Science of Computer Programming

Optimization of Farka's Lemma-Based Linear Invariant Generation Using Divide-and-Conquer with Pruning

--Manuscript Draft--

Manuscript Number:	SCICO-D-24-00345
Article Type:	VSI:Special Issue on TASE24 [MGE- Wei-Ngan W. N. Chin]
Keywords:	program verification; invariant generation; constraint-solving; divide-and-conquer; pruning strategy
Manuscript Region of Origin:	CHINA

Abstract:	Formal verification plays a critical role in contemporary computer science, offering mathematically rigorous methods to ensure the correctness, reliability, and security of programs. Loops, due to their complexity and uncertainty, have become a major challenge in program verification. Loop invariants are often employed to abstract the properties of loops within a program, making the automatic generation of such invariants a pivotal challenge. Among the various methods, template-based frameworks grounded in Farkas' Lemma are recognized for their effectiveness in generating tight invariants in the realm of constraint solving. Recent advances have identified the conversion from CNF to DNF as a major bottleneck, leading to a combinatorial explosion. In this study, we introduce an optimized algorithm to address the combinatorial explosion by trading off space for time efficiency. Our approach employs two key strategies, divide-and-conquer, and pruning, to boost speed. First, we apply a divide-and-conquer strategy to decompose a complex problem into smaller, more manageable subproblems that can be solved quickly and in parallel. Second, we intelligently apply a pruning strategy, navigating the depth-first search process to avoid unnecessary checks. These improvements maintain the accuracy and speed up the analysis. We constructed a small dataset to showcase the superiority of our tool, which achieved an average speedup of 9.27x on this dataset. The experiments demonstrate that our method provides significant acceleration while maintaining accuracy and indicate that our approach outperforms the state-of-the-art methods.

Optimization of Farka's Lemma-Based Linear Invariant Generation Using Divide-and-Conquer with Pruning

Ruibang Liu[a], Hongming Liu[a], Guoqiang Li[a,*]

[a]*Shanghai Jiao Tong University, 200240, Shanghai, China*

Abstract

Formal verification plays a critical role in contemporary computer science, offering mathematically rigorous methods to ensure the correctness, reliability, and security of programs. Loops, due to their complexity and uncertainty, have become a major challenge in program verification. Loop invariants are often employed to abstract the properties of loops within a program, making the automatic generation of such invariants a pivotal challenge. Among the various methods, template-based frameworks grounded in Farkas' Lemma are recognized for their effectiveness in generating tight invariants in the realm of constraint solving. Recent advances have identified the conversion from CNF to DNF as a major bottleneck, leading to a combinatorial explosion. In this study, we introduce an optimized algorithm to address the combinatorial explosion by trading off space for time efficiency. Our approach employs two key strategies, divide-and-conquer, and pruning, to boost speed. First, we apply a divide-and-conquer strategy to decompose a complex problem into smaller, more manageable subproblems that can be solved quickly and in

*Corresponding author.
Email address: li.g@sjtu.edu.cn (Guoqiang Li)

parallel. Second, we intelligently apply a pruning strategy, navigating the depth-first search process to avoid unnecessary checks. These improvements maintain the accuracy and speed up the analysis. We constructed a small dataset to showcase the superiority of our tool, which achieved an average speedup of 9.27x on this dataset. The experiments demonstrate that our method provides significant acceleration while maintaining accuracy and indicate that our approach outperforms the state-of-the-art methods.

Keywords: program verification, invariant generation, constraint-solving, divide-and-conquer, pruning strategy

1. Introduction

An assertion at a program location is termed an *invariant* if it consistently holds true for the values taken by program variables whenever that location is reached during execution. Invariants play a pivotal role in program analysis and verification by providing an over-approximation of reachable program states. They are crucial for proving properties such as safety [1], reachability [2] and time-complexity analysis [3] among others. The quality of the invariants is measured by their accuracy, i.e., the amount of over-approximation against the actual set of reachable program states. Generating high-quality invariants has been a significant challenge for decades. Traditionally, invariants are derived through *inductive invariants* [4]. In this domain, two predominant methodologies are employed: abstract interpretation [5, 6] and constraint solving [4, 7]. Abstract interpretation is one of the earliest and most classical methods applied in loop invariant generation, which often seeks to derive loop invariants by extending traditional abstract interpretation frameworks for over-approximations of program semantics. The form and precision of the invariants it generates are influenced by the abstract domain used. However, due to the use of abstract domains that often contain an infinite number of abstract states, its completeness is typically not guaranteed [8]. Even with the use of widening operators to ensure termination, precision is often lost, resulting in the generation of weaker invariants.

Constraint solving-based method usually formulates the loop invariants as mathematical constraints with parameters, which are then solved using algorithms to find proper parameters. To solve the invariant generation problem, constraint-solving-based approaches usually consider the following workflow: initially establish a template with unknown coefficients for the invariant, then collect constraints from the inductive conditions for invariants, and finally resolve the unknown coefficients in the template to obtain the preset invariants. Specifically, in linear loop invariant generation, Farkas' Lemma provides a complete characterization of the inductive condition [4], which was further improved by quantifier elimination [7] and heuristics [9]. The completeness of constraint solving follows from the fact that any inductive invariant can theoretically be derived from the relevant constraints in constraint solving.

We will then describe a simple example to provide an intuitive understanding of the basic constraint-solving-based method via Farkas' Lemma. Figure 1 illustrates a simple incrementing program written in C, containing a loop that increments the variable `i`. Our goal is to compute a linear loop invariant for this given program with a loop. The constraint-solving-based method first proposes a template, which in this case is a linear polynomial template:

$$c_1 x_1 + \cdots + c_n x_n + d \geq 0$$

where $\{c_1, \cdots, c_n\}$ and d are a series of unknown parameters, and $\{x_1, \cdots, x_n\}$ representing the variables appearing in the program. What we need next is to solve for these parameters based on the relationships between the variables in the program. The program needs to be transformed into a Linear Transition System (LinTS). We will temporarily skip the formal description and simply treat it as a tool that characterizes the linear relationships between variables during transitions. Clearly, LinTS can abstract the control flow and data flow of a program, expressing the relationships between variables in the loop as well as the relationships governing variable updates.

```
1 int i = 0;
2 while(i < 10) {
3   i = i + 1;
4 }
```

(a) A simple incrementing program.

$$X = \{i\}, L = \{l_2, l_3\}$$

$$\theta : i = 0, \mathcal{T} = \{\tau_0, \tau_1\}$$

$$\tau_0 = \langle l_2, l_3, (i < 10) \rangle$$

$$\tau_1 = \langle l_3, l_2, (i' = i + 1) \rangle$$

(b) Corresponding LinTS

Figure 1: A simple example and its corresponding LinTS

In the example depicted in Figure 1, the template is clearly $c_1 i + d \geq 0$, and its corresponding LinTS describes the linear relationships within the loop from line 2 to line 3, with the initial state being $i = 0$. There are two types of transitions: one occurs every time the condition $i < 10$ is satisfied and the loop is entered; the other occurs when the loop terminates, where the variable i is incremented. Farka's Lemma is then applied to establish the relationship between the linear system and the parameterized template. We will temporarily skip the specific details of Farka's Lemma, but suffice it to say that it allows for the equivalence transformation of the implication between the linear system and the parameterized template into a nonlinear solving problem with existential quantifiers. Here, this problem will be transformed into:

$$\begin{aligned}
&\exists \lambda_0, \lambda_1. \quad c_{0,1} = \lambda_1 \wedge d_0 = \lambda_0 \geq 0 \\
&\exists \mu, \lambda_0, \lambda_1. \quad \mu c_{2,1} = \lambda_1 \wedge d_3 = \mu d_2 + \lambda_0 + 10\lambda_1 \wedge \lambda_0 \geq 0 \\
&\exists \mu, \lambda_0, \lambda_1. \quad \mu c_{3,1} + \lambda_1 = 0 \wedge \lambda_1 + c_{2,1} = 0 \wedge \mu d_3 + \lambda_0 + \lambda_1 = d_2 \wedge \lambda_0 \geq 0
\end{aligned}$$

Then, solving this problem will yield two expressions: $i \geq 0$ and $-i+10 \geq 0$, which means we can get the invariant $0 \leq i \leq 10$. Noticing the equivalence transformation within this process, we can observe that this method is theoretically complete. However, correspondingly, due to the need to solve first-order logic expressions with quantifiers and nonlinearity, the process of solving these equations incurs significant computational overhead. In other words, compared with other methods, constraint solving has the advantage of a theoretical guarantee of the accuracy of the generated invariants but typically requires higher runtime complexity. Furthermore, recent advance [9] has highlighted the challenge: the conversion from CNF to DNF leads to a combinatorial explosion, especially when generating invariants with Farkas' Lemma in constraint solving.

In this work, we focus on automatically generating linear inductive invariants with Farkas' Lemma, which is important for both academic research and practical applications. We believe that the constraint-solving-based method using Farkas' Lemma, due to its completeness, has distinct advantages in linear invariant generation. However, we also acknowledge that its significant performance overhead is indeed its main issue at present. We observe that the performance during the CNF-to-DNF conversion process can be improved in two main ways: first, by attempting to decompose and decouple the problem to enable parallelization; second, during the expansion process, some calculations are redundant and unnecessary. To address this issue, we propose an optimized approach aimed at addressing the bottleneck arising from the combinatorial explosion by trading off space for time efficiency. The approach is developed on the abstract model of linear transition systems that capture general affine updates with affine guards between program locations so that it is applicable to general affine programs. The strategies we contribute are as follows:

- First, a divide-and-conquer technique is introduced, which decomposes a complex problem into smaller, more manageable subproblems that can be solved quickly and in parallel, greatly cutting down on the time needed for the entire process.

- Second, the pruning strategy is utilized within two intelligent ways to navigate through the depth-first search process that helps avoid unnecessary checks, named path-recording multi-step-backtracking pruning and full-scope-predictive forward-checking pruning.

This paper is an extension of a previously published conference paper [10].

The main difference lies in several places in this version. We optimized the description of the Introduction section and offered an extra example to better explain our work. Then, we modified parts of the Preliminaries and Overview sections to more clearly differentiate our approach from previous work. Additionally, in the experimental section, we further analyzed the research questions and conducted a more detailed discussion and analysis to highlight the advantages of our tool. Finally, we expanded the Related Work section to include more relevant studies and analyzed the similarities, differences, and relative strengths and weaknesses between our approach and theirs.

Paper Organization: The rest of the paper is organized as follows: Section 2 introduces some preliminaries. Section 3 provides a detailed overview of our approach, illustrated with some examples. Section 4 describes the implementation of the divide-and-conquer strategy, while Section 5 details the pruning strategy. Section 6 presents the experimental results. Section 7 reviews works related to our approach. Section 8 concludes the paper and suggests future work.

2. Preliminaries

In this section, we briefly introduce some necessary background knowledge and provide a more detailed discussion on the use of the fundamental Farkas' Lemma.

2.1. Linear Transition System (LinTS)

A Linear Transition System (LinTS) is described by a tuple $\langle X, X', L, \ell^*, \mathcal{T}, \theta \rangle$, where X denotes a finite set of real-valued variables representing the current

$$
\begin{gathered}
X=\{x,y,t\},\ L=\{\ell_0^*,\ell_1\},\ \mathcal{T}=\{\tau_1,\tau_2\},\ \tau_1:\langle\ell_0,\ell_1,\rho_1\rangle,\ \tau_2:\langle\ell_1,\ell_0,\rho_2\rangle,\\
\theta: x=0\wedge y=0\wedge t=0,\\
\rho_1:\begin{bmatrix} t'-t\le x'-x\le 2(t'-t)\wedge \\ t'-t\le y'-y\le 2(t'-t)\wedge \\ 1\le t'-t\le 2\end{bmatrix},\rho_2:\begin{bmatrix} t'-t\le x'-x\le 2(t'-t)\wedge \\ -2(t'-t)\le y'-y\le -(t'-t)\wedge \\ 1\le t'-t\le 2\end{bmatrix}
\end{gathered}
$$

Figure 2: The LinTS for a Vagrant Robot [9]

value, X' represents the values in the next step, L is a finite set of locations including the initial location ℓ^*, $\mathcal{T}$ is a finite set of transitions, with each transition τ specifying the current and next locations ℓ and ℓ' along with a guard condition ρ, and θ represents the initial condition at ℓ^*.

Example 1. *Consider a scenario of a vagrant robot from [4]. The control of the robot works in two alternating modes ℓ_0, ℓ_1. In ℓ_0, the robot moves forward in the positive direction of both x and y through ρ_1, and in ℓ_1, it moves forward in the positive direction of x and the negative direction of y through ρ_2.*

Fig. 2 shows the LinTS of a vagrant robot that consists of three variables x, y, t and two locations ℓ_0, ℓ_1. The variable t records the amount of the elapsed time. θ specifies the initial condition at ℓ^*. A path under this LinTS is $(\ell_0,(x,y,t)=(0,0,0))$, $(\ell_1,(x,y,t)=(2,2,1))$, $(\ell_0,(x,y,t)=(3,1,2))$. □

2.2. Polyhedra, Polyhedral Cones and Projection

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 $\mathbf{b}\in\mathbb{R}^m$. 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. For a polyhedron $P=\{(\mathbf{x}^{\mathrm{T}},\mathbf{u}^{\mathrm{T}})^{\mathrm{T}}\in$

$\mathbb{R}^{p+q} \mid \mathbf{A} \cdot \mathbf{x} + \mathbf{B} \cdot \mathbf{u} \le \mathbf{c}\}$ where $\mathbf{A} \in \mathbb{R}^{m\times p}, \mathbf{B} \in \mathbb{R}^{m\times q}$ are real matrices and $c \in \mathbb{R}^m$ is a real vector, the projection of P onto the dimensions $\mathbf{x}$ is defined as the polyhedron $P[\mathbf{x}] := \{\mathbf{x} \in \mathbb{R}^p \mid \exists \mathbf{u} \in \mathbb{R}^q.(\mathbf{x}^{\mathrm{T}}, \mathbf{u}^{\mathrm{T}})^{\mathrm{T}} \in P\}$. It is guaranteed by Fourier-Motzkin Elimination [11, Chapter 12.2] that $P[\mathbf{x}]$ will always be a polyhedron.

2.3. Farkas' Lemma and its basic use in constraint solving

First, we will provide a more formal description of Farkas' Lemma. And then we will then provide an introduction to the related basic methods. Farkas' Lemma is a fundamental result in linear programming and optimization theory, which provides conditions under which a system of linear inequalities either has a solution or a certain kind of "no solution" condition holds. Specifically, It transforms a set of linear invariants with universal quantifiers into a set of specific forms of nonlinear equations with existential quantifiers.

Theorem 1 (Farkas' Lemma)**.** *Consider an affine assertion φ over $X = \{x_1, \ldots, x_n\}$, where $\bowtie \in \{=, \ge\}$. When φ is satisfiable, $\varphi \models \psi$ if and only if there exists $\lambda \ge 0$ such that (i) $c_j = \sum_{i=1}^{m} \lambda_i a_{ij}$ for $1 \le j \le n$ and (ii) $d = \lambda_0 + \sum_{i=1}^{m} \lambda_i b_i$ as in Fig. 3. Moreover, φ is unsatisfiable if and only if the inequality $\psi : -1 \ge 0$ can be derived.*

The basic approach [4, 7] pioneers a sound and complete framework to generate invariants via Farkas' Lemma with some techniques like quantifier elimination and reasonable heuristics method. The latest approach [9] improves the scalability of the basic approach with a location-by-location strategy and segment subsumption testing strategy, which is the state-of-the-art

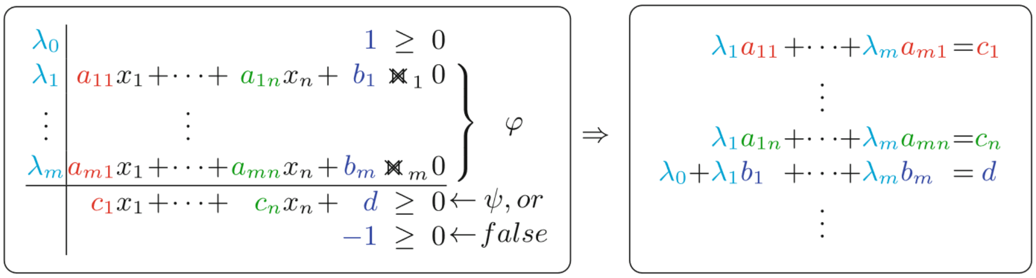


Figure 3: The Tabular Form and Application for Farkas' Lemma [4, 7]

improvement.

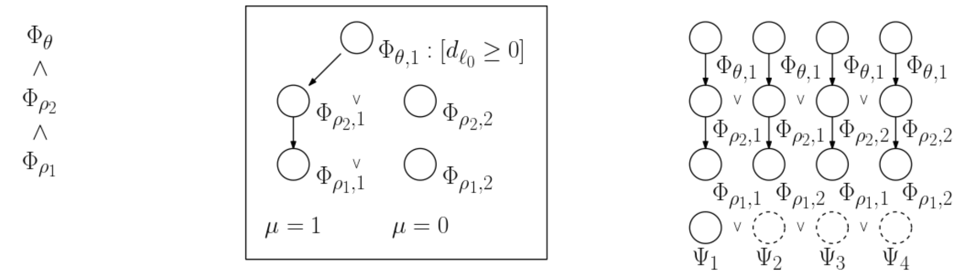
Within this approach, Farkas' Lemma initially transforms the inductive condition for linear invariants into an equivalent system of quadratic constraints in the conjunctive normal form (CNF). After converting CNF to the disjunctive normal form (DNF), we could obtain concrete invariants by solving DNF. To grasp the fundamental ideas of these techniques, we recall the workflow of the basic approach in the following example.

Example 2. *Consider the LinTS in Example 1. The basic approach first establishes a template* $\eta(\ell_i) := c_{\ell_i,1}x + c_{\ell_i,2}y + c_{\ell_i,3}t + d_{\ell_i} \geq 0$ *for* $i \in \{0, 1\}$*. After applying the tabular form of Farkas' Lemma in Fig. 3 and eliminating the* λ_i*'s, we obtain* $\Phi_\theta := [d_{\ell_0} \geq 0]$*,* $\Phi_{\rho,1}$ *by setting* μ *to* 1 *and* $\Phi_{\rho,2}$ *by setting* μ *to* 0 *as shown in Fig. 4 (here substitutes* μ *for* λ *in the specific row of the template), where each atomic proposition is a polyhedral cone over the unknown coefficients* c_{ℓ_i} *and* d_{ℓ_i} *in the CNF formula* $\Phi_\theta \wedge \Phi_{\rho_2} \wedge \Phi_{\rho_1}$*.*

The approach further equivalently expands the CNF formula into the DNF formula. Deriving from the polyhedral cone of one disjunctive clause $\Psi_1 = \Phi_\theta \wedge \Phi_{\rho_{2,1}} \wedge \Phi_{\rho_{1,1}}$*, we obtain their corresponding invariants* $\eta(\ell_0)$

$[-x + 2t \geq 0, -y + t \geq 0, x - t \geq 0, y + t \geq 0]$ *and* $\eta(\ell_1)$

$$[-x+2t \geq 0, -y+t+2 \geq 0, -y+2t \geq 0, t-1 \geq 0, y+t-2 \geq 0, x-t \geq 0].$$



(a) CNF (b) one of expansion (c) DNF

Figure 4: The expansion of CNF-to-DNF following a DFS order

It happens that the invariants obtained from Ψ_1 *above coincide with the ones from the whole DNF formula* $\Psi_1 \vee \Psi_2 \vee \Psi_3 \vee \Psi_4$ *because the polyhedral cone of* $\Psi_2 \vee \Psi_3 \vee \Psi_4$ *were subsumed by* Ψ_1*, i.e.,* $\Psi_1 \supseteq \Psi_2 \vee \Psi_3 \vee \Psi_4$. $\square$

As depicted in Fig. 4, guessing the μ value leads to a combinatorial explosion in the CNF-to-DNF expansion. The latest approach introduces two main techniques to address, including (i) a location-by-location strategy and (ii) a segmented subsumption testing.

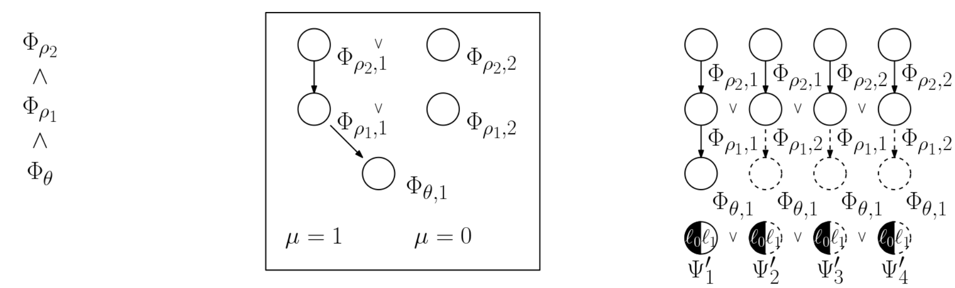
3. An Overview of Our Approach

In this section, we first review the original approaches. Then, we discuss their potential challenges. Finally, aiming at the challenges, we outline our key strategy: divide-and-conquer and pruning.

3.1. The Latest Approach via Farkas' Lemma

Within the latest approach, the location-by-location strategy was employed to split the computation for all locations into separate computations for a single location, and the segmented subsumption testing further decomposes the original CNF-to-DNF converting into a smaller one. These methods still maintain original correctness and accuracy, which is also proved by the latest approach. Since it is the most closely related and recent work to our approach, we revisit its workflow from the perspective of the depth-first search (DFS) in the following examples.

Example 3. *Follow the basic approach in Example 2 and obtain the CNF formula $\Phi_\theta \wedge \Phi_{\rho_2} \wedge \Phi_{\rho_1}$. Recall that the location-by-location strategy focuses on one target location in one invariant generation process by employing two key ideas: (i) reordering and (ii) projection.*



(a) CNF (b) one of expansion (c) DNF

Figure 5: The expansion of CNF-to-DNF reordered by ℓ_1

Suppose that we focus on the target location ℓ_1. (i) Reordering: Based on that the basic approach employs a subsumption testing to detect the redundant disjunctive clauses Ψ_2, Ψ_3, Ψ_4 earlier to mitigate the combinatorial explosion, we can reorder the expansion process then obtain $\Phi_{\rho_2} \wedge \Phi_{\rho_1} \wedge \Phi_\theta$ by prioritizing the expansion related to ℓ_1 as shown in Fig.5a. Notably, Φ_θ only involves ℓ_0, which leads to its de-prioritization. This approach enables the earlier detection of subsumption for redundant disjunctive clauses $\Psi'_2 = \Phi_{\rho_2,1} \wedge \Phi_{\rho_1,2}$

than $\Psi_2 = \Phi_{\rho_2,1} \wedge \Phi_{\rho_1,2} \wedge \Phi_{\theta,1}$*, resulting in* $\Psi_1' \supseteq \Psi_2'$ *as shown in Fig.5c. (ii) Projection: For the polyhedral cone* Ψ_1 *in Fig. 4c, we project the polyhedral cone onto the dimensions of coefficients* c_{1j}*'s (*$1 \leq j \leq 3$*) and* d_1 *that are related to the location* ℓ_1*, to obtain the polyhedral cone* Ψ_1' *as shown in Fig. 5c, where the white right-hand-side represents the reserved dimensions related to* ℓ_1*. The final resultant invariants derived from* Ψ_1' *coincide with the invariants* $\eta(\ell_1)$ *in Example 2. Treat the target location* ℓ_0 *in a similar way.*

Suppose that under the segmented subsumption testing strategy. Consider the scenario that the CNF formula to be expanded is $\Phi = \Phi_1 \wedge \cdots \wedge \Phi_i \wedge \cdots \wedge \Phi_m$ *(*$1 \leq i \leq m$*) where each* Φ_i *is a disjunction* $\Phi_{i,1} \vee \cdots \vee \Phi_{i,j} \vee \cdots \vee \Phi_{i,n}$ *(*$1 \leq j \leq n$*). This strategy divides the CNF into small segments without overlap such as* $\Phi_p \wedge \cdots \wedge \Phi_q$ *(*$1 \leq p \leq q \leq m$*) and then performs the local subsumption testing in each segment. The potential advantage of applying segmented subsumption testing is that small segments may detect local subsumption and get simplified, so that the combinatorial explosion could be mitigated.* □

3.2. Challenge for the Original Approaches

The basic approach uses reasonable heuristics and subsumption testing to avoid the high runtime complexity in quantifier elimination, however, leading to a rapid and massive combinatorial explosion. The latest approach leverages two main improvements to mitigate the combinatorial explosion but still leaves two challenges: (i) how to enable the parallel processing for segmented subsumption testing and (ii) there are redundant and unnecessary checking in the expansion of CNF-to-DNF.

To further reduce the complexity of problems, we improve the approach with two main strategies: (i) a divide-and-conquer strategy to decompose the tasks into mutually independent minimal units to enable parallel processing and (ii) a pruning strategy to reduce the complexity in CNF-to-DNF expansion. In the following, we will outline two key improvements of our approach.

3.3. Divide-and-Conquer Strategy

This strategy targets the CNF-to-DNF expansion process. Segmented subsumption testing divides the CNF formula into segments for local checks but leaves one issue: the scalability of the segmented subsumption testing, which serves as a pre-processing step. We address this issue by introducing a framework that extends the segmented subsumption testing method into a divide-and-conquer strategy. This framework is designed to enhance scalability and further enable parallel processing of CNF-to-DNF expansion.

Example 4. *Continuing with Example 3, we first obtain the CNF* $\Phi_{\rho_2} \wedge \Phi_{\rho_1} \wedge \Phi_\theta$ *equivalent to Fig. 5a. The divide-and-conquer strategy comprises two processes: (i) divide and (ii) conquer.*

In the divide process illustrated in Fig. 6a, we divide the CNF into sub-CNFs, i.e., $\Phi_{\rho_2} \wedge \Phi_{\rho_1}$ *and* Φ_θ*, which are then transformed into sub-DNFs, i.e.,* $\Psi_{a,1}, \Psi_{a,2}, \Psi_{a,3}, \Psi_{a,4}$ *and* $\Psi_{b,1}$*. These sub-DNFs can be operated in parallel. This allows for local checks—determining if* $\Psi_{a,1} \supseteq \Psi_{a,2}$ *or* $\Psi_{a,1} \subseteq \Psi_{a,2}$*, for instance—to be operated in parallel. Operating* n *sub-DNFs costs* $n(n-1)$ *unit-processes and* $n(n-1)$ *unit-time within a single processing framework. However, leveraging parallel processing significantly reduces the computation time among* $\Psi_{a,1}$*,* $\Psi_{a,2}$*,* $\Psi_{a,3}$*, and* $\Psi_{a,4}$ *from* $4 \times (4-1)$ *unit-time to just one unit-time. It is worth mentioning that, reducing the number of conjunctions*

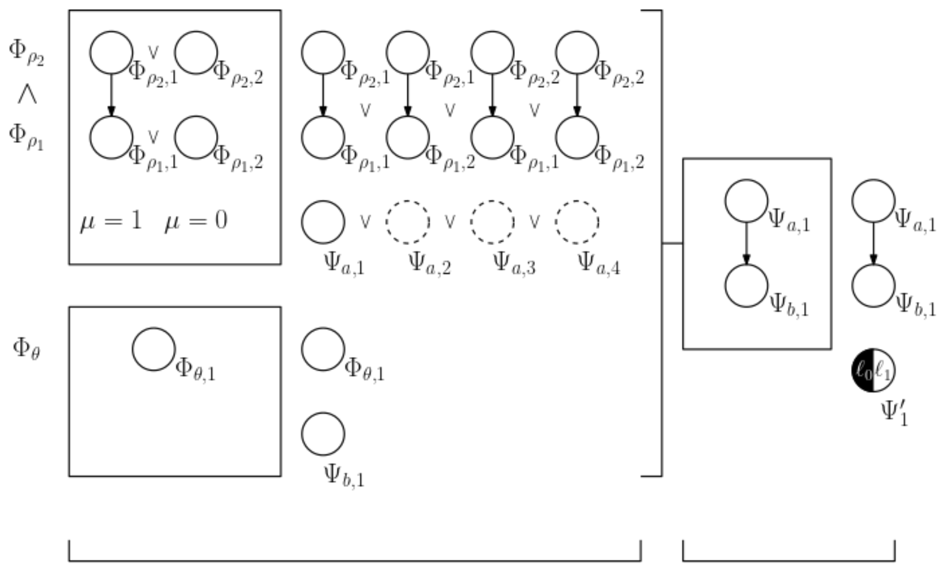


Figure 6: The divide-and-conquer strategy for Example 4

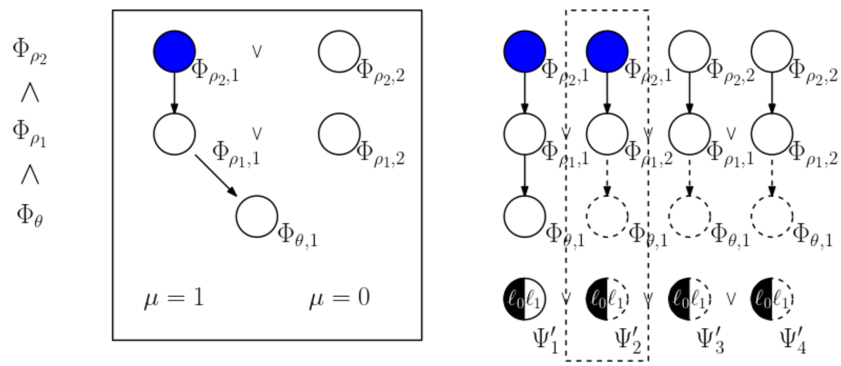
within the sub-CNFs, such as limiting to at most two components like $\Phi_1 \wedge \Phi_2$ *and* Φ_3, *effectively decreases the computational load during polyhedral computations. Finally, the divide process results in two distinct formulas,* $\Psi_{a,1}$ *and* $\Psi_{b,1}$. *In the conquer process illustrated in Fig. 6b, following the divide process, we combine two formulas* $\Psi_{a,1}$ *and* $\Psi_{b,1}$ *into one. The conquer process iteratively merges pairs of sub-DNFs until a complete disjunctive clause is formed, for example,* $\Phi_{\rho_2,i} \wedge \Phi_{\rho_1,j} \wedge \Phi_\theta$. *This process involves continuously conjuncting sub-DNFs until no further sub-DNFs exist. Overall, after obtaining* $\Psi_{a,1}$ *and* $\Psi_{b,1}$ *by the divide process, we merge these to form the final complete disjunctive clause* Ψ'_1, *which is achieved by projecting* $\Psi_{a,1} \wedge \Psi_{b,1}$ *onto the target location* ℓ_1. $\square$

3.4. Pruning Strategy

This strategy focuses on enhancing the pruning capabilities during subsumption checking. We observed that the original approaches rely heavily on the conventional pruning technique limited to a single-step backtrack each time, which unfortunately leads to redundant and unnecessary checks. To address this inefficiency and significantly improve the pruning capabilities for the CNF-to-DNF expansion, we introduce two distinct and innovative pruning strategies.

Path-Recording Multi-Step-Backtracking Pruning (PMP) The PMP strategy tackles the limitations inherent in traditional backtracking pruning by comparing recorded formulas against existing invariants, thus eliminating multiple unnecessary subsumption checks.

Example 5. *Continuing with Example 3, we first obtain* $\Phi_{\rho_2} \wedge \Phi_{\rho_1} \wedge \Phi_\theta$ *equivalent to Fig. 5a. The PMP strategy reveals subsumptions starting with the most comprehensive combination and narrowing down, identifying the earliest subsumption point, which is achieved through the comparison between recorded invariants* η *and current formulas* Φ *of disjunctive clauses. Suppose that under this strategy, we focus on target* ℓ_1*. Upon updating the invariants* $\eta(\ell_1)$ *to include* Ψ_1'*, i.e.,* $\eta(\ell_1) \leftarrow \eta(\ell_1) \cup inv(\Psi_1')$*, we locate the earliest point of subsumption at* $\Phi_{\rho_2,1}$*. To identify this point, we sequentially perform subsumption checks by comparing* $\eta(\ell_1)$ *against progressively smaller sets of formulas within the clause* $\Phi_{\rho_2,1} \wedge \Phi_{\rho_1,1} \wedge \Phi_{\theta,1}$*. The process begins with a comparison involving the entire clause* $\Phi_{\rho_2,1} \wedge \Phi_{\rho_1,1} \wedge \Phi_{\theta,1}$*, then narrows down to* $\Phi_{\rho_2,1} \wedge \Phi_{\rho_1,1}$*, and ultimately focuses on* $\Phi_{\rho_2,1}$ *alone, which determines that the earliest point of subsumption is* $\Phi_{\rho_2,1}$*. The detection of subsumption at*



(a) Pruned Clause

Figure 7: The PMP strategy for Example 5

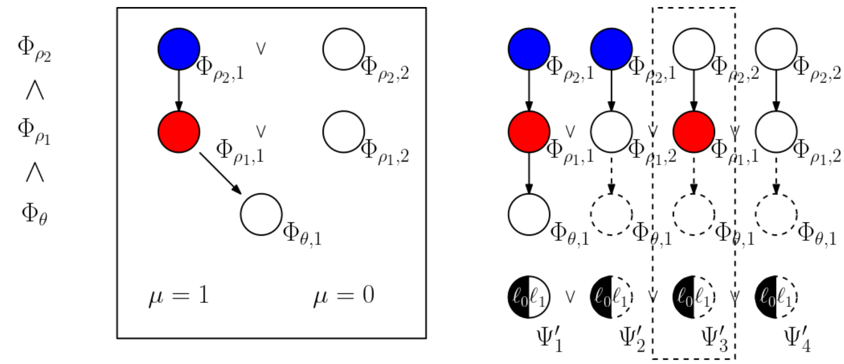
$\Phi_{\rho_2,1}$ *prompts us to prune disjunctive clauses originating from* $\Phi_{\rho_2,1}$*, such as* $\Psi'_2 = \Phi_{\rho_2,1} \wedge \Phi_{\rho_1,2} \wedge \Phi_\theta$*, as demonstrated in Fig. 7a. Here, clauses like* Ψ'_2*, indicated by dashed lines, are exempt from further subsumption checks. Through this strategy, we efficiently eliminate redundant subsumption checks.*

□

Full-Scope-Predictive Forward-Checking Pruning (FFP) The FFP strategy proactively eliminates unnecessary nodes—those already subsumed by newly updated invariants—through subsumption checks between the single formula and recorded invariants, immediately following an invariant update. This strategy ensures that only necessary formulas are processed, significantly reducing the exploration of unnecessary formulas.

Example 6. *Continuing with Example 5, we first obtain* $\Phi_{\rho_2} \wedge \Phi_{\rho_1} \wedge \Phi_\theta$ *equivalent to Fig. 5a. The FFP strategy could prune unnecessary formulas, which is achieved through the comparison between recorded invariants* η *and the single formula* Φ *of disjunctive clauses, such as the polyhedral cones* $\Phi_{\rho_{i},j}$

and $\Phi_{\theta,1}$ *as shown in Fig.8. Suppose that under this strategy, we focus on*



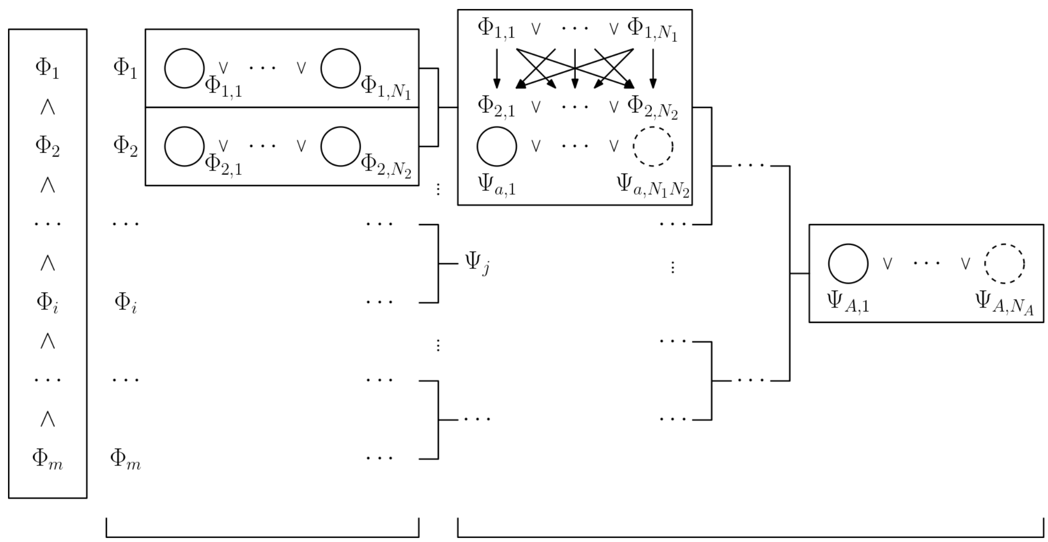
(a) Pruned Clause

Figure 8: The FFP strategy for Example 6

ℓ_1*. Upon updating the invariants* $\eta(\ell_1)$ *to include* Ψ'_1*, i.e.,* $\eta(\ell_1) \leftarrow \eta(\ell_1) \cup inv(\Psi'_1)$*, we locate the point of subsumption at* $\Phi_{\rho_1,1}$*, i.e.,* $\eta(\ell_1) \supseteq \Phi_{\rho_1,1}$*. This point is identified by comparing* $\eta(\ell_1)$ *against each formula* $\Phi_{\rho_i,j}$ *or* $\Phi_{\theta,1}$*. The subsumption happens at* $\Phi_{\rho_1,1}$ *and thus we prune these disjunctive clauses traversing through* $\Phi_{\rho_1,1}$ *such as* $\Psi'_3 = \Phi_{\rho_2,2} \wedge \Phi_{\rho_1,1}$ *as shown in Fig. 8a, where the disjunctive clause* Ψ'_3 *in dashed line do not need subsumption checking. This pruning strategy will remove the unnecessary polyhedral cone.* □

4. Divide-and-Conquer Strategy

This section presents the implementation for the divide-and-conquer strategy (illustrated in Fig. 9). An intuition is that we initially decompose the CNF formula $\Phi_1 \wedge \cdots \wedge \Phi_i \wedge \cdots \wedge \Phi_m$ into independent conjunctive clauses $\Phi_1, \cdots, \Phi_i, \cdots, \Phi_m$ where $\Phi_i = \Phi_{i,1} \vee \cdots \vee \Phi_{i,N_i}$, and we conjunct two neighboring conjunctive clauses into sub-DNF clauses as the segmented subsumption testing to obtain the segmented disjunctive clauses Ψ_j's, then we repeat the merging process until all the neighboring segmented disjunctive clauses were conjuncted together into a set of disjunctive clauses $\Psi := \{\Phi_{1,k_1} \wedge \cdots \wedge \Phi_{i,k_i} \wedge \cdots \wedge \Phi_{m,k_m} | 1 \leq k_i \leq N_i\}$, finally we generate invariants from these disjunctive clauses. With this framework, the combinatorial explosion could be solved by enabling parallel processing.



(a) Divide (b) Conquer

Figure 9: The workflow of Divide-and-Conquer strategy

4.0.1. Divide

For the divide process, we decompose the original CNF formula into the conjunctive clauses by simply breaking the conjunction linking the two neighboring conjunctive clauses, for instance, as shown in Fig.9a. We obtain the CNF formula $\Phi_1 \wedge \cdots \wedge \Phi_i \wedge \cdots \wedge \Phi_m$ where $\Phi_i = \Phi_{i,1} \vee \cdots \vee \Phi_{i,N_i}$ and $1 \leq i \leq m$. Next, we break the conjunction which is connecting every two neighboring conjunctive clauses and obtain these conjunctions $\Phi_1, \cdots, \Phi_i, \cdots, \Phi_m$. Notably, each disjunctive clause Φ_{i,k_i} for $1 \leq k_i \leq N_i$ in Φ_i has been checked their subsumption, and thus, there is no subsumption from each other. $\square$

4.0.2. Conquer

For the conquer process, we repeat the merging process until all the decomposed conjunctive clauses are conjuncted together into the DNF formula, which the DNF formula we obtained is carried without redundant disjunctive clauses. As shown in Fig. 9, we assume that m and $i+1$ are even for the sake of simple illustration. After we obtain the unlinked conjunctions $\Phi_1, \cdots, \Phi_i, \cdots, \Phi_m$ followed by the process of divide, we conquer the CNF-to-DNF process with every two neighboring unit. Below we present the main technical details in following the steps (**Step B1** – **Step B2**).

Step B1. As shown in Fig.9b, we merge each adjacency clause Φ_i and Φ_{i+1} and conjunct every pairwise Φ_{i,k_i} and $\Phi_{i+1,k_{i+1}}$ where there will be $N_i N_{i+1}$ pair of segmented disjunctive clauses. For instance, we merge Φ_1 and Φ_2 by conjuncting Φ_{1,k_1} and Φ_{2,k_2} resulting $N_1 N_2$ pair of disjunctive clauses $(\Phi_{1,1} \wedge \Phi_{2,1}) \vee \cdots \vee (\Phi_{1,1} \wedge \Phi_{2,N_2}) \vee \cdots \vee (\Phi_{1,N_1} \wedge \Phi_{2,1}) \vee \cdots \vee (\Phi_{1,N_1} \wedge \Phi_{2,N_2})$, and we apply subsumption testing to check whether one of them is subsumed by others. Then, the redundant Ψ_{a,k_a} is removed where $\Psi_{a,1} = \Phi_{1,1} \wedge \Phi_{2,1}, \cdots, \Psi_{a,N_1N_2} = \Phi_{1,N_1} \wedge \Phi_{2,N_2}$ and $1 \leq k_a \leq N_1 N_2$, and suppose that the resultant $\Psi_a = \Psi_{a,1} \vee \cdots \vee \Psi_{a,N_a}$. Then we repeat the merging process until every pair in $\Phi_1, \cdots, \Phi_i, \cdots, \Phi_m$ was conjuncted into a disjunctive clause such as $\Psi_a, \cdots, \Psi_j, \cdots$.

Step B2. After collecting $\Psi_a, \cdots, \Psi_j, \cdots$, we apply the method in **Step B1** again on these disjunctive clauses. And then we repeat the process in **Step B2** until all the decomposed conjunctive clauses were conjuncted together into a set of disjunctive clauses $\Psi_A = \Psi_{A,1} \vee \cdots \vee \Psi_{A,N_A}$ which is denoted by $\{\Phi_{1,k_1} \wedge \cdots \wedge \Phi_{i,k_i} \wedge \cdots \wedge \Phi_{m,k_m} | 1 \leq k_i \leq N_i\}$.

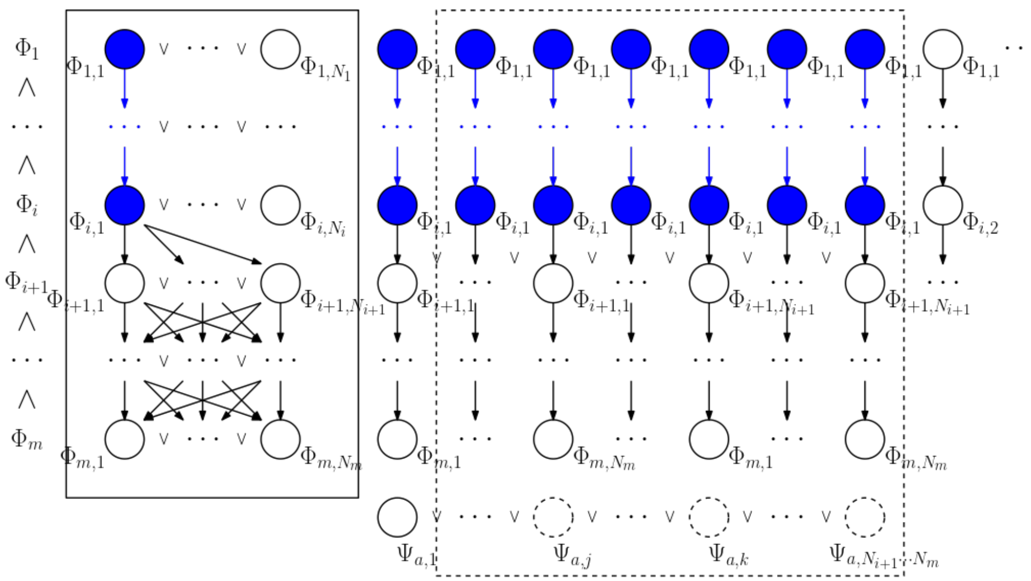
Finally, we generate invariants from these disjunctive clauses of their polyhedral cones, which the DNF formula we obtained has removing redundant disjunctive clauses. □

5. Pruning Strategy

This section presents the pruning strategy. Consider the original subsumption checking strategy as shown in Example 2 and Example 3, the original approaches are burdened with redundant and unnecessary subsumption checking through the DFS search. Our pruning strategy aims at the pruning process and escape the redundant and unnecessary subsumption checking by employing two pruning ideas: (i) Path-Recording Multi-Step-Backtracking Pruning and (ii) Full-Scope-Predictive Forward-Checking Pruning.

5.1. Path-Recording Multi-Step-Backtracking Pruning (PMP)

This section demonstrates the PMP strategy (illustrated in Fig.10). Recall that the invariants will be updated after adding new invariants generated by the polyhedral cones of a complete disjunctive clause $\Psi := \{\Phi_{1,k_1} \wedge \cdots \wedge \Phi_{i,k_i} \wedge \cdots \wedge \Phi_{m,k_m} | 1 \leq k_i \leq N_i\}$, which happens that we do not find successful subsumption along with the current disjunctive clauses. For instance, after we update invariants obtained from the disjunctive clause $\Psi_{a,1} = \Phi_{1,1} \wedge \cdots \wedge \Phi_{i,1} \wedge \Phi_{i+1,1} \wedge \cdots \wedge \Phi_{m,1}$ which is the first disjunctive clause in DFS-searching. Then, we check the subsumption between the current invariants with each polyhedral cone of current disjunctive clauses $\Psi_{a,1}$ step-by-step in a reversing forward. We check whether the polyhedral cones



(a) pruned clauses

Figure 10: The PMP strategy

of $\Phi_{1,1} \wedge \cdots \wedge \Phi_{k,1}$ for $1 \leq k \leq m$ were subsumed by current invariants where the k starts at m and ends at 1. As shown in Fig.10, the earlier subsumption is detected at the disjunctive clauses $\Phi_{1,1} \wedge \cdots \wedge \Phi_{i,1}$. Consider this scenario, all the redundant disjunctive clauses originating from these clauses $\Phi_{1,1} \wedge \cdots \wedge \Phi_{i,1}$ will certainly be subsumed by current invariants and hence should be removed. With this pruning strategy, the redundant disjunctive clauses will be removed without wasting much running time, as shown in Fig. 10. The pseudo-code is given in Algorithm 1.

5.2. Full-Scope-Predictive Forward-Checking Pruning (FFP)

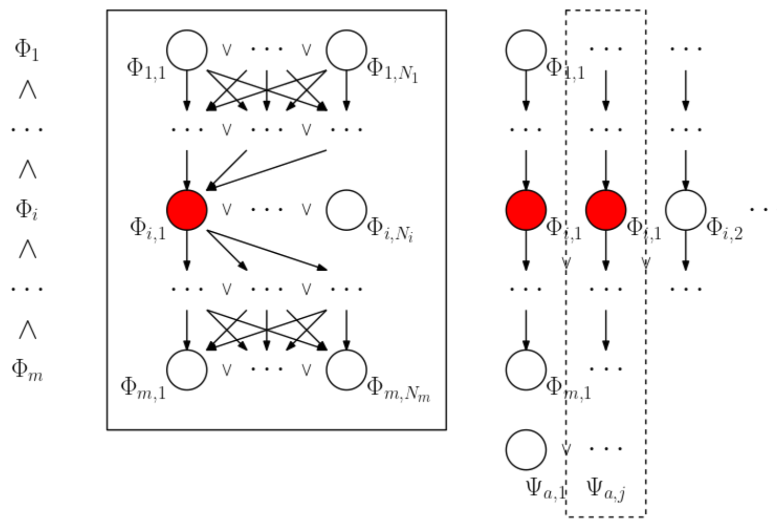
This section demonstrates the FFP strategy (illustrated in Fig. 11). Recall the same happening in Section 5.1 that the invariants updates arisen from $\Psi := \{\Phi_{1,k_1} \wedge \cdots \wedge \Phi_{i,k_i} \wedge \cdots \wedge \Phi_{m,k_m} | 1 \leq k_i \leq N_i\}$. For instance, we up-

Algorithm 1: Path-Recording Multi-Step-Backtracking Pruning

Input: $\bigwedge_{i=1}^{m} \Phi_i$: the CNF formula; $\textsc{Gen}(poly)$: a procedure that generates the invariants given a polyhedron *poly* over the unknown coefficients

Output: a collection *inv* of linear invariants

$\bigwedge_{i=1}^{m} \Phi_i$; $//\Phi_i = \bigvee_{j=1}^{N_i} \Phi_{i,j}$
$inv \leftarrow \emptyset$; $i \leftarrow 1$; $n_i \leftarrow 1$ (for $1 \leq i \leq m$);
while $i > 0$ **do**
    if $n_i \leq N_i$ **then**
        $d_i \leftarrow \Phi_{i,n_i}$; $n_i \leftarrow n_i + 1$;
        if $\bigwedge_{s=1}^{i} d_s \nsubseteq inv$ **then**
            if $i = m$ **then**
                $inv \leftarrow inv \cup \textsc{Gen}(\bigwedge_{s=1}^{m} d_s)$;
                /*Path-Recording Pruning*/
                while $\bigwedge_{s=1}^{i-1} d_s \subseteq inv$ **do**
                    $n_i \leftarrow 1$; $i \leftarrow i - 1$;
            else
                $i \leftarrow i + 1$;
    else
        $n_i \leftarrow 1$; $i \leftarrow i - 1$;
return *inv*;



(a) pruned clauses

Figure 11: The FFP strategy

date invariants obtained from the disjunctive clause $\Psi_{a,1} = \Phi_{1,1} \wedge \cdots \wedge \Phi_{i,1} \wedge \Phi_{i+1,1} \wedge \cdots \wedge \Phi_{m,1}$ which is the first disjunctive clause in DFS-searching. By predictively detecting the subsumption between the current invariants with each polyhedral cone of corresponding disjunctive clauses, we locate that the polyhedral cone $\Phi_{i,1}$ is subsumed. As shown in Fig.11, every disjunctive clause traversing through the $\Phi_{i,1}$ will certainly be subsumed by current invariants so that all the disjunctive clauses carried with $\Phi_{i,1}$ are removed, i.e., the disjunctive clauses $\Psi_{a,j} = \Phi_{1,k_1} \wedge \cdots \wedge \Phi_{i,1} \wedge \cdots \wedge \Phi_{m,k_m}$ for $1 \leq k_i \leq N_i$ except $\Psi_{a,1}$. With this pruning strategy, the disjunctive clauses traversing through the unnecessary polyhedral cones will be removed. The algorithm could be achieved by detecting the unnecessary polyhedral cone (i.e., DFS vertex in Figure 11) and removing it.

6. Evaluation

In this section, we will evaluate the method we propose based on experiments. First, in line with our objectives, we can pose the following research questions to evaluate our method:

RQ1: Are these two strategies, the Divide-and-Conquer strategy and the Pruning strategy, effective individually?

RQ2: Do the two strategies affect each other when accelerating with both strategies applied simultaneously?

Next, we will first introduce the setup of our experiments and then provide a brief explanation and interpretation of our experimental results in relation to the research questions.

6.1. Setup

Dataset We consider benchmarks from a variety of application domains [12, 13, 7, 14]. ***SVCOMP*** *[12].* The C programming language benchmarks selected from the category of loop-invgen in Competition on Software Verification [12]. ***Scheduler*** *[13, 7].* The invariant analysis for cooperative multi-task scheduling activated by interrupts and pre-emptive programming can be used to ensure the liveness of scheduling. The category "Scheduler*" contains original benchmarks from [7] that follow a non-standard setting and "Scheduler" considers these benchmarks with the standard setting [13]. ***Fischer*** *[14].* The Fischer mutual exclusion protocol is a real-time mutual exclusion algorithm for a distributed system with timing skew. ***Cars*** *[7].* A scenario of car systems is illustrated as a dynamic decision problem in which invariant analysis can be used to ensure the safety of autopilot.

Baseline Since the underlying principles of all methods in constraint solving via Farkas' Lemma are consistent, we obtain the same invariants as the original approaches [4, 7, 9] so that we mainly focus on the comparison of runtime(**Time(s)**), successful subsumption count(**SS**) and pruning count(**P/F**). More specifically, In all tables, "Lines of Code" means the number of lines of C code used to measure the scale, "Our Approach" means the experimental results obtained by our approach, "StInG" means the experimental results obtained by basic approach in StInG [7], "OOPSLA22" means the experimental results obtained by latest approach in OOPSLA22 [9], "Time (s)" means the runtime for DNF transformation following preliminary preparations measured in seconds, "SS" means the number of successful subsumption in the process of CNF-to-DNF expansion, "Speedup" shows the ratio of the runtime consumed by OOPSLA22 against our corresponding approach specified in the table (i.e., OOPSLA22/Ours), "D-and-C" means the experimental results obtained by divide-and-conquer strategy, "P-Pruning" means the experimental results obtained by path-recording multi-step-backtracking pruning strategy, "F-Pruning" means the experimental results obtained by full-scope-predictive forward-checking pruning strategy, "P" means the number of pruning in the process of P-Pruning, "F" means the number of pruning in the process of F-Pruning, the symbol "-" means that not applicable due to either the absence of code representation or time-out or out-of-memory. We set a time-out of 10 minutes for all the tables.

Environment Experiments were conducted on a system with an Intel Core i7-7700 CPU (3.6 GHz) and 16 GiB RAM, running Ubuntu 22.04.4 LTS.

6.2. Overview of results

Table 1: Experimental Results for Divide-and-Conquer Strategy

Benchmarks			StInG		OOPSLA22		Our Approach							
Name		Lines of Code	Time (s)	SS	Time (s)	SS	Only D-and-C Strategy			D-and-C + P + F				
							Time (s)	SS	Speedup	Time (s)	SS	P	F	Speedup
SVCOMP	down	-	0	9	0.01	15	0.01	27	1.00x	<0.01	22	2	3	-
	frag	-	0	109	0.05	102	0.05	234	1.00x	0.03	224	13	2	1.67x
	half	-	0	32	0.02	49	0.01	123	2.00x	0.01	116	27	3	2.00x
	large	-	0	123	0.03	146	0.02	503	1.50x	0.03	496	20	2	1.00x
	seq	-	0	39	0.01	48	0.01	108	1.00x	0.01	99	11	4	1.00x
	upb	-	0	9	0.01	15	0.01	27	1.00x	0.01	22	2	3	1.00x
	up	-	0	60	0.02	52	0.01	113	2.00x	0.01	106	20	0	2.00x
Scheduler*	2p	-	0.01	26	0.01	26	0.01	46	1.00x	<0.01	47	3	2	-
	3p	-	0.17	380	0.06	207	0.04	407	1.50x	0.05	410	13	3	1.20x
	4p	-	60.81	26,629	0.48	1,348	0.19	2,010	2.53x	0.26	2,013	33	4	1.85x
	5p	-	7,436.34	2,548,704	6.88	15,172	0.82	10,177	8.39x	0.73	10,178	65	5	9.42x
Scheduler	3p	336	0.17	279	0.05	234	0.03	517	1.67x	0.05	519	10	3	1.00x
	4p	609	4.16	2,895	0.36	1,318	0.21	3,110	1.71x	0.23	3,111	23	5	1.57x
	5p	1017	135.8	39,150	3.8	6,728	0.83	17,095	4.58x	0.98	17,099	55	6	3.88x
	6p	1587	7,541.53	906,454	68.46	51,342	4.53	57,661	15.11x	4.05	57,668	82	7	16.90x
Fischer	6p	710	9.18	8,423	0.67	2,808	0.33	8,559	2.03x	0.32	8,553	100	0	2.09x
	7p	987	59.16	32,668	2.62	6,188	0.72	39,345	3.64x	0.73	39,338	121	0	3.59x
	8p	1327	373.62	127,918	10.69	14,231	1.17	148,044	9.14x	1.33	148,036	205	0	8.04x
Cars	2p	216	0.01	28	0.01	61	0.01	130	1.00x	0.01	131	2	1	1.00x
	3p	616	560.7	788,508	0.94	2,279	0.07	1,912	13.43x	0.06	1,913	4	1	15.67x
	4p	1283	-	-	83.87	37,525	0.48	43,931	174.73x	0.70	43,931	7	1	119.81x

We summarize experimental results in the following tables which are Table 1 and Table 2. Then, we describe the tables and discuss the details of our experimental results.

In Table 1, we compare the runtime, successful subsumption count, and pruning count with StInG and OOPSLA22 under the setting that our approach employs P-Pruning and F-Pruning strategy with divide-and-conquer strategy, where requires a trading off between using more memory and processing more local subsumption in parallel. We can observe that the D-and-C

Table 2: Experimental Results for Pruning Strategy

Benchmarks			**StInG**		**OOPSLA22**		**Our Approach**							
Name		**Lines of Code**	**Time (s)**	**SS**	**Time (s)**	**SS**	**Only P-Pruning Strategy**				**Only F-Pruning Strategy**			
							Time (s)	**SS**	**P**	**Speedup**	**Time (s)**	**SS**	**F**	**Speedup**
SVCOMP	down	-	0	9	0.01	15	0.01	6	5	1.00x	0.01	7	3	1.00x
	frag	-	0	109	0.05	102	0.03	71	15	1.67x	0.05	91	2	1.00x
	half	-	0	32	0.02	49	0.01	26	30	2.00x	0.02	35	3	1.00x
	large	-	0	123	0.03	146	0.03	110	22	1.00x	0.03	133	2	1.00x
	seq	-	0	39	0.01	48	0.02	25	15	0.50x	0.01	29	4	1.00x
	upb	-	0	9	0.01	15	0.01	6	5	1.00x	0.01	7	3	1.00x
	up	-	0	60	0.02	52	0.01	32	20	2.00x	0.01	52	0	2.00x
Scheduler*	2p	-	0.01	26	0.01	26	0	13	8	-	0.01	19	4	1.00x
	3p	-	0.17	380	0.06	207	0.05	102	23	1.20x	0.05	176	7	1.20x
	4p	-	60.81	26,629	0.48	1,348	0.46	1,043	51	1.04x	0.47	1,347	9	1.02x
	5p	-	7,436.34	2,548,704	6.88	15,172	6.28	14,454	86	1.10x	7.84	15,076	13	0.88x
	6p	-	-	-	299.77	54,862	295.21	52,348	151	1.02x	298.89	54,692	29	1.00x
Scheduler	3p	336	0.17	279	0.05	234	0.07	112	22	0.71x	0.08	169	9	0.63x
	4p	609	4.16	2,895	0.36	1,318	0.73	933	44	0.49x	0.66	1,210	14	0.55x
	5p	1017	135.8	39,150	3.8	6,728	3.84	5,677	85	0.99x	5.08	6,419	20	0.75x
	6p	1587	7,541.53	906,454	68.46	51,342	66.35	49,206	122	1.03x	68.43	50,988	27	1.00x
	7p	2352	-	-	742.25	299,471	730.70	295,661	187	1.02x	747.35	298,531	35	0.99x
Fischer	6p	710	9.18	8,423	0.67	2,808	0.82	1,496	100	0.82x	0.82	2,808	0	0.82x
	7p	987	59.16	32,668	2.62	6,188	3.83	3,936	121	0.68x	3.21	6,188	0	0.82x
	8p	1327	373.62	127,918	10.69	14,231	5.28	10,290	205	2.02x	13.63	14,231	0	0.78x
	9p	1736	2,345.96	503,369	43.14	34,033	36.78	28,117	236	1.17x	47.47	34,033	0	0.91x
	10p	2218	14,664.68	1,985,857	186.40	90,362	161.33	81,102	366	1.16x	195.32	90,362	0	0.95x
	11p	2780	-	-	766.24	235,812	720.10	222,982	409	1.06x	783.37	235,812	0	0.98x
Cars	2p	216	0.01	28	0.01	61	0.02	51	3	0.50x	0.01	57	1	1.00x
	3p	616	560.7	788,508	0.94	2,279	1	2,245	5	0.94x	1.53	2,275	1	0.61x
	4p	1283	-	-	83.87	37,525	89.4	37,454	10	0.94x	85.4	37,519	1	0.98x

strategy could reduce the runtime. The successful subsumption count for the D-and-C strategy is more than StInG and OOPSLA22 because the subsumption was solved in the segment. The pruning count also represents that the pruning strategy works and the speedup shows that the D-and-C strategy actually has a positive effect.

In Table 2, we compare the runtime, successful subsumption count, and pruning count with the tool of StInG and OOPSLA22 under the setting that our approach only employs P-Pruning strategy or only employs F-Pruning strategy without divide-and-conquer strategy. We can observe that the

P-Pruning strategy could reduce the successful subsumption count, which means that successful subsumption could be detected earlier leading to a lower subsumption count. The pruning count also represents that the P-Pruning strategy could work and the speedup also indicates that less runtime is needed if the redundant and unnecessary checking is removed.

Table 1 also shows that when both strategies are used simultaneously, they both perform effectively and contribute to acceleration. By combining these two strategies, we achieve an average speedup of 9.27x, while still maintaining efficient subsumption and pruning processes.

6.3. Answers to research questions

The answer to RQ1 The table 1 shows that the Divide and Conquer strategy could reduce the runtime, and increase the number of successful subsumption counts, which means even when using this D-and-C individually, it leads to significant improvements in time speedup for all examples and provides highly efficient acceleration. This improvement is particularly noticeable in more complex examples with larger codebases (such as 4p in Cars). In the original method, complex linear systems often contain more clauses, which also indicates that there is more room for running the D-and-C strategy effectively.

Correspondingly, Table 2 also demonstrates that the pruning strategy provides a good speedup and, by identifying prunable paths, significantly reduces the number of successful subsumptions counts. This reduction in subsumptions leads to a substantial decrease in memory usage during algorithm execution. As shown, for some very large examples, the use of the pruning strategy enables these examples to be computed within limited time and space constraints.

The answer to RQ2 The experiment result of Table 1 shows that the combined approach, which strengthens both strategies, also achieves a significant speedup, demonstrating excellent performance. At the same time, compared to using D-and-C alone, it significantly reduces the number of subsumptions. Compared to using pruning alone, it greatly reduces the computation time. The combination of these two strategies substantially lowers the computational resource requirements for solving the constraint-solving problem raised from invariant generation via Farka's Lemma. It further proves that, in most cases, these two strategies can be used together with quite favorable results.

7. Related Work

Our method is an improvement to an existing constraint-solving-based approach for the automatic generation of linear loop invariants. Therefore, we will first discuss the similarities and differences between our approach and other constraint-solving-based methods. Next, we will introduce several other methods for generating loop invariants, including the classical abstract interpretation approach and the more recent learning-based methods.

7.1. Constraint Solving

Our invariant generation approach falls in the category of constraint solving. Most relevantly, we compare our approach with the template-based linear invariant generation which is based on Farkas' Lemma. The approach proposed in [4] pioneers the framework of linear invariant generation using

Farkas' Lemma, which is based on the template. It generates linear invariants through quantifier elimination and several heuristics but suffers from high runtime complexity. After that, the approach proposed in [9] focuses on improving the scalability of the approaches [7] by employing the location-by-location strategy with the segmented subsumption testing. Furthermore, our approach employs two strategies to mitigate the combinatorial explosion caused by using heuristics, hence enhancing the scalability of the latest approach [9].

The approach proposed in [15] introduces a method for reducing the second-order constraints (derived from Farkas' Lemma) to SAT formulate which is then solved by the SAT solver, focusing on the field of inter-procedural program verification, weakest precondition and strongest postcondition inference. However, their approach aims to present new techniques and applications in program analysis, but our approach aims to boost up the process of program analysis. Thus, their approach is orthogonal to ours.

The approach proposed in [16] and the tool InvGen [17] introduce a method for integrating abstract interpretation and constraint solving, while our approach aims at speeding up and enhancing the scalability in constraint solving. Thus, their approaches are orthogonal to ours.

The approach proposed in [18] suggests a matrix-algebra method with Farkas' Lemma to synthesize linear invariant only over the single unnested while-loop, but our approach considers the general transition system for arbitrary program structure. Thus, its approach is orthogonal to ours.

The approach proposed in [19] concentrates on the invariants of the bit-vector program. Li et al. [20] leads to considering invariant generation under constraint-solving methods that involve modular arithmetic. Other relevant approaches proposed in [21, 22, 23, 24, 25, 26] describe the method that handles the polynomial invariant generation problems, which focuses on a different aspect to linear invariant generation and thus is orthogonal to ours.

7.2. Abstraction Interpretation

The most classical method for invariant generation is based on the abstract interpretation framework [5]. This framework first defines the abstract domain and then uses different abstract states to approximate various program states. Inferences are made based on each abstract successor to derive invariants [27]. However, this method often faces termination issues due to the infinite nature of abstract states, making it difficult to guarantee completeness [28]. Although later methods incorporating widening operators [29, 6, 30] significantly addressed this problem, they introduced further challenges related to precision.

Another major category of invariant generation methods that can also be considered under the abstract interpretation framework involves predicate abstraction, including techniques such as abductive inference [31] and Craig interpolants [32]. Some works focus on lazy invariant generation [33, 34, 35], where a specific assertion after or within a loop is given, and the generated invariant only needs to satisfy this assertion. In contrast, most constraint-solving-based approaches are eager, aiming to generate tighter and more precise invariants whenever possible.

The recurrence analysis [36, 37] is only applicable to the scenarios where the recurrence relation can be solved with a closed form solution. In contrast, constraint solving offers a more generally applicable approach, capable of addressing a wider range of problems even when a closed form solution does not exist.

7.3. Other Methods

Learning-based methods often extend template-based approaches by optimizing the parameters within these templates using various machine learning techniques, ultimately employing an SMT solver to filter and select suitable invariants. C2I [38] adopts a randomized search strategy to identify candidate invariants, followed by a validation step using a checker. ICE [39] employs learning techniques to synthesize invariants, leveraging both examples and counterexamples, as well as implications, and subsequently integrates these with decision trees [40]. Xu et al. [41] enhance invariant learning by utilizing interval counterexamples. CODE2INV [42, 43] employs reinforcement learning in conjunction with graph neural networks to model and learn program structures. LIPuS [44] also leverages reinforcement learning, incorporating pruning techniques to reduce the search space. CLN2INV [45] introduces Continuous Logic Networks (CLN) to automatically learn loop invariants directly from program execution traces, while Gated-CLN (G-CLN) [46] extends this approach to enable the model to robustly learn general invariants across a wide range of terms. However, this approach is very fast, but its correctness is difficult to guarantee, and it struggles to handle programs and invariants that differ significantly from the training data structure.

Recently, several methods for invariant generation based on Large Language Models (LLMs) have emerged. Autospec [47] employs hierarchical specification generation to produce ACSL-style annotations, including loop invariants and has shown strong performance in handling numerical programs. Kexin et al. [48] fine-tuned a pre-trained LLM to perform abstract reasoning over program executions, facilitating the generation of loop invariants. Their approach integrates a dynamic analyzer to trace program executions, which are subsequently used to derive invariants. LIG-SE [49] fine-tuned a large model and, in conjunction with a checker, improved the model's capacity to handle separation logic. Although these tools have made progress in terms of generality, enabling the generation of invariants for programs with different structures, they tend to generate weaker invariants for each case. Furthermore, they cannot guarantee soundness or completeness. Only some of the tools that incorporate determinate checkers can ensure soundness, but this reliance on the performance of the checker introduces an additional dependency and potential limitation in terms of scalability and efficiency.

There are also some methods that use dynamic analysis [50, 51, 52] to generate loop invariants, which is clearly not aligned with the static analysis tools we aim for, and therefore, it falls outside the scope of our discussion.

8. Conclusion

We propose an optimized algorithm to address the combinatorial explosion issue by trading off space for time-efficiency. Our approach employs two key strategies to boost speed. First, we apply a divide-and-conquer strategy to decompose a complex problem into smaller, more manageable subproblems that can be solved quickly and in parallel. Second, we utilize the pruning strategy within two intelligent ways to navigate through the depth-first search process to avoid redundant and unnecessary checks. These improvements maintain the accuracy and speed up the analysis. Our experiments indicate that our approach outperforms the state-of-the-art, demonstrating significant speed improvements. With this solution, we make a significant advance in speeding up the invariant generation with Farkas' Lemma. Future work could consider more powerful pruning strategies to reduce redundant and unnecessary checks.

References

[1] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, S. Shoham, Ivy: safety verification by interactive generalization, in: PLDI, ACM, 2016, pp. 614–630. doi:10.1145/2908080.2908118.

[2] A. Asadi, K. Chatterjee, H. Fu, A. K. Goharshady, M. Mahdavi, Polynomial reachability witnesses via stellensätze, in: PLDI, ACM, 2021, pp. 772–787. doi:10.1145/3453483.3454076.

[3] K. Chatterjee, H. Fu, A. K. Goharshady, Non-polynomial worst-case analysis of recursive programs, ACM Trans. Program. Lang. Syst. 41 (2019) 20:1–20:52. doi:10.1145/3339984.

[4] M. Colón, S. Sankaranarayanan, H. Sipma, Linear invariant generation using non-linear constraint solving, in: CAV, volume 2725 of *LNCS*, Springer, 2003, pp. 420–432. doi:10.1007/978-3-540-45069-6_39.

[5] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL

'77, Association for Computing Machinery, New York, NY, USA, 1977, p. 238–252. URL: https://doi.org/10.1145/512950.512973. doi:10.1145/512950.512973.

[6] E. Rodríguez-Carbonell, D. Kapur, An abstract interpretation approach for automatic generation of polynomial invariants, in: SAS, volume 3148 of *LNCS*, Springer, 2004, pp. 280–295. doi:10.1007/978-3-540-27864-1_21.

[7] S. Sankaranarayanan, H. B. Sipma, Z. Manna, Constraint-based linear-relations analysis, in: SAS, volume 3148 of *LNCS*, Springer, 2004, pp. 53–68. doi:10.1007/978-3-540-27864-1_7.

[8] R. Giacobazzi, F. Ranzato, Completeness in abstract interpretation: A domain perspective, in: Algebraic Methodology and Software Technology, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 231–245.

[9] H. Liu, H. Fu, Z. Yu, J. Song, G. Li, Scalable linear invariant generation with farkas' lemma, Proc. ACM Program. Lang. 6 (2022) 204–232. doi:10.1145/3563295.

[10] H. Liu, G. Li, Empirically scalable invariant generation leveraging divide-and-conquer with pruning, in: Theoretical Aspects of Software Engineering, Springer Nature Switzerland, Cham, 2024, pp. 324–342.

[11] A. Schrijver, Theory of linear and integer programming, Wiley-Interscience series in discrete mathematics and optimization, Wiley, 1999.

[12] SV-COMP, Software verification competition, `https://sv-comp.sosy-lab.org`, 2023.

[13] N. Halbwachs, Y. Proy, P. Roumanoff, Verification of real-time systems using linear relation analysis, Formal Methods Syst. Des. 11 (1997) 157–185. doi:10.1023/A:1008678014487.

[14] L. Lamport, A fast mutual exclusion algorithm, ACM Trans. Comput. Syst. 5 (1987) 1–11. doi:10.1145/7351.7352.

[15] S. Gulwani, S. Srivastava, R. Venkatesan, Program analysis as constraint solving, in: PLDI, ACM, 2008, pp. 281–292. doi:10.1145/1375581.1375616.

[16] A. Gupta, R. Majumdar, A. Rybalchenko, From tests to proofs, in: Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, volume 5505 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 262–276. doi:10.1007/978-3-642-00768-2_24.

[17] A. Gupta, A. Rybalchenko, Invgen: An efficient invariant generator, in: CAV, volume 5643 of *LNCS*, Springer, 2009, pp. 634–640. doi:10.1007/978-3-642-02658-4_48.

[18] Y. Ji, H. Fu, B. Fang, H. Chen, 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, volume 13371 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 257–281. doi:10.1007/978-3-031-13185-1_13.

[19] P. Yao, J. Ke, J. Sun, H. Fu, R. Wu, K. Ren, Demystifying template-based invariant generation for bit-vector programs, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023, pp. 673–685. doi:10.1109/ASE56229.2023.00069.

[20] L. Yuchen, F. Hongfei, L. Haowen, G. Li, Constraint based invariant generation with modular operations, in: Dependable Software Engineering. Theories, Tools, and Applications, 2024, pp. 1–21.

[21] E. Rodríguez-Carbonell, D. Kapur, Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations, in: ISSAC, ACM, 2004, pp. 266–273. doi:10.1145/1005285.1005324.

[22] Y. Chen, C. Hong, B. Wang, L. Zhang, Counterexample-guided polynomial loop invariant generation by lagrange interpolation, in: CAV, volume 9206 of *LNCS*, Springer, 2015, pp. 658–674. doi:10.1007/978-3-319-21690-4_44.

[23] A. Adjé, P. Garoche, V. Magron, Property-based polynomial invariant generation using sums-of-squares optimization, in: SAS, volume 9291 of *LNCS*, Springer, 2015, pp. 235–251.

[24] S. de Oliveira, S. Bensalem, V. Prevosto, Polynomial invariants by linear algebra, in: ATVA, volume 9938 of *LNCS*, 2016, pp. 479–494. doi:10.1007/978-3-319-46520-3_30.

[25] E. Hrushovski, J. Ouaknine, A. Pouly, J. Worrell, Polynomial invariants for affine programs, in: LICS, ACM, 2018, pp. 530–539. doi:10.1145/3209108.3209142.

[26] K. Chatterjee, H. Fu, A. K. Goharshady, E. K. Goharshady, Polynomial invariant generation for non-deterministic recursive programs, in: PLDI, ACM, 2020, pp. 672–687. doi:10.1145/3385412.3385969.

[27] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: POPL, ACM Press, 1978, pp. 84–96. doi:10.1145/512760.512770.

[28] R. Giacobazzi, F. Ranzato, Completeness in abstract interpretation: A domain perspective, in: AMAST, volume 1349 of *LNCS*, Springer, 1997, pp. 231–245. doi:10.1007/BFb0000474.

[29] E. Rodríguez-Carbonell, D. Kapur, Program verification using automatic generation of invariants, in: ICTAC, volume 3407 of *LNCS*, Springer, 2004, pp. 325–340. doi:10.1007/978-3-540-31862-0_24.

[30] E. Rodríguez-Carbonell, D. Kapur, Automatic generation of polynomial invariants of bounded degree using abstract interpretation, Sci. Comput. Program. 64 (2007) 54–75. doi:10.1016/j.scico.2006.03.003.

[31] I. Dillig, T. Dillig, B. Li, K. L. McMillan, Inductive invariant generation via abductive inference, in: OOPSLA, ACM, 2013, pp. 443–456. doi:10.1145/2509136.2509511.

[32] S.-W. Lin, J. Sun, H. Xiao, Y. Liu, D. Sanán, H. Hansen, Fib: Squeezing loop invariants by interpolation between forward/backward pred-

icate transformers, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 793–803. doi:10.1109/ASE.2017.8115690.

[33] K. L. McMillan, Lazy abstraction with interpolants, in: Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 123–136.

[34] K. L. McMillan, Lazy annotation for program testing and verification, in: Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 104–118.

[35] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, N. Sharygina, Lazy abstraction with interpolants for arrays, in: Logic for Programming, Artificial Intelligence, and Reasoning, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 46–61.

[36] A. Farzan, Z. Kincaid, Compositional recurrence analysis, in: FMCAD, IEEE, 2015, pp. 57–64.

[37] J. Breck, J. Cyphert, Z. Kincaid, T. W. Reps, Templates and recurrences: better together, in: PLDI, ACM, 2020, pp. 688–702. doi:10.1145/3385412.3386035.

[38] R. Sharma, A. Aiken, From invariant checking to invariant inference using randomized search, in: Computer Aided Verification, Springer International Publishing, Cham, 2014, pp. 88–105.

[39] P. Garg, C. Löding, P. Madhusudan, D. Neider, Ice: a robust framework for learning invariants, in: Computer Aided Verification, Springer International Publishing, Cham, 2014, pp. 69–87.

[40] P. Garg, D. Neider, P. Madhusudan, D. Roth, Learning invariants using decision trees and implication counterexamples, SIGPLAN Not. 51 (2016) 499–512. URL: https://doi.org/10.1145/2914770.2837664. doi:10.1145/2914770.2837664.

[41] R. Xu, F. He, B. Wang, Interval counterexamples for loop invariant learning, in: ESEC/FSE, ACM, 2020, pp. 111–122. doi:10.1145/3368089.3409752.

[42] X. Si, H. Dai, M. Raghothaman, M. Naik, L. Song, Learning loop invariants for program verification, in: Advances in Neural Information Processing Systems, volume 31, Curran Associates, Inc., Palais des Congrès de Montréal, Montréal CANADA, 2018, pp. 7762–7773.

[43] X. Si, A. Naik, H. Dai, M. Naik, L. Song, Code2inv: A deep learning framework for program verification, in: Computer Aided Verification, Springer International Publishing, Cham, 2020, pp. 151–164.

[44] S. Yu, T. Wang, J. Wang, Loop invariant inference through smt solving enhanced reinforcement learning, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Association for Computing Machinery, New York, NY, USA, 2023, p. 175–187. URL: https://doi.org/10.1145/3597926.3598047. doi:10.1145/3597926.3598047.

[45] G. Ryan, J. Wong, J. Yao, R. Gu, S. Jana, Cln2inv: Learning loop invariants with continuous logic networks, in: International Conference on Learning Representations, OpenReview.net, Addis Ababa, Ethiopia, 2020, pp. 114–122. URL: https://openreview.net/forum?id=HJlfuTEtvB.

[46] J. Yao, G. Ryan, J. Wong, S. Jana, R. Gu, Learning nonlinear loop invariants with gated continuous logic networks, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 106–120. URL: https://doi.org/10.1145/3385412.3385986. doi:10.1145/3385412.3385986.

[47] C. Wen, J. Cao, J. Su, Z. Xu, S. Qin, M. He, H. Li, S.-C. Cheung, C. Tian, Enchanting program specification synthesis by large language models using static analysis and program verification, in: Computer Aided Verification, Springer Nature Switzerland, Cham, 2024, pp. 302–328.

[48] K. Pei, D. Bieber, K. Shi, C. Sutton, P. Yin, Can large language models reason about program invariants?, in: Proceedings of the 40th International Conference on Machine Learning, volume 202 of *Proceedings of Machine Learning Research*, PMLR, Honolulu, Hawaii, USA, 2023, pp. 27496–27520. URL: https://proceedings.mlr.press/v202/pei23a.html.

[49] C. Liu, X. Wu, Y. Feng, Q. Cao, J. Yan, Towards general loop invariant

generation: A benchmark of programs with memory manipulation, 2024. URL: https://arxiv.org/abs/2311.10483. arXiv:2311.10483.

[50] C. Csallner, N. Tillmann, Y. Smaragdakis, Dysy: dynamic symbolic execution for invariant inference, in: ICSE, ACM, 2008, pp. 281–290. doi:10.1145/1368088.1368127.

[51] T. Nguyen, D. Kapur, W. Weimer, S. Forrest, Using dynamic analysis to discover polynomial and array invariants, in: ICSE, IEEE Computer Society, 2012, pp. 683–693. doi:10.1109/ICSE.2012.6227149.

[52] T. C. Le, G. Zheng, T. Nguyen, 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, Association for Computing Machinery, New York, NY, USA, 2019, p. 788–801. URL: https://doi.org/10.1145/3314221.3314634. doi:10.1145/3314221.3314634.