Automatic Verification of Linear Integer Planning Programs via Forgetting in LIAUPF

Liangda Fang
Jinan University
Pazhou Lab
Guangzhou, China
fangld@jnu.edu.cn

Shikang Chen
Jinan University
Guangzhou, China
jnu202134261129@stu2021.jnu.edu.cn

Xiaoman Wang
Jinan University
Guangzhou, China
wangxm@stu2022.jnu.edu.cn

Xiaoyou Lin
Jinan University
Guangzhou, China
linxiaoyou@stu2019.inu.edu.cn

Chenyi Zhang
University of Canterbury
Christchurch, New Zealand
chenyi.zhang@gmail.com

Qingliang Chen
Jinan University
Guangzhou, China
tpchen@jnu.edu.cn

Quanlong Guan*
Jinan University
Guangzhou, China
gql@jnu.edu.cn

Kaile Su†
Yantai University
Yantai, China
kailepku@gmail.com

ABSTRACT

The goal of generalized planning (GP) is to find a generalized solution for a class of planning problems. One of effective means to solve GP is to transform a GP problem into an abstract planning problem, which can be easily solved. Recently, Lin et al. proposed a novel abstract model for GP, namely generalized linear integer numeric planning (GLINP), whose solution is an algorithmic-like structure called a planning program. They also developed an inductive approach to generating planning programs for GLINP. However, it has no theoretical guarantee that the generated planning program holds for infinitely many problem instances. To address this defect, we propose an automatic approach to verify whether the planning program works for infinitely many problem instances in this paper. We translate the planning program into a set of trace axioms finitely represented by linear integer arithmetic with uninterpreted predicate and function symbols (LIAUPF), and reduce the problem to the entailment problem of LIAUPF. Due to the undecidability of entailment problem in LIAUPF, we identify a class of planning programs whose trace axioms can be simplified in linear integer arithmetic (LIA), that is, a decidable fragment of LIAUPF, when reasoning about only the input and output of planning programs. As a result, the correctness verification of this class of programs becomes decidable.

KEYWORDS

Generalized Planning, Program Verification, Linear Integer Arithmetic

*Corresponding author.
†Corresponding author.

Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Y. Vorobeychik, S. Das, A. Nowé (eds.), May 19 – 23, 2025, Detroit, Michigan, USA. © 2025 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

ACM Reference Format:
Liangda Fang, Shikang Chen, Xiaoman Wang, Xiaoyou Lin, Chenyi Zhang, Qingliang Chen, Quanlong Guan, and Kaile Su. 2025. Automatic Verification of Linear Integer Planning Programs via Forgetting in LIAUPF. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 9 pages.

1 INTRODUCTION

Automated planning is an important area of Artificial Intelligence. Two fundamental classes of automated planning are classical planning and numeric planning. A classical planning problem is defined as (1) a collection of propositional variables that are used to describe states; (2) a set of actions that indicate the transition relation between states; (3) an initial state; (4) a goal formula, which is a propositional formula that captures a set of goal states. Numeric planning [12, 24] is a numeric extension to classical planning, which involves not only propositional variables but also numeric variables. Both classical and numeric planning aim to identify a finite sequence of actions that ensures goal achievement from the initial state. The solution, however, is strongly related to the specific planning problem and is not generally applicable to other similar planning problems.

A crucial extension to numeric or classical planning is generalized planning (GP), which focuses on finding a generalized solution for a (finite or infinite) set of numeric or classical planning problems that share the same set of actions but have different sets of propositional or numeric variables, initial states, and goal formulas [13, 27]. A number of GP problems enjoy the following characteristics: (1) the planning problems have the same set of propositional and numeric variables and the same goal formula; (2) the sole difference among planning problems is their initial states; and (3) the set of planning problems is infinite. The majority of research efforts [6, 14, 15, 17, 28, 30] investigate GP with the above characteristics, which is the focus of this paper. This type of GP can be formalized as numeric planning except that we use an initial formula that captures a possibly infinite number of initial states.

One of the effective means to tackle GP is to transform a GP problem into an abstract planning problem, which can be easily solved. The solution to the original GP problem is then created based on the solution to the abstract problem. Srivastava et al. [29] proposed an abstract model for GP, namely *qualitative numeric planning (QNP)*, a class of numeric planning with the following restrictions: (1) each numeric variable has a non-negative value; (2) actions have nondeterministic effects and can change the values of variables by any positive amount; and (3) initial states, goal states and preconditions of actions are expressed by simple formulas. A simple formula is the Boolean combination of propositional literals (p or $\neg p$) and simple numeric literals ($v = 0$ or $v > 0$). It is proved that QNP is decidable, more precisely, EXPTIME-Complete [5]. By using concept language [2], many GP problems can be cast as a QNP problem [4].

However, QNP lacks rich expressiveness to formalize some GP problems due to only allowing simple formulas and not supporting conditional effects. To address this deficit, Lin et al. [20] proposed another abstract model for GP problems, namely *generalized linear integer numeric planning (GLINP)*. The initial states, goal states and preconditions of actions in GLINP are described in linear integer arithmetic (LIA) and the conditional effects of actions are permitted. Compared to QNP, GLINP is more expressive to formalize a greater variety of GP problems. Additionaly, Lin et al. [20] developed an inductive approach to creating a planning program for GLINP. It first samples a finite set S of initial states. Then, it utilizes a numeric planner (*e.g.*, Metric-FF [12]) to produce the sequential plan for each sampled initial state $s \in S$. After that, it infers a skeleton of planning program expressed in a regular expression that incorporates the above sequential plans. The final planning program δ results from the skeleton by completing the missing conditions of loop structures and branch structures. Although the planning program δ derived by Lin et al.'s approach works for the sampled set S of initial states, there is no theoretical guarantee on the correctness of δ for the initial states not in S.

In this paper, we propose an automatic method to verify the correctness of planning programs for an infinite number of initial states. We design a translation from every program to a finite set of axioms formalized in linear integer arithmetic with uninterpreted predicate and function symbols (LIAUPF) so that the correctness verification of planning programs can be reduced to the entailment problem of the result of forgetting redundant symbols in such set of axioms. This entailment problem is in general undecidable and hence this reduction is not a decidable method. Therefore, we identify a class of planning programs in which the result of forgetting redundant symbols in the set of axioms is expressed in LIA, a decidable fragment of LIAUPF. As a result, the correctness verification of this class of programs becomes decidable.

2 PRELIMINARIES

2.1 LIAUPF

Let $\mathcal{Z}$ be the set of integers and $\mathcal{B}$ the set of $\{\mathbf{T}, \mathbf{F}\}$. Throughout this paper, we fix $\mathcal{F}$ a set of function symbols and $\mathcal{Q}$ a set of predicate symbols. A nullary function symbol is called a numeric symbol and a nullary predicate symbol is called a propositional symbol. The sets of *terms* (TermUPF) and *formulas* (FormUPF) of LIAUPF are defined by the following grammar:

$$e \in \texttt{TermUPF} :: c \mid x \mid e + e \mid f(e, \cdots, e)$$

$$\phi \in \texttt{FormUPF} :: \top \mid \bot \mid e = e \mid e < e \mid q(e, \cdots, e) \mid \neg\phi \mid \phi \wedge \phi \mid \exists x \phi$$

where $c \in \mathcal{Z}$, $f \in \mathcal{F}$, $q \in \mathcal{Q}$ and x is a variable.

We use $\exists \Omega \phi$ for $\exists x_1 \cdots \exists x_n \phi$ where $\Omega = \{x_1, \cdots, x_n\}$. The formula $\phi_1 \vee \phi_2$ is the shorthand for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi_1 \rightarrow \phi_2$ for $\neg\phi_1 \vee \phi_2$, $\phi_1 \leftrightarrow \phi_2$ for $(\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$, $\forall x \phi$ for $\neg\exists x(\neg\phi)$, and $e_1 \leq e_2$ for $e_1 = e_2 \vee e_1 < e_2$.

A formula ϕ or a term e is *closed* if it contains no free occurrence of a variable. A *sentence* is a closed formula. A *theory* is a set of sentences. We use $\mathcal{F}(\Phi)$ for the set of function symbols occurring in the theory Φ and $\mathcal{Q}(\Phi)$ for the set of predicate symbols.

The domain of each structure M of LIAUPF is the set $\mathcal{Z}$ of integers. Every m-ary function symbol f is interpreted as an m-ary integer function $f^M : \mathcal{Z}^m \rightarrow \mathcal{Z}$. Every m-ary predicate symbol q is interpreted as an m-ary relation over integers $q^M \subseteq \mathcal{Z}^m$. The addition function, the ordering relation and the equality relation are interpreted as usual. A structure M *satisfies* a sentence ϕ, if $M \models \phi$. A structure M *satisfies* a theory Φ, if $M \models \phi$ for every $\phi \in \Phi$. A set Φ of sentences *entails* a sentence ψ, denoted by $\Phi \models \psi$, if $M \models \psi$ for every structure M s.t. $M \models \Phi$.

As a corollary of Gödel first incompleteness theorem [11], the entailment problem of LIAUPF is undecidable. However, the entailment problems of the following two fragments of LIAUPF are decidable: (1) LIA: every function and predicate symbol is nullary, and (2) qf-LIA: a fragment of LIA where every formula is quantifier-free. We use Term for the set of term of LIA, Form for the set of LIA-formulas, and qf-Form for the set of qf-LIA-formulas.

2.2 Forgetting in LIAUPF

The concept of forgetting dates back to [7], who first considered forgetting propositional symbols in a propositional formula. Lin and Reiter [19] extended the above idea to forgetting predicate symbols in first-order logic. We hereafter give a model-theoretic definition of forgetting function and predicate symbols in LIAUPF.

Let $\Omega \subseteq \mathcal{F} \cup \mathcal{Q}$. Two structures M and M' are equivalent except on Ω, denoted by $M \sim_\Omega M'$, if they agree on everything except possibly on the interpretation of Ω.

Definition 1. Let M be a structure, Φ a theory and $\Omega \subseteq \mathcal{F} \cup \mathcal{Q}$. We say a theory, denoted by $\tilde{\exists}\Omega.\Phi$, is the result of forgetting Ω in Φ, if M' is a model of $\tilde{\exists}\Omega.\Phi$ iff there is a model M of Φ s.t. $M \sim_\Omega M'$.

For simplicity, we use $\tilde{\exists} f.\Phi$ for the result of forgetting a function symbol f in Φ and $\tilde{\exists} q.\Phi$ for the result of forgetting a predicate symbol q in Φ.

For all queries that do not mention Ω, the original theory Φ and the new one $\tilde{\exists}\Omega.\Phi$ are equivalent.

Proposition 1 ([19]). *For any sentence ϕ s.t. $\Omega \notin \mathcal{F}(\phi) \cup \mathcal{Q}(\phi)$, $\Phi \models \phi$ iff $\tilde{\exists}\Omega.\Phi \models \phi$.*

The result of forgetting a propositional symbol in a sentence ϕ is the disjunction of the positive restriction $\phi(p \leftarrow \top)$ of ϕ w.r.t. p and the negative restriction $\phi(p \leftarrow \bot)$ where $\phi(p \leftarrow \varphi)$ denotes the formula results from ϕ by replacing every occurrence of p by φ.

Proposition 2 ([19]). *Let p be a propositional symbol and ϕ a sentence. Then, $\tilde{\exists} p.\phi \equiv \phi(p \leftarrow \top) \vee \phi(p \leftarrow \bot)$.*

Forgetting a numeric symbol in a sentence ϕ can be accomplished by using an existential quantification over a new variable x.

Proposition 3 ([19]). *Let f be a numeric symbol, ϕ a sentence and x a new variable not in ϕ. Then, $\tilde{\exists} f.\phi \equiv \exists x.\phi(f \leftarrow x)$ where $\phi(f \leftarrow x)$ denotes the formula results from ϕ by replacing every occurrence of f by x.*

The above two propositions implies that forgetting propositional and numeric symbols in LIA-formulas is still LIA-definable.

2.3 Generalized Linear Integer Numeric Planning

Definition 2. A LINP domain $\mathcal{D}$ is a tuple $\langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$ where

- $\mathcal{P}$: a finite set of propositional symbols[1];
- $\mathcal{V}$: a finite set of numeric symbols;
- $\mathcal{A}$: a finite set of actions defined by a pair $\langle \mathtt{pre}, \mathtt{eff} \rangle$ where $\mathtt{pre} \in \mathtt{qf\text{-}Form}$[2] denotes the precondition and $\mathtt{eff}$ is a finite set of propositional and numeric effects.

A *state* s is a simplified model that interprets every propositional symbol p as $p^s \in \mathcal{B}$ and every numeric symbol v as $v^s \in \mathcal{Z}$. Given a state s, we evaluate a term e into an integer e^s to which the expression simplifies when substituting every numeric variable v with their respective value v^s. The Boolean value ϕ^s of a formula ϕ can be determined in a similar way.

A *propositional effect* is a tuple $\langle \psi, p, \phi \rangle$ where $\psi, \phi \in \mathtt{qf\text{-}Form}$ and $p \in \mathcal{P}$. Intuitively, it means that if ψ holds in a state s, the Boolean value of p becomes ϕ^s after performing the action; otherwise, it remains unchanged. A *numeric effect* is a tuple $\langle \psi, v, e \rangle$ where $\psi \in \mathtt{qf\text{-}Form}$, $v \in \mathcal{V}$ and $e \in \mathtt{Term}$. The meaning of numeric effects is similar to that of propositional effects.

Given an action a, we use $\mathtt{pre}(a)$ for its precondition and $\mathtt{eff}(a)$ for the set of its effects. We require that every action a is not self-contradictory, that is, for every state s and every two propositional effects (ψ_1, p, ϕ_1) and (ψ_2, p, ϕ_2) of a, it is impossible that ψ_1 and ψ_2 holds in s but $\phi_1^s \neq \phi_2^s$, and similarly for numeric effects. An effect is *unconditional*, if $\psi = \top$. An action a is *unconditional*, if all of its effect is unconditional; otherwise, it is *conditional*.

The *successor state* $\tau(s, a)$ of applying an action a over s is defined as follows:

$$p^{\tau(s,a)} = \begin{cases} \phi^s, & \text{if } \langle \psi, p, \phi \rangle \in \mathtt{eff}(a) \text{ and } \psi^s = \mathbf{T}; \\ p^s, & \text{otherwise.} \end{cases}$$

$$v^{\tau(s,a)} = \begin{cases} e^s, & \text{if } \langle \psi, v, e \rangle \in \mathtt{eff}(a) \text{ and } \psi^s = \mathbf{T}; \\ v^s, & \text{otherwise.} \end{cases}$$

An action a is *executable* in a state s, if $s \models \mathtt{pre}(a)$. We remark that $\tau(s, a)$ is well-defined even if a is not executable in s. The *resulting state* of performing a finite sequence $[a_1, \cdots, a_n]$ of actions on s is recursively defined by $\tau(s, [a_1, \cdots, a_n]) = \tau(\tau(s, [a_1, \cdots, a_{n-1}]), a_n)$ and $\tau(s, \varepsilon) = s$ where ε is an empty sequence. A sequence $[a_1, a_2, \cdots]$ of actions is *executable* in a state s, if $s \models \mathtt{pre}(a_1)$ and $\tau(s, [a_1 \cdots a_i]) \models \mathtt{pre}(a_{i+1})$ for $i \geq 1$.

We hereafter provide the definition of GLINP problems.

Definition 3. A generalized LINP (GLINP) problem Σ is a tuple $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where

- $\mathcal{D}$: a LINP domain $\langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$;
- $\mathcal{I} \in \mathtt{qf\text{-}Form}$: a formula denoting a set of possibly infinitely many initial states;
- $\mathcal{G} \in \mathtt{qf\text{-}Form}$: a formula denoting a set of possibly infinitely many goal states.

The solution to GLINP problems is a form of *planning programs*.

Definition 4. The set of planning programs ($\mathtt{Prog}$) for a LINP domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$ is recursively defined by

$$\delta \in \mathtt{Prog} :: \varepsilon \mid a \mid \delta; \delta \mid \textbf{if } \phi \textbf{ then } \delta \textbf{ else } \delta \textbf{ fi} \mid \textbf{while } \phi \textbf{ do } \delta \textbf{ od}$$

where $a \in \mathcal{A}$ and $\phi \in \mathtt{qf\text{-}Form}$.

The construct $\delta_1; \delta_2$ is the sequential structure; **if** ϕ **then** δ_1 **else** δ_2 **fi** is the branch structure and **while** ϕ **do** δ **od** is the loop structure. We say ϕ is the *condition* of the branch structure **if** ϕ **then** δ_1 **else** δ_2 **fi**. Likewise, ϕ is the condition of the loop structure **while** ϕ **do** δ **od**.

We use $\#(\delta)$ for the depth of δ, and $\Theta(s, \delta)$ for the action sequence of executing δ in s. The above two notations was defined in [20]. A loop structure is *simple* if its depth is 1; otherwise, it is *nested*.

The solution to a GLINP problem is a program satisfying the following three properties.

Definition 5. Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$ be a LINP domain, δ a program for $\mathcal{D}$, and s a state. The program δ is

- *terminating* in s, iff $\Theta(s, \delta)$ is finite;
- *executable* in s, iff $\Theta(s, \delta)$ is executable in s;
- *ϕ-reaching* in s, iff δ is terminating and executable in s only if $\tau(s, \delta) \models \phi$.

Definition 6. Let $\Sigma = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a GLINP problem. A program δ is a solution to Σ, if δ is terminating, executable and $\mathcal{G}$-reaching in every initial state.

Example 1. We illustrate the GLINP problem with the TestOn problem. Initially, there are two towers of blocks. The first tower contains a block x while the second one contains another block y. The goal is to put block x on y. The numeric symbols nx and ny denote the number of blocks above x and y, respectively. The propositional symbol $onxy$ indicates block x is on y. The actions $UnstackX$ pick the top block of the first tower and put it aside on the table, and $UnstackY$ is similar for the second tower. The action $StackXOnY$ means picking block x on y if no blocks are above x and y.

- $\mathcal{P} : \{onxy\}$ and $\mathcal{V} : \{nx, ny\}$;
- $\mathcal{A} : \{UnstackX, UnstackY, StackXOnY\}$;
- $\mathtt{pre}(UnstackX) : nx > 0$;
- $\mathtt{eff}(UnstackX) : \{\langle \top, nx, nx - 1 \rangle\}$;
- $\mathtt{pre}(UnstackY) : ny > 0$;
- $\mathtt{eff}(UnstackY) : \{\langle \top, ny, ny - 1 \rangle\}$;
- $\mathtt{pre}(StackXOnY) : nx = 0 \wedge ny = 0 \wedge \neg onxy$;
- $\mathtt{eff}(StackXOnY) : \{\langle \top, onxy, \top \rangle, \langle \top, ny, ny + 1 \rangle\}$;
- $\mathcal{I} : nx > 0 \wedge ny > 0 \wedge \neg onxy$ and $\mathcal{G} : onxy$.

[1] To distinguish variable symbol in logic and variable in planning, we use the terminology "variable symbol" for the variable symbol in logic, "propositional symbol" for propositional variable in planning, "numeric symbol" for numeric variable in planning.
[2] In the remaining of this paper, we assume every formula $\phi \in \mathtt{qf\text{-}Form}$ is over propositional symbols in $\mathcal{P}$ and numeric symbols in $\mathcal{V}$, and every term $e \in \mathtt{Term}$ is over numeric symbols in $\mathcal{V}$.

The solution δ to the TestOn problem is:
while $ny \neq 0$ **do** $UnstackY$ **od**;
while $nx \neq 0$ **do** $UnstackX$ **od**;
$StackXOnY$ □

3 CORRECTNESS VERIFICATION

In this section, we develop a method to verify the correctness of the planning program. The main insight behind our method is as follows: We first transform a planning program into a trace axiom that is finitely represented in LIAUPF, which records the value of every propositional and numeric symbol during the execution of a planning program (§ 3.1). With the help of trace axioms, we reduce the verification of three properties of planning programs to the entailment problem of LIAUPF (§ 3.2).

3.1 Trace Axioms

We first introduce some notations, followed by the construction of trace axioms by adapting the translation method proposed in [18].

In order to indicate different positions of the same subprograms, we mark different instances of the same subprograms with superscripts. For example, the program $\delta = \gamma;\gamma$ where $\gamma =$ **if** ϕ **then** a **else** b **fi** becomes $\delta = \gamma^1;\gamma^2$ where $\gamma^1 =$ **if** ϕ **then** a^1 **else** b^1 **fi** and $\gamma^2 =$ **if** ϕ **then** a^2 **else** b^2 **fi**. In the remaining of this paper, we assume that every planning program is with superscripts.

For each $p \in \mathcal{P}$ and each program δ, we consider two versions p_δ and p'_δ. The two propositional symbols p_δ and p'_δ are called an *input* and an *output*, respectively, of δ that are used to represent the value of p before performing the program δ and the value of p after. Similarly, for each $v \in \mathcal{V}$, we also use two versions v_δ and v'_δ. Given a term e and a formula ϕ, e_δ, e'_δ, ϕ_δ and ϕ'_δ can be defined in a recursive way, *e.g.*, $(\neg\phi)_\delta$ is $\neg\phi_\delta$, $(\phi \wedge \psi)'_\delta$ is $\phi'_\delta \wedge \psi'_\delta$.

To distinguish the value of a term e and a formula ϕ at each iteration of executing a loop structure δ, we use $e[k]$ and $\phi[k]$ to record the value of e and ϕ after the body of δ has been executed k times, respectively. They are recursively defined as follows:

- $c[k]$ is c, $x[k]$ is x, $(e_1 + e_2)[k]$ is $e_1[k] + e_2[k]$
- $f(e_1, \cdots, e_m)[k]$ is $f(e_1[k], \cdots, e_m[k], k)$ where f is a function symbol,
- $q(e_1, \cdots, e_m)[k]$ is $q(e_1[k], \cdots, e_m[k], k)$ where q is a predicate symbol,
- $(e_1 = e_2)[k]$ is $e_1[k] = e_2[k]$, $(e_1 < e_2)[k]$ is $e_1[k] < e_2[k]$,
- $(\neg\phi_1)[k]$ is $\neg\phi_1[k]$, $(\phi_1 \wedge \phi_2)[k]$ is $\phi_1[k] \wedge \phi_2[k]$,
- $(\exists x\phi_1)[k]$ is $\exists x\phi_1[k]$.

Let δ_1 and δ_2 be two programs. We use $\mathcal{E}^{io}_{\delta_1,\delta_2}$ for the set $\{v_{\delta_1} = v'_{\delta_2} \mid v \in \mathcal{V}\} \cup \{p_{\delta_1} \leftrightarrow p'_{\delta_2} \mid p \in \mathcal{P}\}$. Intuitively, it says that each input of δ_1 has the same value as the corresponding output of δ_2. The sets $\mathcal{E}^{ii}_{\delta_1,\delta_2}$, $\mathcal{E}^{oi}_{\delta_1,\delta_2}$ and $\mathcal{E}^{oo}_{\delta_1,\delta_2}$ can be similarly defined.

Definition 7. Let δ be a planning program. The trace axiom $\mathcal{T}(\delta)$ of δ is a set of LIAUPF-formulas defined as follows:

- $\mathcal{T}(\varepsilon) = \mathcal{E}^{io}_{\delta,\delta}$;
- $\mathcal{T}(a) = \{(\mathtt{pre}(a))_\delta\} \cup$
  $\{\psi_\delta \rightarrow (p'_\delta \leftrightarrow \phi_\delta) \mid (\psi, p, \phi) \in \mathtt{eff}(a)\} \cup$
  $\{\psi_\delta \rightarrow (v'_\delta = e_\delta) \mid (\psi, v, e) \in \mathtt{eff}(a)\} \cup$
  $\{\neg(\bigvee_{(\psi,p,\phi)\in\mathtt{eff}(a)} \psi_\delta) \rightarrow (p'_\delta \leftrightarrow p_\delta) \mid p \in \mathcal{P}\} \cup$
  $\{\neg(\bigvee_{(\psi,v,e)\in\mathtt{eff}(a)} \psi_\delta) \rightarrow (v'_\delta = v_\delta) \mid v \in \mathcal{V}\}$;
- $\mathcal{T}(\delta_1;\delta_2) = \mathcal{T}(\delta_1) \cup \mathcal{T}(\delta_2) \cup \mathcal{E}^{ii}_{\delta,\delta_1} \cup \mathcal{E}^{oi}_{\delta_1,\delta_2} \cup \mathcal{E}^{oo}_{\delta,\delta_2}$;
- $\mathcal{T}($**if** ϕ **then** δ_1 **else** δ_2 **fi**$) =$
  $\{\phi_\delta \rightarrow \psi \mid \psi \in \mathcal{T}(\delta_1) \cup \mathcal{E}^{ii}_{\delta,\delta_1} \cup \mathcal{E}^{oo}_{\delta,\delta_1}\} \cup$
  $\{\neg\phi_\delta \rightarrow \psi \mid \psi \in \mathcal{T}(\delta_2) \cup \mathcal{E}^{ii}_{\delta,\delta_2} \cup \mathcal{E}^{oo}_{\delta,\delta_2}\}$;
- $\mathcal{T}($**while** ϕ **do** δ_1 **od**$) =$
  $\{n \geq 0 \wedge \neg(\phi_{\delta_1})[k](k \leftarrow n) \wedge \forall 0 \leq k < n.\phi_{\delta_1}[k]\} \cup$
  $\{\forall 0 \leq k < n.\psi[k] \mid \psi \in \mathcal{T}(\delta_1)\} \cup$
  $\{p_\delta \leftrightarrow p_{\delta_1}(0) \mid p \in \mathcal{P}\} \cup \{p'_\delta \leftrightarrow p_{\delta_1}(n) \mid p \in \mathcal{P}\} \cup$
  $\{v_\delta = v_{\delta_1}(0) \mid v \in \mathcal{V}\} \cup \{v'_\delta = v_{\delta_1}(n) \mid v \in \mathcal{V}\} \cup$
  $\{\forall 0 \leq k < n.p_{\delta_1}(k+1) \leftrightarrow p'_{\delta_1}(k) \mid p \in \mathcal{P}\} \cup$
  $\{\forall 0 \leq k < n.v_{\delta_1}(k+1) = v'_{\delta_1}(k) \mid v \in \mathcal{V}\}$
  where n is a new numeric symbol not in $\mathcal{T}(\delta_1)$, k is a new variable not in $\mathcal{T}(\delta_1)$, and $\forall 0 \leq k < n.\varphi$ is the shorthand for $\forall k[(0 \leq k \wedge k < n) \rightarrow \varphi]$.

When δ is an empty plan, $\mathcal{T}(\varepsilon)$ indicates that the value of each symbol is left unchanged. The axiom $\mathcal{T}(a)$ states that the value of each symbol varies according to the effect of a. It consists of the precondition, the effect axioms and the frame axioms of a. The effect axioms indicate that the value of p (resp. v) becomes $\phi(s)$ (resp. $e(s)$) if the condition ψ holds in the state s. The frame axioms stipulate that the value of p (resp. v) remains unchanged if no condition ψ s.t. $(\psi, p, \phi) \in \mathtt{eff}(a)$ (resp. $(\psi, v, e) \in \mathtt{eff}(a)$) holds.

The formula $\mathcal{T}(\delta_1;\delta_2)$ is constructed from $\mathcal{T}(\delta_1)$ and $\mathcal{T}(\delta_2)$ by directly linking the inputs of δ to the inputs of δ_1, linking the outputs of δ_1 to the inputs of δ_2, and linking the outputs of δ_2 to the outputs of δ. The formula $\mathcal{T}($**if** ϕ **then** δ_1 **else** δ_2 **fi**$)$ has the following meaning. If the condition ϕ holds, then it executes the first branch δ_1, and hence each input (resp. output) of δ has the same value as the corresponding input (resp. output) of δ_1. Otherwise, it enters the other branch δ_2, and the trace axiom for the other branch are constructed as the same as that for the first one.

When δ is a loop structure **while** ϕ **do** δ_1 **od**, we introduce a new numeric symbol n that stands for the number of iterations of δ_1. Each predicate symbol occurring in the subprogram δ_1 is extended to a predicate symbol with an extra argument k, which denotes the Boolean value of the predicate symbol after δ_1 has been executed k times. The treatment of each function symbol is similar. During the iteration of δ_1, each input of the $(k+1)$-th iteration of δ_1 has the same value as the corresponding output of the k-th iteration. At the end of the loop structure, the values of each output of δ and the corresponding input of the n-th iteration of δ_1 are exactly the same.

In contrast to the trace axiom proposed in [18], ours is more complicated. Lin [18] focuses on imperative programs, which are slightly different from planning programs we consider in this paper. The base construct of imperative programs is variable assignment statement, which can be defined as an action with valid precondition and an unconditional effect. (1) Imperative programs generally run in computer systems while planning programs aim to control robots in real-world scenarios. Since real-world scenarios is more complicated than the runtime environment of computer systems, it is essential to formulate conditions in which an action can be performed. Moreover, it is essential to take into account the executability condition of programs. Neither the action precondition nor the program executability condition is considered in [18]. (2) Conditional effects [22] allows actions with context-dependent effects. Some efforts to

compile away conditional effects have been made in classical planning [1, 10, 16]. It, however, was shown that the method to compile away conditional effects introduces exponentially many auxiliary actions in the worst case [21]. Conditional effects provide a more convenient and succinct way to model complex AI scenarios. Our trace axiom exactly describe the change due to an action as the combination of the effect axiom and the frame axiom while Lin [18] simply uses an equation to represent a variable assignment statement.

Example 2. The program δ shown in Example 1 can be divided into three slices: δ_1, δ_2 and δ_3 where δ_1 is the loop structure **while** $ny \neq 0$ **do** $UnstackY$ **od**, δ_2 is **while** $nx \neq 0$ **do** $UnstackX$ **od** and δ_3 is action $StackXOnY$.

We first construct the trace axiom $\mathcal{T}(\delta_1)$ for δ_1. For simplicity, we use a_1 for the action $UnstackY$ occurring in δ_1. The trace axiom $\mathcal{T}(a_1)$ contains the following four sentences: (1) $ny_{a_1} > 0$, (2)$\top \rightarrow (ny'_{a_1} = ny_{a_1} - 1)$, (3) $\top \rightarrow (nx'_{a_1} = nx_{a_1})$, and (4) $\top \rightarrow (onxy'_{a_1} \leftrightarrow onxy_{a_1})$. It can be simplified as the following equivalent axiom: $\{ny_{a_1} > 0, ny'_{a_1} = ny_{a_1} - 1, nx'_{a_1} = nx_{a_1}, onxy'_{a_1} \leftrightarrow onxy_{a_1}\}$.

We introduce a new numeric symbol n_1 to denote the total number of iterations of executing the body $UnstackY$. We extend the symbols $onxy_{a_1}$, $onxy'_{a_1}$, nx_{a_1}, nx'_{a_1}, ny_{a_1} and ny'_{a_1} with an additional parameter k to represent the value of the symbols after loop body has been executed k times.

So $\mathcal{T}(\delta_1)$ is equivalent to

$$\{n_1 \geq 0 \wedge ny_{a_1}(n_1) = 0 \wedge \forall 0 \leq k < n_1.[ny_{a_1}(k) \neq 0]\} \cup$$
$$\{\forall 0 \leq k < n_1.[ny_{a_1}(k) > 0 \wedge ny'_{a_1}(k) = ny_{a_1}(k) - 1 \wedge nx'_{a_1}(k) = nx_{a_1}(k) \wedge onxy'_{a_1}(k) \leftrightarrow onxy_{a_1}(k)]\} \cup$$
$$\{onxy_{\delta_1} \leftrightarrow onxy_{a_1}(0), onxy'_{\delta_1} \leftrightarrow onxy_{a_1}(n_1), nx_{\delta_1} = nx_{a_1}(0), nx'_{\delta_1} = nx_{a_1}(n_1), ny_{\delta_1} = ny_{a_1}(0), ny'_{\delta_1} = ny_{a_1}(n_1)\} \cup$$
$$\{\forall 0 \leq k < n_1.[onxy_{a_1}(k+1) \leftrightarrow onxy'_{a_1}(k) \wedge nx_{a_1}(k+1) = nx'_{a_1}(k) \wedge ny_{a_1}(k+1) = ny'_{a_1}(k)]\}.$$

Similarly, we use a_2 for the primitive action $UnstackX$ occurring in δ_2 and a new numeric symbol n_2 for the total number of iterations of executing the body $UnstackY$. Hence, $\mathcal{T}(\delta_2)$ is equivalent to

$$\{n_2 \geq 0 \wedge nx_{a_2}(n_2) = 0 \wedge \forall 0 \leq k < n_2.[nx_{a_2}(k) \neq 0]\} \cup$$
$$\{\forall 0 \leq k < n_2.[nx_{a_2}(k) > 0 \wedge nx'_{a_2}(k) = nx_{a_2}(k) - 1 \wedge ny'_{a_2}(k) = ny_{a_2}(k) \wedge onxy'_{a_2}(k) \leftrightarrow onxy_{a_2}(k)]\} \cup$$
$$\{onxy_{\delta_2} \leftrightarrow onxy_{a_2}(0), onxy'_{\delta_2} \leftrightarrow onxy_{a_2}(n_2), nx_{\delta_2} = nx_{a_2}(0), nx'_{\delta_2} = nx_{a_2}(n_2), ny_{\delta_2} = ny_{a_2}(0), ny'_{\delta_2} = ny_{a_2}(n_2)\} \cup$$
$$\{\forall 0 \leq k < n_2.[onxy_{a_2}(k+1) \leftrightarrow onxy'_{a_2}(k) \wedge nx_{a_2}(k+1) = nx'_{a_2}(k) \wedge ny_{a_2}(k+1) = ny'_{a_2}(k)]\}.$$

The trace axiom $\mathcal{T}(\delta_3)$ for δ_3 is equivalent to

$$\{nx_{\delta_3} = 0 \wedge ny_{\delta_3} = 0 \wedge \neg onxy_{\delta_3}, onxy'_{\delta_3}, ny'_{\delta_3} = ny_{\delta_3} + 1, nx'_{\delta_3} = nx_{\delta_3}\}.$$

Finally, we get:

$$\mathcal{E}^{ii}_{\delta,\delta_1} : \{onxy_{\delta} \leftrightarrow onxy_{\delta_1}, nx_{\delta} = nx_{\delta_1}, ny_{\delta} = ny_{\delta_1}\}$$
$$\mathcal{E}^{oi}_{\delta_1,\delta_2} : \{onxy'_{\delta_1} \leftrightarrow onxy_{\delta_2}, nx'_{\delta_1} = nx_{\delta_2}, ny'_{\delta_1} = ny_{\delta_2}\}$$
$$\mathcal{E}^{oi}_{\delta_2,\delta_3} : \{onxy'_{\delta_2} \leftrightarrow onxy_{\delta_3}, nx'_{\delta_2} = nx_{\delta_3}, ny'_{\delta_2} = ny_{\delta_3}\}$$
$$\mathcal{E}^{oo}_{\delta,\delta_3} : \{onxy'_{\delta} \leftrightarrow onxy'_{\delta_3}, nx'_{\delta} = nx'_{\delta_3}, ny'_{\delta} = ny'_{\delta_3}\}.$$

In summary, we have $\mathcal{T}(\delta_1;\delta_2;\delta_3) = \mathcal{T}(\delta_1) \cup \mathcal{T}(\delta_2) \cup \mathcal{T}(\delta_3) \cup \mathcal{E}^{ii}_{\delta,\delta_1} \cup \mathcal{E}^{oi}_{\delta_1,\delta_2} \cup \mathcal{E}^{oi}_{\delta_2,\delta_3} \cup \mathcal{E}^{oo}_{\delta,\delta_3}$. □

We provide the correspondence between models of trace axioms and action sequences of executing planning programs. Given a model M of $\mathcal{T}(\delta)$, we say a state s is an input-projection of M, if $(p_\delta)^M = p^s$ for $p \in \mathcal{P}$ and $(v_\delta)^M = v^s$ for $v \in \mathcal{V}$, and s is an output-projection of M, if $(p'_\delta)^M = p^s$ for $p \in \mathcal{P}$ and $(v'_\delta)^M = v^s$ for $v \in \mathcal{V}$.

Proposition 4. *Let δ be a planning program and $\mathcal{T}(\delta)$ the trace axiom of δ. Then, the following hold*

- *if M is a model of $\mathcal{T}(\delta)$ and s_0 is the input-projection of M, then $\Theta(s_0, \delta)$ is finite and executable in s_0, and $\tau(s_0, \delta)$ is the output-projection of M;*
- *if $\Theta(s_0, \delta)$ is finite and executable in s_0, then there is a model M of $\mathcal{T}(\delta)$ s.t. s_0 is the input-projection of M and $\tau(s_0, \delta)$ is the output-projection of M.*

3.2 Verifying Goal-Reaching, Termination and Executability Properties

Based on trace axioms, we are ready to verify goal-reaching, termination and executability properties of programs. The following theorem states that verifying goal-reaching property of a planning program can be reduced to deciding if the initial formula entails the goal formula under the background theory defined by the trace axiom.

Theorem 1. *Let $\Sigma = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a GLINP problem and δ a planning program. Then, $\mathcal{T}(\delta) \models \mathcal{I}_\delta \rightarrow \mathcal{G}'_\delta$ iff δ is $\mathcal{G}$-reaching in every initial state.*

We can verify termination and executability properties of planning programs together via forgetting. By Proposition 4, each model M of trace axiom $\mathcal{T}(\delta)$ with the input-projection s_0 corresponds to a finite action sequence $\Theta(s_0, \delta)$ which is executable in s_0. It is observed that the set of input-projections of all models of $\mathcal{T}(\delta)$ is the set of states in which δ is terminating and executable. We can obtain a sentence that exactly captures the set of input-projections by resorting to forgetting every predicate and function symbol except $\mathcal{P}_\delta \cup \mathcal{V}_\delta$ in $\mathcal{T}(\delta)$. Thus, we get the sufficient and necessary condition under which a program δ is terminating and executable in every initial state.

Theorem 2. *Let $\Sigma = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a GLINP problem and δ a planning program. Let ψ be the formula equivalent to $\tilde{\exists}\overline{\mathcal{P}}_\delta \tilde{\exists}\overline{\mathcal{V}}_\delta.\mathcal{T}(\delta)$ where $\overline{\mathcal{P}}_\delta = \mathcal{Q}(\mathcal{T}(\delta)) \setminus \mathcal{P}_\delta$ and $\overline{\mathcal{V}}_\delta = \mathcal{F}(\mathcal{T}(\delta)) \setminus \mathcal{V}_\delta$. Then, $\mathcal{I}_\delta \models \psi$ iff δ is terminating and executable in every initial state.*

4 A DECIDABLE FRAGMENT

The axiom $\mathcal{T}(\delta)$ is represented in LIAUPF whose entailment problem is undecidable. Therefore, our approach is not a decidable method to verify correctness of arbitrary programs. In this section, we identify a decidable class of programs, namely *unconditional pseudo-primitive programs*.

When verifying goal-reaching, termination and executability properties, by Theorems 1 and 2, we reason about only the input and output symbols of programs, and do not consider the following redundant symbols: (1) the intermediate predicate and function symbols of subprograms, and (2) the newly introduced numeric symbol denoting the number of iterations in loop structures. In this case, an unconditional pseudo-primitive program δ can be considered as a primitive action with only unconditional effects. An unconditional pseudo-primitive program δ has similar notations p-pre(δ) (Definition 10) and p-eff(δ) (Definition 9) to actions. The

formula $\texttt{p-pre}(\delta)$ represents the termination and executability condition of δ and $\texttt{p-eff}(\delta)$ denotes the effect that exactly captures the change of values of each propositional and numeric symbol by performing δ. In addition, forgetting redundant symbols in the trace axiom of δ can be simplified as the union of its termination and executability condition and its effect, and is LIA-definable (Theorem 3). Hence, the correctness verification of unconditional pseudo-primitive programs becomes decidable (Corollary 1).

Let δ be a program. We say a propositional symbol p is *static* in δ, iff for every state s s.t. δ is terminating in s, we have $p^{\tau(s,\delta)} = p^s$. We say a numeric symbol v is d-incremental in δ where $d \in \mathcal{Z}$, iff for every state s s.t. δ is terminating in s, we have $v^{\tau(s,\delta)} = v^s + d$.

Definition 8. We say a program δ is *unconditional pseudo-primitive*, if one of the following conditions hold:

- $\delta = \varepsilon$;
- $\delta = a$ where a is an unconditional action;
- $\delta = \delta_1; \delta_2$ where δ_1 and δ_2 are unconditional pseudo-primitive;
- $\delta =$ **while** ϕ **do** δ_1 **od** and
  - the body δ_1 is a sequence of unconditional actions; and
  - every propositional symbol p is static in δ_1; and
  - ϕ is $c_0 + \sum_{u \in \mathcal{V}'} c_u \cdot u \neq 0$ where $\mathcal{V}' \subseteq \mathcal{V}$ and $c_0, c_u \in \mathcal{Z}$; and
  - there is exactly one $w \in \mathcal{V}'$ s.t. w is d_w-incremental in δ_1, and $c_w, d_w \in \{-1, 1\}$; and
  - every numeric symbol $u \in \mathcal{V}' \setminus \{w\}$ is 0-incremental in δ_1; and
  - every numeric symbol $v \in \mathcal{V} \setminus \mathcal{V}'$ is d_v-incremental in δ_1 where $d_v \in \mathcal{Z}$.

Unconditional pseudo-primitive programs have the following limitations: (1) every primitive action is unconditional; (2) no branch structure is permitted; (3) every loop structure has depth 1 and is subject to the restrictions illustrated in Definition 8. Every loop structure δ is associated with the condition of the form $c_0 + \sum_{u \in \mathcal{V}'} c_u \cdot u \neq 0$. In the condition, there is only one numeric symbol w such that it is 1- or -1-incremental in the loop body δ_1 and its coefficient c_w is either 1 or -1. Other numeric symbols u of $\mathcal{V}'$ remains unchanged by performing δ_1. Suppose that either $c_w = -1$ and w is 1-incremental in δ_1, or $c_w = 1$ and w is -1-incremental in δ_1. We let e denote the term $c_0 + \sum_{u \in \mathcal{V}'} c_u \cdot u$. It is easily verified that each iteration decreases the value of the term e by 1. If $e \geq 0$ is true initially, then the value of e will eventually reach to 0 and δ will come to an end. In this case, the number of iterations is e. If $e < 0$ holds initially, then e decreases forever and δ_1 executes indefinitely. In a word, the termination condition of δ is therefore $e \geq 0$. Similarly, in the two situations that $c_w = 1$ and w is 1-incremental in δ_1, and that $c_w = -1$ and w is -1-incremental in δ_1, the termination condition of δ is $e \geq 0$ where $e = -c_0 - \sum_{u \in \mathcal{V}'} c_u \cdot u$. The membership problem of unconditional pseudo-primitive programs is decidable.

We hereafter provide the effect of unconditional pseudo-primitive programs. A *symbol substitution* Ψ is a function that maps from $\mathcal{P}$ to qf-LIA-formulas and from $\mathcal{V}$ to LIA-terms. In fact, a symbol substitution Ψ represents a set of unconditional propositional and numeric effects. For example, $\Psi(p) = \phi$ corresponds to the effect $\langle \top, p, \phi \rangle$. Given a formula ϕ and a symbol substitution Ψ, we use $\phi[\Psi]$ to represent the result of simultaneously replacing every occurrence of $p \in \mathcal{P}$ and $v \in \mathcal{V}$ in ϕ by $\Psi(v)$ and $\Psi(p)$, respectively. Intuitively, $\phi[\Psi]$ is the weakest precondition of ϕ w.r.t. Ψ, that is, ϕ holds after applying the effect denoted by Ψ iff $\phi[\Psi]$ holds before.

The effect of an unconditional pseudo-primitive program δ is represented by a symbol substitution as $\texttt{p-eff}(\delta)$.

Definition 9. Let δ be an unconditional pseudo-primitive program. The effect of δ is recursively defined as follows:

- $\texttt{p-eff}(\varepsilon)(p) = p$ and $\texttt{p-eff}(\varepsilon)(v) = v$;
- $\texttt{p-eff}(a)(p) = \begin{cases} \phi, & \text{if } \langle \top, p, \phi \rangle \in \texttt{eff}(a); \\ p, & \text{otherwise.} \end{cases}$
- $\texttt{p-eff}(a)(v) = \begin{cases} e, & \text{if } \langle \top, v, e \rangle \in \texttt{eff}(a); \\ v, & \text{otherwise.} \end{cases}$
- $\texttt{p-eff}(\delta_1; \delta_2)(p) = \texttt{p-eff}(\delta_2)(p)[\texttt{p-eff}(\delta_1)]$;
- $\texttt{p-eff}(\delta_1; \delta_2)(v) = \texttt{p-eff}(\delta_2)(v)[\texttt{p-eff}(\delta_1)]$;
- $\texttt{p-eff}($**while** ϕ **do** δ_1 **od**$)(p) = p$;
- $\texttt{p-eff}($**while** ϕ **do** δ_1 **od**$)(v) = v + d_v \cdot e$ where
  - ϕ is $c_0 + \sum_{u \in \mathcal{V}'} c_u \cdot u \neq 0$;
  - v is d_v-incremental in δ_1;
  - e is $c_0 + \sum_{u \in \mathcal{V}'} c_u \cdot u$, if $c_w = -1$ and w is 1-incremental in δ_1, or $c_w = 1$ and w is -1-incremental in δ_1;
  - e is $-c_0 - \sum_{u \in \mathcal{V}'} c_u \cdot u$, if $c_w = 1$ and w is 1-incremental in δ_1, or $c_w = -1$ and w is -1-incremental in δ_1.

The notation $\texttt{p-eff}(\delta)(p) = \phi$ means that when the program δ is terminating, the Boolean value of p after the execution of δ equals that of ϕ before. The meaning of $\texttt{p-eff}(\delta)(v) = e$ is similar. The value of every propositional and numeric symbol remains unchanged in the empty plan. The effect $\texttt{p-eff}(a)$ of an unconditional action a corresponds to the original effect $\texttt{eff}(a)$ of a. As for the sequence structure $\delta_1; \delta_2$, the Boolean value of a propositional symbol p following the execution of the two subprograms δ_1 and δ_2 in succession is equivalent to that of $\texttt{p-eff}(\delta_2)(p)$ after executing δ_1. Again, p holds after performing $\delta_1; \delta_2$ iff the formula $\texttt{p-eff}(\delta_2)(p)[\texttt{p-eff}(\delta_1)]$ holds initially. The effect of a sequence structure for each numeric symbol is similar. If the loop structure **while** ϕ **do** δ_1 **od** is terminating, the value of every propositional symbol p remains unchanged after performing **while** ϕ **do** δ_1 **od** since p is static in the body δ_1. Meanwhile, the loop structure increases the value of every numeric symbol v by $d_v \cdot e$ in that v is d_v-incremental in δ_1 and e is the number of iterations of δ_1.

We hereafter present the termination and executability condition of unconditional pseudo-primitive programs.

Definition 10. Let δ be an unconditional pseudo-primitive program. The termination and executability condition of δ is recursively defined as follows:

- $\texttt{p-pre}(\varepsilon) = \top$ and $\texttt{p-pre}(a) = \texttt{pre}(a)$;
- $\texttt{p-pre}(\delta_1; \delta_2) = \texttt{p-pre}(\delta_1) \wedge \texttt{p-pre}(\delta_2)[\texttt{p-eff}(\delta_1)]$;
- $\texttt{p-pre}($**while** ϕ **do** δ_1 **od**$) =$
  $e \geq 0 \wedge \forall 0 \leq k < e(\phi \wedge \texttt{p-pre}(\delta_1))[\Psi]$ where
  - ϕ is defined as in Definition 8 and e is defined as in Definition 9;
  - $\Psi(p) = p$;
  - $\Psi(v) = v + d_v \cdot k$ where v is d_v-incremental in δ_1.

The empty plan always terminates and is executable. The termination and executability condition of an action is its precondition. The sequence structure $\delta_1; \delta_2$ terminates and is executable in a state

s iff δ_1 is terminating and executable in s, and then δ_2 is terminating and executable in $\tau(s, \delta_1)$. It implies that the conjunction of $\texttt{p-pre}(\delta_1)$ and $\texttt{p-pre}(\delta_2)[\texttt{p-eff}(\delta_1)]$ is the termination and executability condition of the sequence structure $\delta_1; \delta_2$. For the loop structure **while** ϕ **do** δ_1 **od**, the term e denotes the number of iterations of loop body. The loop structure is terminating and exeuctable iff both of the following are met: (1) the condition of the loop structure can be satisfied after a finite number of iterations, which is captured by $e \geq 0$; and (2) every iteration of δ_1 is terminating and executable, which is represented by $(\phi \wedge \texttt{p-pre}(\delta_1))[\Psi]$ where Ψ specifies the effect of the first k-th iteration of δ_1.

The trace axiom of an unconditional pseudo-primitive program δ can be simplified as the union of its termination and executability condition and its effect when reasoning about only inputs and outputs of δ.

Theorem 3. *Let δ be an unconditional pseudo-primitive program, $\overline{\mathcal{P}}_{\delta\cup\delta'} = \mathcal{Q}(\mathcal{T}(\delta)) \setminus (\mathcal{P}_\delta \cup \mathcal{P}'_\delta)$ and $\overline{\mathcal{V}}_{\delta\cup\delta'} = \mathcal{F}(\mathcal{T}(\delta)) \setminus (\mathcal{V}_\delta \cup \mathcal{V}'_\delta)$. Then, $\tilde{\exists}\overline{\mathcal{P}}_{\delta\cup\delta'}\tilde{\exists}\overline{\mathcal{V}}_{\delta\cup\delta'}.\mathcal{T}(\delta) \equiv \{(\texttt{p-pre}(\delta))_\delta\} \cup \{p'_\delta \leftrightarrow (\texttt{p-eff}(\delta)(p))_\delta \mid p \in \mathcal{P}\} \cup \{v'_\delta = (\texttt{p-eff}(\delta)(v))_\delta \mid v \in \mathcal{V}\}$.*

We remark that Definition 9 provides the effect of unconditional pseudo-primitive programs for the case where δ is terminating. When δ is non-terminating, its effect is undefined since the value of some symbols may be changed forever. However, Theorem 3 still holds for the case where δ is non-terminating. In this case, the termination and executability condition $\texttt{p-pre}(\delta)$ of δ is false and so is the trace axiom $\mathcal{T}(\delta)$. It is easily verified that the result of forgetting redundant symbols in $\mathcal{T}(\delta)$ is also false.

Definition 10 ensures that $\texttt{p-pre}(\delta)$ is LIA-formula. It is easily verified from Definition 9 that each $\texttt{p-eff}(\delta)(p)$ is an LIA-formula and every $\texttt{p-eff}(\delta)(v)$ is an LIA-term. These, together with Theorem 3, imply that forgetting redundant symbols in the trace axiom of an unconditional pseudo-primitive program is LIA-definable. We concentrate on only the input and output of δ when analyzing the goal-reaching property of δ according to Theorem 1. By Proposition 1, we know that $\mathcal{T}(\delta) \models \mathcal{I}_\delta \rightarrow \mathcal{G}_\delta$ iff $\mathcal{T}'(\delta) \models \mathcal{I}_\delta \rightarrow \mathcal{G}_\delta$ where $\mathcal{T}'(\delta) = \tilde{\exists}\overline{\mathcal{P}}_{\delta\cup\delta'}\tilde{\exists}\overline{\mathcal{V}}_{\delta\cup\delta'}.\mathcal{T}(\delta)$. Hence, checking the goal-reaching property of unconditional pseudo-primitive programs is decidable. Every output symbol is a propositional or numeric symbol. In addition, Propositions 2 and 3, along with the fact that $\mathcal{T}'(\delta)$ is LIA-definable, imply that forgetting all output symbols in $\mathcal{T}'(\delta)$ is also LIA-definable. Therefore, checking the termination and executability properties of unconditional pseudo-primitive programs is also decidable. As a corollary, we obtain:

Corollary 1. *Checking if an unconditional pseudo-primitive program is a solution to a GLINP problem is decidable.*

Example 3. We continue with Example 2 and consider forgetting immediate symbols in $\mathcal{T}(\delta)$. We first present their effects and termination and executability conditions of δ_1, δ_2 and δ_3, respectively.

The body of δ_1 is unconditional action $UnstackY$. The condition of δ_1 is $ny \neq 0$. Clearly, the coefficient of ny is 1 and ny is -1-incremental in $UnstackY$. The number of iterations of δ_1 is ny. At the k-th iteration, ny will be decreased by k. The action $UnstackY$ does not affect the value of both symbols $onxy$ and nx. It is easily verified that the subprogram δ_1 is unconditional pseudo-primitive. Therefore, we get

- $\texttt{p-eff}(\delta_1)(onxy) : onxy$;
- $\texttt{p-eff}(\delta_1)(nx) : nx$;
- $\texttt{p-eff}(\delta_1)(ny) : ny + (-1) \cdot ny = 0$;
- $\texttt{p-pre}(\delta_1) : ny \geq 0 \wedge \forall 0 \leq k < ny.(ny - k \neq 0 \wedge ny - k > 0) \equiv ny \geq 0$.

Similarly, the effect and termination and executability condition of δ_2 are as follows:

- $\texttt{p-eff}(\delta_2)(onxy) : onxy$;
- $\texttt{p-eff}(\delta_2)(nx) : nx + (-1) \cdot nx = 0$;
- $\texttt{p-eff}(\delta_2)(ny) : ny$;
- $\texttt{p-pre}(\delta_2) : nx \geq 0 \wedge \forall 0 \leq k < nx.(nx - k \neq 0 \wedge nx - k > 0) \equiv nx \geq 0$.

As δ_3 is a primitive action, it is easy to obtain its effect and termination and executability condition.

- $\texttt{p-eff}(\delta_3)(onxy) : \top$;
- $\texttt{p-eff}(\delta_3)(nx) : nx$;
- $\texttt{p-eff}(\delta_3)(ny) : ny + 1$;
- $\texttt{p-pre}(\delta_3) : nx = 0 \wedge ny = 0 \wedge \neg onxy$.

By combing the effects and termination and executability conditions of δ_1 and δ_2, we obtain the following:

- $\texttt{p-eff}(\delta_1; \delta_2)(onxy) : onxy$;
- $\texttt{p-eff}(\delta_1; \delta_2)(nx) : 0$;
- $\texttt{p-eff}(\delta_1; \delta_2)(ny) : 0$;
- $\texttt{p-pre}(\delta_1; \delta_2) \equiv nx \geq 0 \wedge ny \geq 0$.

Hence the effect and termination and executability condition of δ.

- $\texttt{p-eff}(\delta)(onxy) : \top$;
- $\texttt{p-eff}(\delta)(nx) : 0$;
- $\texttt{p-eff}(\delta)(ny) : 1$;
- $\texttt{p-pre}(\delta) \equiv nx \geq 0 \wedge ny \geq 0 \wedge \neg onxy$.

The above means that after performing δ, block x is on y and no block is above x. When the numbers of blocks above x and y are non-negative and block x is not on y, the program δ is terminating and executable.

Immediate symbols in the trace axiom $\mathcal{T}(\delta)$ contain $\mathcal{Q}(\mathcal{T}(\delta)) \setminus (\mathcal{P}_\delta \cup \mathcal{P}'_\delta)$ (written $\overline{\mathcal{P}}_{\delta\cup\delta'}$) and $\mathcal{F}(\mathcal{T}(\delta)) \setminus (\mathcal{V}_\delta \cup \mathcal{V}'_\delta)$ (written $\overline{\mathcal{V}}_{\delta\cup\delta'}$). By Theorem 3, we get that $\tilde{\exists}\overline{\mathcal{P}}_{\delta\cup\delta'}\tilde{\exists}\overline{\mathcal{V}}_{\delta\cup\delta'}.\mathcal{T}(\delta) \equiv ny_\delta \geq 0 \wedge nx_\delta \geq 0 \wedge \neg onxy_\delta \wedge onxy'_\delta \wedge nx'_\delta = 0 \wedge ny'_\delta = 1$. The goal formula $\mathcal{G}$ is $onxy$. Clearly, $\tilde{\exists}\overline{\mathcal{P}}_{\delta\cup\delta'}\tilde{\exists}\overline{\mathcal{V}}_{\delta\cup\delta'}.\mathcal{T}(\delta) \models onxy'_\delta$. By Proposition 1 and Theorem 1, the program δ satisfies the goal-reaching property.

In addition, immediate symbols together with output symbols include $\mathcal{Q}(\mathcal{T}(\delta)) \setminus \mathcal{P}_\delta$ (written $\overline{\mathcal{P}}_\delta$) and $\mathcal{F}(\mathcal{T}(\delta)) \setminus \mathcal{V}_\delta$ (written $\overline{\mathcal{V}}_\delta$). It is easily verified that $\tilde{\exists}\overline{\mathcal{P}}_\delta\tilde{\exists}\overline{\mathcal{V}}_\delta.\mathcal{T}(\delta) \equiv ny_\delta \geq 0 \wedge nx_\delta \geq 0 \wedge \neg onxy_\delta$. The initial formula $\mathcal{I}$ is $nx > 0 \wedge ny > 0 \wedge \neg onxy$. Clearly, $\mathcal{I}_\delta \models \tilde{\exists}\overline{\mathcal{P}}_\delta\tilde{\exists}\overline{\mathcal{V}}_\delta.\mathcal{T}(\delta)$. By Theorem 2, the program δ satisfies termination and executability properties. In summary, δ is the solution to the TestOn problem. □

Scala et al. [25] proposed the computation of effects for action repetitions, that is, a sequence with the same action a of a fixed length k. The main idea is to apply the effect of a by k times. Macro actions [23, 26] are an action sequence that occurs frequently in the plan to numeric planning. Clearly, unconditional pseudo-primitive program allows the loop structures and hence is more expressive

than both action repetitions and macro actions. In program verification, unconditional pseudo-primitive programs are a subclass of the triangular program [9] when every action is always executable. However, in AI, it is essential to define the precondition of actions. Hence, unconditional pseudo-primitive programs are orthogonal to triangular programs. In addition, the current theoretical results about verification of triangular programs are limited to termination and do not consider the other two properties: goal-reaching and executability. Our method based on trace axioms is a unified approach to verifying the three properties of programs.

5 EXPERIMENTAL EVALUATION

We have implemented the verification method for planning programs in Python using RegexSkeleton [20], Metric-FF[12] and Z3 [8][3]. RegexSkeleton is used to synthesize a planning program given a set of initial states. Metric-FF planner is used to produce the sequential plan for each initial state. Z3 is used to generate initial states and to decide the entailment problem in the verification phase. The experiment runs on a machine with Intel Core i7-10700 2.90 GHz CPU and 16GB RAM. We test 47 GLINP problems that originate from the work on GP and QNP widely recognized in generalized planning [3, 5, 6, 17, 28, 29][4]. All programs illustrated in [20] are considered in our paper.

We try to synthesize 4 programs for each problem using 2 to 5 initial states by RegexSkeleton. The set S of initial states are generated according to the method proposed in [20]. RegexSkeleton fails to synthesize programs for the two problems MNestVar7 with 4 and 5 states and MNestVar8 with 2 - 5 initial states since the numeric planner Metric-FF[12] cannot generate the sequential plan. Finally we obtain totally 182 candidate programs and all programs are synthesized within 60 seconds.

We summarize the experimental results shown in Table 1, which contains only problems for which at least one synthesized program is unconditional pseudo-primitive. We can make several observations from Table 1. Firstly, 31 out of 47 problems have at least one unconditional pseudo-primitive synthesized programs. The analysis about the other programs are the following. (1) Conditional statements occurs in the generated program in the two problems Delivery and Gripper. (2) The generated program contains nested loops in the problems MNestVar with 2 - 7 variables, Miconic, Rewards and Visitall. (3) The problems Corner-R, D-Return-R, Hall-R, and Visitall-R contain conditional effects. (4) There are more than one numeric symbols w s.t. w is d_w-incremental in the loop body and $c_w, d_w \in \{-1, 1\}$ in the generated program of D-Return when using 2 and 3 initial states. Secondly, Our approach is able to check against the correctness of all unconditional pseudo-primitive programs within 1 second. In particular, the generated problem of the problem NestVar8 has length 541. Hence, our approach is an effective one to correctness verification of diverse and complex planning programs. In addition, except for D-Return, in 30 problems shown in Table 1, RegexSkeleton is able to synthesize unconditional pseudo-primitive programs given 2 - 5 initial states and to obtain correct programs using only 5 initial states.

[3]Source codes are provided in https://github.com/oldg00se/GLINP.
[4]The files of test cases can be downloaded from https://github.com/oldg00se/Domain-for-GLINP.

Problems	Number of Initial States			
	2	3	4	5
Arith	✓	✓	✓	✓
Baking	✓	✓	✓	✓
Barman2-8	✓	✓	✓	✓
Childsnack	G	G	G	G
Chop	✓	✓	✓	✓
ClearBlock	✓	✓	✓	✓
Corner-A	✓	✓	✓	✓
Corridor	✓	✓	✓	✓
D-Return	-	-	×	×
Grid	G	✓	✓	✓
Hall-A	TE	TE	TE	✓
Hiking	✓	✓	✓	✓
Intrusion	✓	✓	✓	✓
Lock	✓	✓	✓	✓
NestVar2-8	✓	✓	✓	✓
PlaceBlock	✓	✓	✓	✓
Snow	✓	✓	✓	✓
Spanner	✓	✓	✓	✓
TestOn	✓	✓	✓	✓

Table 1: Experimental results. "✓" denotes that the synthesized program satisfies goal-reaching, terminating and executability properties; "G" means that the program satisfies goal-reaching property but not terminating and executability properties; "TE" denotes that the program satisfies both of terminating and executability properties but not goal-reaching property; "×" indicates that the program satisfies none of the three properties; "-" denotes that the program is not unconditional pseudo-primitive.

6 CONCLUSIONS

In this paper, we have designed a theoretical and practical method for automatic correctness verification of linear integer planning programs. The fundamental idea is to translate a planning program into a finite set of LIAUPF-formulas, namely trace axioms. Then, we reduce the correctness verification of planning programs to the entailment problem of the result of forgetting redundant symbols in the trace axiom. Our approach is a symbolic way without the aid of loop invariants and ranking functions. In general, checking the correctness of programs is undecidable. To obtain the decidability result, we identify a class of planning programs, namely unconditional pseudo-primitive programs, such that the result of forgetting redundant symbols in the transformed formula is LIA-definable. Hence, the correctness verification of unconditional pseudo-primitive programs becomes decidable.

ACKNOWLEDGMENTS

We are grateful to Chenglin Wang for his insightful comments on the paper. We also appreciate the contributions of Zexin Cai and Yichao Hong to the implementation of our approach. This paper was supported by National Key R&D Program of China (2022YFE0116800), National Natural Science Foundation of China (62276114), Guangdong Basic and Applied Basic Research Foundation (2023B1515120064 and 2024A1515011762), Science and Technology Planning Project of Guangzhou (202206030007, 2025A03J3565 and Nansha District: 2023ZD001), the Fundamental Research Funds for the Central Universities, JNU (21623202).

REFERENCES

[1] Corin R. Anderson, David E. Smith, and Daniel S. Weld. 1998. Conditional Effects in Graphplan. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-1998)*. 44–53.

[2] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (Eds.). 2003. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

[3] Blai Bonet, Guillem Francès, and Hector Geffner. 2019. Learning Features and Abstract Actions for Computing Generalized Plans. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-2019)*. 2703–2710.

[4] Blai Bonet and Hector Geffner. 2018. Features, Projections, and Representation Change for Generalized Planning. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-2018)*. 4667–4673.

[5] Blai Bonet and Hector Geffner. 2020. Qualitative Numerical Planning: Reductions and Complexity. *Journal of Artificial Intelligence Research* (2020), 923–961.

[6] Blai Bonet, Héctor Palacios, and Hector Geffner. 2009. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-2009)*. 34–41.

[7] George Boole. 1854. *An Investigation of the Laws of Thought*. Walton & Maberly.

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-2008)*. 337–340.

[9] Florian Frohn and Jürgen Giesl. 2019. Termination of Triangular Integer Loops is Decidable. In *Proceedings of the Thirty-First International Conference on Computer Aided Verification (CAV-2009)*. Lecture Notes in Computer Science, Vol. 11562. Springer, 426–444.

[10] B. Cenk Gazen and Craig A. Knoblock. 1997. Combining the Expressivity of UCPOP with the Efficiency of Graphplan. In *Proceedings of the Fourth European Conference on Planning (ECP-1997)*. Lecture Notes in Computer Science, Vol. 1348. Springer, 221–233.

[11] Kurt Gödel. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik* (1931), 173–198.

[12] Jörg Hoffmann. 2003. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research* 20 (2003), 291–341.

[13] Yuxiao Hu and Giuseppe De Giacomo. 2011. Generalized Planning: Synthesizing Plans that Work for Multiple Environments. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI-2011)*. 918–923.

[14] Yuxiao Hu and Hector J. Levesque. 2010. A Correctness Result for Reasoning about One-Dimensional Planning Problems. In *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR-2010)*. 362–371.

[15] León Illanes and Sheila A. McIlraith. 2019. Generalized Planning via Abstraction: Arbitrary Numbers of Objects. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-2019)*. 7610–7618.

[16] Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. 1997. Extending Planning Graphs to an ADL Subset. In *Proceedings of the Fourth European Conference on Planning (ECP-1997) (Lecture Notes in Computer Science, Vol. 1348)*. Springer, 273–285.

[17] Hector J. Levesque. 2005. Planning with Loops. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-2005)*. 509–515.

[18] Fangzhen Lin. 2016. A formalization of programs in first-order logic with a discrete linear order. *Artificial Intelligence* 235 (2016), 1–25.

[19] Fangzhen Lin and Raymond Reiter. 1994. Forget It!. In *Proceedings of AAAI Fall Symposium on Relevance*. 154–159.

[20] Xiaoyou Lin, Qingliang Chen, Liangda Fang, Quanlong Guan, Weiqi Luo, and Kaile Su. 2022. Generalized Linear Integer Numeric Planning. In *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS-2022)*. 241–251.

[21] Bernhard Nebel. 2000. On the Compilability and Expressive Power of Propositional Planning Formalisms. *Journal of Artificial Intelligence Research* 12 (2000), 271–315.

[22] Edwin P. D. Pednault. 1989. ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR-1989)*. 324–332.

[23] Enrico Scala. 2014. Plan Repair for Resource Constrained Tasks via Numeric Macro Actions. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS-2014)*. 280–288.

[24] Enrico Scala, Patrik Haslum, and Sylvie Thiébaux. 2016. Heuristics for Numeric Planning via Subgoaling. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-2016)*. 3228–3234.

[25] Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramírez. 2020. Subgoaling Techniques for Satisficing and Optimal Numeric Planning. *Journal of Artificial Intelligence Research* 68 (2020), 691–752.

[26] Enrico Scala and Pietro Torasso. 2015. Deordering and Numeric Macro Actions for Plan Repair. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-2015)*. 1673–1681.

[27] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. 2018. Computing Hierarchical Finite State Controllers with Classical Planner. *Journal of Artificial Intelligence Research* 62 (2018), 755–797.

[28] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. 2011. A new representation and associated algorithms for generalized planning. *Artificial Intelligence* 175, 2 (2011), 615–647.

[29] Siddharth Srivastava, Shlomo Zilberstein, Neil Immerman, and Hector Geffner. 2011. Qualitative Numeric Planning. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI-2011)*. 1010–1016.

[30] Elly Winner and Manuela Veloso. 2003. DISTILL: Learning Domain-Specific Planners by Example. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*. 800–807.