Springer (2002). https://doi.org/10.1007/3-540-45789-5_17, https://doi.org/10.1007/3-540-45789-5_17
7. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: Chakraborty, S., Halbwachs, N. (eds.) Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009. pp. 49–58. ACM (2009). https://doi.org/10.1145/1629335.1629343, https://doi.org/10.1145/1629335.1629343
8. Becchi, A., Zaffanella, E.: Pplite: zero-overhead encoding of nnc polyhedra. Information and Computation **275**, 104620 (2020)
9. Boutonnet, R., Halbwachs, N.: Disjunctive relational abstract interpretation for interprocedural program analysis. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings. LNCS, vol. 11388, pp. 136–159. Springer (2019). https://doi.org/10.1007/978-3-030-11245-5_7, https://doi.org/10.1007/978-3-030-11245-5_7
10. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: CAV. LNCS, vol. 3576, pp. 491–504. Springer (2005). https://doi.org/10.1007/11513988_48, https://doi.org/10.1007/11513988_48
11. Brat, G., Navas, J.A., Shi, N., Venet, A.: Ikos: A framework for static analysis based on abstract interpretation. In: Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings 12. pp. 271–277. Springer (2014)
12. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700, https://doi.org/10.1145/2049697.2049700

13. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: CAV. LNCS, vol. 8044, pp. 511–526. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_34, https://doi.org/10.1007/978-3-642-39799-8_34
14. Chatterjee, K., Fu, H., Goharshady, A.K.: Non-polynomial worst-case analysis of recursive programs. ACM Trans. Program. Lang. Syst. **41**(4), 20:1–20:52 (2019). https://doi.org/10.1145/3339984, https://doi.org/10.1145/3339984
15. Chatterjee, K., Fu, H., Goharshady, A.K., Goharshady, E.K.: Polynomial invariant generation for non-deterministic recursive programs. In: PLDI. pp. 672–687. ACM (2020). https://doi.org/10.1145/3385412.3385969, https://doi.org/10.1145/3385412.3385969
16. Chen, Y., Xia, B., Yang, L., Zhan, N., Zhou, C.: Discovering non-linear ranking functions by solving semi-algebraic systems. In: ICTAC. LNCS, vol. 4711, pp. 34–49. Springer (2007). https://doi.org/10.1007/978-3-540-75292-9_3, https://doi.org/10.1007/978-3-540-75292-9_3
17. Chen, Y., Hong, C., Wang, B., Zhang, L.: Counterexample-guided polynomial loop invariant generation by lagrange interpolation. In: CAV. LNCS, vol. 9206, pp. 658–674. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_44, https://doi.org/10.1007/978-3-319-21690-4_44
18. Clang static analyzer: A source code analysis tool that finds bugs in c, c++, and objective-c programs. https://clang-analyzer.llvm.org/ (2022)
19. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: CAV. LNCS, vol. 2725, pp. 420–432. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_39, https://doi.org/10.1007/978-3-540-45069-6_39
20. Colón, M., Sipma, H.: Synthesis of linear ranking functions. In: TACAS. LNCS, vol. 2031, pp. 67–81. Springer (2001). https://doi.org/10.1007/3-540-45319-9_6, https://doi.org/10.1007/3-540-45319-9_6
21. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: VMCAI. LNCS, vol. 3385, pp. 1–24. Springer (2005). https://doi.org/10.1007/978-3-540-30579-8_1, https://doi.org/10.1007/978-3-540-30579-8_1
22. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973, https://doi.org/10.1145/512950.512973
23. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astrée analyzer. In: Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 14. pp. 21–30. Springer (2005)
24. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. pp. 84–96. ACM Press (1978). https://doi.org/10.1145/512760.512770, https://doi.org/10.1145/512760.512770
25. Cpachecker: The configurable software-verification platform. https://cpachecker.sosy-lab.org (2022)
26. Csallner, C., Tillmann, N., Smaragdakis, Y.: Dysy: dynamic symbolic execution for invariant inference. In: ICSE. pp. 281–290. ACM (2008). https://doi.org/10.1145/1368088.1368127, https://doi.org/10.1145/1368088.1368127

27. Cyphert, J., Breck, J., Kincaid, Z., Reps, T.W.: Refinement of path expressions for static analysis. Proc. ACM Program. Lang. **3**(POPL), 45:1–45:29 (2019). https://doi.org/10.1145/3290358, https://doi.org/10.1145/3290358
28. Darke, P., Agrawal, S., Venkatesh, R.: 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. pp. 458–462. Springer (2021)
29. David, C., Kesseli, P., Kroening, D., Lewis, M.: Danger invariants. In: FM. LNCS, vol. 9995, pp. 182–198 (2016). https://doi.org/10.1007/978-3-319-48989-6_12, https://doi.org/10.1007/978-3-319-48989-6_12
30. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: OOPSLA. pp. 443–456. ACM (2013). https://doi.org/10.1145/2509136.2509511, https://doi.org/10.1145/2509136.2509511
31. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Yahav, E. (ed.) SAS. LNCS, vol. 6887, pp. 351–368. Springer (2011). https://doi.org/10.1007/978-3-642-23702-7_26, https://doi.org/10.1007/978-3-642-23702-7_26
32. Farkas, J.: A fourier-féle mechanikai elv alkalmazásai (Hungarian). Mathematikaiés Természettudományi Értesitö **12**, 457–472 (1894)
33. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: FMCAD. pp. 57–64. IEEE (2015)
34. Gan, T., Xia, B., Xue, B., Zhan, N., Dai, L.: Nonlinear craig interpolant generation. In: CAV. LNCS, vol. 12224, pp. 415–438. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_20, https://doi.org/10.1007/978-3-030-53288-8_20
35. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV. LNCS, vol. 8559, pp. 69–87. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_5, https://doi.org/10.1007/978-3-319-08867-9_5
36. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL. pp. 499–512. ACM (2016). https://doi.org/10.1145/2837614.2837664, https://doi.org/10.1145/2837614.2837664
37. Gopan, D., Reps, T.W.: Guided static analysis. In: Nielson, H.R., Filé, G. (eds.) Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings. LNCS, vol. 4634, pp. 349–365. Springer (2007). https://doi.org/10.1007/978-3-540-74061-2_22, https://doi.org/10.1007/978-3-540-74061-2_22
38. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 375–385. ACM (2009). https://doi.org/10.1145/1542476.1542518, https://doi.org/10.1145/1542476.1542518
39. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI. pp. 281–292. ACM (2008). https://doi.org/10.1145/1375581.1375616, https://doi.org/10.1145/1375581.1375616
40. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: CAV. LNCS, vol. 5643, pp. 634–640. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_48, https://doi.org/10.1007/978-3-642-02658-4_48

41. He, J., Singh, G., Püschel, M., Vechev, M.T.: Learning fast and precise numerical analysis. In: PLDI. pp. 1112–1127. ACM (2020). `https://doi.org/10.1145/3385412.3386016`, `https://doi.org/10.1145/3385412.3386016`
42. Henry, J., Monniaux, D., Moy, M.: PAGAI: A path sensitive static analyser. Electron. Notes Theor. Comput. Sci. **289**, 15–25 (2012). `https://doi.org/10.1016/j.entcs.2012.11.003`, `https://doi.org/10.1016/j.entcs.2012.11.003`
43. Hrushovski, E., Ouaknine, J., Pouly, A., Worrell, J.: Polynomial invariants for affine programs. In: LICS. pp. 530–539. ACM (2018). `https://doi.org/10.1145/3209108.3209142`, `https://doi.org/10.1145/3209108.3209142`
44. Humenberger, A., Jaroschek, M., Kovács, L.: Automated generation of non-linear loop invariants utilizing hypergeometric sequences. In: ISSAC. pp. 221–228. ACM (2017). `https://doi.org/10.1145/3087604.3087623`, `https://doi.org/10.1145/3087604.3087623`
45. Ji, Y., Fu, H., Fang, B., Chen, H.: Affine loop invariant generation via matrix algebra. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 257–281. Springer (2022). `https://doi.org/10.1007/978-3-031-13185-1_13`, `https://doi.org/10.1007/978-3-031-13185-1_13`
46. K., H.G.V., Shoham, S., Gurfinkel, A.: Solving constrained horn clauses modulo algebraic data types and recursive functions. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022). `https://doi.org/10.1145/3498722`, `https://doi.org/10.1145/3498722`
47. Kapur, D.: 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 (2005), `http://drops.dagstuhl.de/opus/volltexte/2006/511`
48. Ke, J., Fu, H., Liu, H., Chen, L., Li, G.: Affine disjunctive invariant generation with farkas' lemma. arXiv preprint arXiv:2307.13318 (2023)
49. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.W.: Compositional recurrence analysis revisited. In: PLDI. pp. 248–262. ACM (2017). `https://doi.org/10.1145/3062341.3062373`, `https://doi.org/10.1145/3062341.3062373`
50. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.W.: Non-linear reasoning for invariant synthesis. Proc. ACM Program. Lang. **2**(POPL), 54:1–54:33 (2018). `https://doi.org/10.1145/3158142`, `https://doi.org/10.1145/3158142`
51. Larraz, D., Rodríguez-Carbonell, E., Rubio, A.: Smt-based array invariant generation. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) 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, pp. 169–188. Springer (2013). `https://doi.org/10.1007/978-3-642-35873-9_12`, `https://doi.org/10.1007/978-3-642-35873-9_12`
52. Le, T.C., Zheng, G., Nguyen, T.: SLING: using dynamic analysis to infer program invariants in separation logic. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 788–801. ACM (2019). `https://doi.org/10.1145/3314221.3314634`, `https://doi.org/10.1145/3314221.3314634`
53. Lin, W., Wu, M., Yang, Z., Zeng, Z.: Proving total correctness and generating preconditions for loop programs via symbolic-numeric computation methods. Frontiers Comput. Sci. **8**(2), 192–202 (2014)

54. Lin, Y., Zhang, Y., Chen, S., Song, F., Xie, X., Li, X., Sun, L.: Inferring loop invariants for multi-path loops. In: International Symposium on Theoretical Aspects of Software Engineering, TASE 2021, Shanghai, China, August 25-27, 2021. pp. 63–70. IEEE (2021). https://doi.org/10.1109/TASE52547.2021.00030, https://doi.org/10.1109/TASE52547.2021.00030
55. Liu, H., Fu, H., Yu, Z., Song, J., Li, G.: Scalable linear invariant generation with Farkas' lemma. Proc. ACM Program. Lang. **6**(OOPSLA2) (oct 2022). https://doi.org/10.1145/3563295, https://doi.org/10.1145/3563295
56. Manna, Z., Pnueli, A.: Temporal verification of reactive systems - safety. Springer (1995)
57. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 413–427. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_31, https://doi.org/10.1007/978-3-540-78800-3_31
58. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: ICSE. pp. 683–693. IEEE Computer Society (2012). https://doi.org/10.1109/ICSE.2012.6227149, https://doi.org/10.1109/ICSE.2012.6227149
59. Nguyen, T., Nguyen, K., Duong, H.: Syminfer: inferring numerical invariants using symbolic states. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. pp. 197–201 (2022)
60. de Oliveira, S., Bensalem, S., Prevosto, V.: Polynomial invariants by linear algebra. In: ATVA. LNCS, vol. 9938, pp. 479–494 (2016). https://doi.org/10.1007/978-3-319-46520-3_30, https://doi.org/10.1007/978-3-319-46520-3_30
61. de Oliveira, S., Bensalem, S., Prevosto, V.: Synthesizing invariants by solving solvable loops. In: ATVA. LNCS, vol. 10482, pp. 327–343. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_22, https://doi.org/10.1007/978-3-319-68167-2_22
62. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: PLDI. pp. 614–630. ACM (2016). https://doi.org/10.1145/2908080.2908118, https://doi.org/10.1145/2908080.2908118
63. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI. LNCS, vol. 2937, pp. 239–251. Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_20, https://doi.org/10.1007/978-3-540-24622-0_20
64. Riley, D., Fedyukovich, G.: Multi-phase invariant synthesis. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 607–619 (2022)
65. Rodríguez-Carbonell, E., Kapur, D.: An abstract interpretation approach for automatic generation of polynomial invariants. In: SAS. LNCS, vol. 3148, pp. 280–295. Springer (2004). https://doi.org/10.1007/978-3-540-27864-1_21, https://doi.org/10.1007/978-3-540-27864-1_21
66. Rodríguez-Carbonell, E., Kapur, D.: Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In: ISSAC. pp. 266–273. ACM (2004). https://doi.org/10.1145/1005285.1005324, https://doi.org/10.1145/1005285.1005324
67. Ryan, G., Wong, J., Yao, J., Gu, R., Jana, S.: 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 (2020), https://openreview.net/forum?id=HJlfuTEtvB

68. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: POPL. pp. 318–329. ACM (2004). https://doi.org/10.1145/964001.964028, https://doi.org/10.1145/964001.964028
69. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: SAS. LNCS, vol. 3148, pp. 53–68. Springer (2004). https://doi.org/10.1007/978-3-540-27864-1_7, https://doi.org/10.1007/978-3-540-27864-1_7
70. Schrijver, A.: Theory of linear and integer programming. Wiley-Interscience series in discrete mathematics and optimization, Wiley (1999)
71. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. Formal Methods Syst. Des. **48**(3), 235–256 (2016). https://doi.org/10.1007/s10703-016-0248-5, https://doi.org/10.1007/s10703-016-0248-5
72. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 703–719. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_57, https://doi.org/10.1007/978-3-642-22110-1_57
73. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: ESOP. LNCS, vol. 7792, pp. 574–592. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_31, https://doi.org/10.1007/978-3-642-37036-6_31
74. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: Civl: the concurrency intermediate verification language. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2015)
75. Silverman, J., Kincaid, Z.: Loop summarization with rational vector addition systems. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 97–115. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_7, https://doi.org/10.1007/978-3-030-25543-5_7
76. Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 46–59. ACM (2017)
77. Somenzi, F., Bradley, A.R.: IC3: where monolithic and incremental meet. In: Bjesse, P., Slobodová, A. (eds.) International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011. pp. 3–8. FMCAD Inc. (2011), http://dl.acm.org/citation.cfm?id=2157657
78. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 223–234. ACM (2009). https://doi.org/10.1145/1542476.1542501, https://doi.org/10.1145/1542476.1542501
79. Sting: Stanford invariant generator. http://theory.stanford.edu/~srirams/Software/sting.html (2006)
80. Software verification competition. https://sv-comp.sosy-lab.org (2023)
81. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM journal on computing **1**(2), 146–160 (1972)

82. Wang, C., Lin, F.: Solving conditional linear recurrences for program verification: The periodic case. In: OOPSLA. ACM (2023), to appear
83. Wang, J., Sun, Y., Fu, H., Chatterjee, K., Goharshady, A.K.: Quantitative analysis of assertion violations in probabilistic programs. In: PLDI. pp. 1171–1186. ACM (2021). https://doi.org/10.1145/3453483.3454102, https://doi.org/10.1145/3453483.3454102
84. Wen, C., Cao, J., Su, J., Xu, Z., Qin, S., He, M., Li, H., Cheung, S.C., Tian, C.: Enchanting program specification synthesis by large language models using static analysis and program verification. In: International Conference on Computer Aided Verification. pp. 302–328. Springer (2024)
85. Xie, X., Chen, B., Liu, Y., Le, W., Li, X.: Proteus: computing disjunctive loop summary via path dependency analysis. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 61–72. ACM (2016). https://doi.org/10.1145/2950290.2950340, https://doi.org/10.1145/2950290.2950340
86. Xu, R., He, F., Wang, B.: Interval counterexamples for loop invariant learning. In: ESEC/FSE. pp. 111–122. ACM (2020). https://doi.org/10.1145/3368089.3409752, https://doi.org/10.1145/3368089.3409752
87. Yang, L., Zhou, C., Zhan, N., Xia, B.: Recent advances in program verification through computer algebra. Frontiers Comput. Sci. China **4**(1), 1–16 (2010). https://doi.org/10.1007/s11704-009-0074-7, https://doi.org/10.1007/s11704-009-0074-7
88. Yao, J., Ryan, G., Wong, J., Jana, S., Gu, R.: Learning nonlinear loop invariants with gated continuous logic networks. In: PLDI. pp. 106–120. ACM (2020). https://doi.org/10.1145/3385412.3385986, https://doi.org/10.1145/3385412.3385986
89. Z3. https://github.com/Z3Prover/z3 (2023)