Applicability Conditions for Plans with Loops: Computability Results and Algorithms

Siddharth Srivastava, Neil Immerman, Shlomo Zilberstein

Department of Computer Science
University of Massachusetts,
Amherst, MA 01003

Abstract

The utility of including loops in plans has been long recognized by the planning community. Loops in a plan help increase both its applicability and the compactness of representation. However, progress in finding such plans has been limited largely due to lack of methods for reasoning about the correctness and safety properties of loops of actions. We present novel algorithms for determining the applicability and progress made by a general class of loops of actions. These methods can be used for directing the search for plans with loops towards greater applicability while guaranteeing termination, as well as in post-processing of computed plans to precisely characterize their applicability. Experimental results demonstrate the efficiency of these algorithms. We also discuss the factors which can make the problem of determining applicability conditions for plans with loops incomputable.

Keywords: Automated planning, plans with loops, plan verification, reachability in abacus programs, generalized planning

1. Introduction

The problem of planning in AI is to compute a plan, or a procedure which can be executed by an agent to achieve a certain goal. This paper presents methods which can be used for the computation of generalized, algorithmic procedures for solving this problem.

In the classical formulation of AI planning, the agent's state is assumed to be completely observable, and effects of actions are assumed to be determined entirely by this state. Classical plans consist of linear sequences of actions which lead to a goal state from a particular initial state. Even in this restricted formulation, the planning problem is PSPACE-complete (Bylander, 1994). More general formulations which allow the agent to possess only partial information

Email addresses: `siddharth@cs.umass.edu` (Siddharth Srivastava), `immerman@cs.umass.edu` (Neil Immerman), `shlomo@cs.umass.edu` (Shlomo Zilberstein)

about its current state, and its actions to be non-deterministic make the problem significantly harder (Rintanen, 2004). Consequently, numerous approaches have been proposed for reusing sequences of actions computed for related problems (Fikes et al., 1972; Hammond, 1986) and for computing generalized plans which can be used to solve large classes of planning problems (Shavlik, 1990; Levesque, 2005; Winner and Veloso, 2007; Srivastava et al., 2010c).

Approaches for the latter formulation of this problem build extensively upon the power of including loops of actions for representing cyclic flows of control in plans. Not only are such constructs necessary when the input problem instances can be unbounded in size, but they also allow significant reductions in plan sizes for larger problems—particularly when contingent solutions are required in order to deal with partial observability (Srivastava et al., 2010b). Plans with loops therefore present two very appealing advantages: they can be more compact, and thus easier to synthesize, and they often solve many problem instances, offering greater generality.

Loops in plans, however, are inherently unsafe structures because it is hard (and even impossible, in general) to determine the general conditions under which they will terminate and achieve their intended goals. It is therefore crucial to determine when a plan with loops can be safely applied to a problem instance. Unfortunately, there is currently very little understanding of when the applicability conditions of plans with loops can even be found, and if so, whether this can be done efficiently. This limitation significantly impacts the development and usability of approaches for finding generalized plans.

In this paper, we present methods for efficiently determining the conditions under which plans with some classes of simple and nested loops can solve a problem instance. We initially assume that planning actions come from a simple, but powerful class of action operators, which can only increment or decrement a finite set of registers by unit amounts. Although this class of actions appears to be too primitive for planning domains, in Section 7 we show that plans with loops for many interesting planning problems can be automatically translated into plans with such actions.

The class of actions considered in this work is captured by *abacus programs*—an abstract computational model as powerful as Turing machines. The halting problem for abacus programs is thus undecidable. In other words, finding closed-form applicability conditions, or preconditions for plans with loops of just the primitive abacus actions is undecidable. Despite this negative result, we show that closed-form preconditions *can* be found very efficiently for structurally restricted classes of abacus programs, and demonstrate that such structures are sufficient to solve interesting planning problems. In addition to determining the scope of applicability of existing plans, these methods can also be used during plan computation for identifying useful loops of actions.

We start with a formal definition of abacus programs in the next section. This is followed by a study of the problem of finding preconditions of abacus programs with simple loops (Section 3) and a class of nested loops (Section 4). In Section 5.1 we study the problem of computing preconditions of abacus programs in the presence of non-deterministic actions similar to those used in planning.

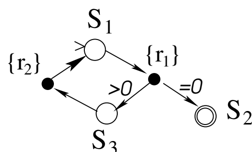


Figure 1: A simple abacus machine for the program: `while` $(r_1 > 0)$ { $r_1--;r_2++$}

In this paper, we restrict our attention to abacus programs whose nested loops satisfy a *monotonicity* condition; in Section 6, we show that the problem of determining reachability without this condition becomes undecidable. Finally, we illustrate how plans with loops can be translated into abacus programs in Section 7, and conclude with a demonstration of the scope and efficiency of the presented methods.

2. Abacus Programs

We now introduce the formal framework of abacus programs (Lambek, 1961). Abacus programs are finite automata whose states are labeled with actions that increment or decrement a fixed set of registers. Formally,

Definition 1. (Abacus Programs) *An abacus program* $\langle \mathcal{R}, \mathcal{S}, s_0, s_h, \ell \rangle$ *consists of a finite set of registers* $\mathcal{R}$*, a finite set of states* $\mathcal{S}$ *with special initial and halting states* $s_0, s_h \in \mathcal{S}$ *and a labeling function* $\ell : \mathcal{S} \setminus \{s_h\} \mapsto \text{Act}$*. The set of actions,* Act*, consists of actions of the form:*

- *Inc*(r, s)*: increment* $r \in \mathcal{R}$*; goto* $s \in \mathcal{S}$*, and*
- *Dec*(r, s_1, s_2)*: if* $r = 0$ *goto* $s_1 \in \mathcal{S}$ *else decrement* r *and goto* $s_2 \in \mathcal{S}$

We represent abacus programs as bipartite graphs with edges from nodes representing states to nodes representing actions and vice-versa. State-nodes have at most one outgoing edge and action-nodes have at most two outgoing edges; the two edges out of a decrement action are labeled $= 0$ and > 0 respectively (see Fig. 1). A more succinct representation that does not use state-nodes is also possible, but we use state-nodes to improve clarity and maintain a correspondence with planning scenarios (see Section 7).

Given an initial valuation of its registers, the execution of an abacus program starts at s_0. At every step, an action is executed, the corresponding register is updated, and a new node is reached. An abacus program *terminates* iff its execution reaches the halt node. The set of final register values in this case is called the *output* of the abacus program.

Abacus programs are equivalent to Minsky Machines (Minsky, 1967), which are as powerful as Turing machines and thus have an undecidable halting problem:

Fact 1. *The problem of determining the set of initial register values for which an abacus program will reach the halt node is undecidable.*

Nevertheless, we identify in this paper a general class of loops for which the halting problem *is* decidable.

As discussed in the introduction, our approach for determining the utility and applicability conditions of loops of planning actions is to view them as abacus programs with the same loop structure. Maintaining the loop structure while doing this translation is important as it will allow us to selectively construct plans with the structures that we know can be handled. However, the abacus program framework is restrictive from this point of view: it does not include non-deterministic actions. In planning on the other hand, non-deterministic sensing actions are common and we need a way to translate them into the abacus framework effectively, without changing the loop structure. For this purpose, we extend the abacus program framework with the following non-deterministic action, in the representation of Def. 1:

Definition 2. (Non-deterministic Abacus Programs) *Non-deterministic abacus programs are abacus programs whose set of actions,* Act *includes, in addition to the* Inc *and* Dec *actions, non-deterministic actions of the form:*

- $\mathrm{NSet}(r, s_1, s_2)$*: set* r *to* 0 *and goto* $s_1 \in \mathcal{S}$ ***or*** *set* r *to* 1 *and goto* $s_2 \in \mathcal{S}$*.*

where $\mathcal{S}$ *is the set of states of the abacus program and* r *is a register that is not used by deterministic actions.*

A non-deterministic action thus has two outgoing edges in the graph representation, corresponding to the two possible values it can assign to a register value. Either of these branches may be taken during execution. Although the original formulation of abacus programs is sufficient to capture any computation, these actions will allow us to conveniently treat a powerful class of nested loops (encountered in partially observable planning) as a set of independent simple loops.

3. Applicability Conditions for Deterministic Simple-Loop Abacus Programs

We now show that for any simple-loop abacus program, we can efficiently characterize the exact set of register values that lead not just to termination, but to any desired "goal" node defined by a given set of register values (Theorem 1). We only consider deterministic actions in this section; the case for simple loops with non-deterministic actions is analogous and can also be handled as a special case of the methods presented in Section 5.1 for a more general class of loops.

We define simple-loop abacus programs as follows:

Definition 3. (Simple-Loop Abacus Programs) *A simple loop in a graph is a strongly connected component consisting of exactly one cycle. A* simple-loop

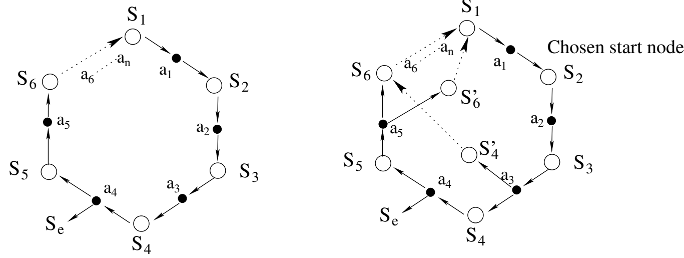


Figure 2: A simple loop with (right) and without (left) shortcuts

abacus program is one all of whose non-trivial strongly connected components are simple loops.

Let $S_1, a_1, \ldots, S_n, a_n, S_1$ be a simple loop (see Fig. 2). We denote register values at nodes using vectors. For example, $\bar{R}^0 = \langle R_1^0, R_2^0, \ldots, R_m^0 \rangle$ denotes the initial values of registers $R_1, \ldots, R_m$ at node S_1. Let $a(i)$ denote the index of the register changed by action a_i. Since these are abacus actions, if there is a branch at a_i, it will be determined by whether or not the value of $R_{a(i)}$ is greater than or equal to 0 at the *previous* node.

We use subscripts on vectors to project the corresponding registers, so that the initial count of action a_i's register can be represented as $\bar{R}^0_{a(i)}$. Let Δ^i denote the vector of changes in register values $R_1, \ldots, R_m$ for action a_i corresponding to its branch along the loop. Let $\Delta^{1..i} = \Delta^1 + \Delta^2 + \cdots + \Delta^i$ denote the register-change vector due to a sequence of abacus actions $a_1, \ldots, a_i$. Given a linear segment of an abacus program, we can easily compute the preconditions for reaching a particular register value and node combination:

Proposition 1. *Suppose $S_1 \xrightarrow{a_1} S_2 \xrightarrow{a_2} \cdots S_n$ is a linear segment of an abacus program where S_i are nodes, a_i are actions and $\bar{F}$ is a vector of register values. A set of necessary and sufficient linear constraints on the initial register values $\bar{R}^0$ at S_1 can be computed under which S_n will be reached with register values $\bar{F}$.*

Proof. We know $\bar{F} = \bar{R}^0 + \Delta^{1..n}$. We only need to collect the conditions necessary to take all the correct action branches, keeping us on this path. Since the sequence of actions is known, register values at each node S_i can be represented in terms of $\bar{R}^0$. These expressions can be used to state the inequality that must hold for following the desired branch of the next action. □

Proposition 2. *Suppose we are given a simple loop, $S_1, a_1, \ldots, S_n, a_n, S_1$, of an abacus program. Then in $O(n)$ time we can compute a set of linear constraints, $C(\bar{R}^0, \bar{F}, \ell)$, that are satisfied by initial and final register tuples, $\bar{R}^0, \bar{F}$, and natural number, ℓ, iff starting an execution at S_1 with register values $\bar{R}^0$ will result in ℓ iterations of the loop, after which we will be in S_1 with register values $\bar{F}$.*

Proof. Consider the action a_4 in the left loop in Fig. 2. Suppose that the condition that causes us to stay in the loop after action a_4 is that $R_{a(4)} > 0$. Then the loop branch is taken during the first iteration starting with fluent-vector $\bar{R}^0$ if $(\bar{R}^0 + \Delta^{1..3})_{a(4)} > 0$. This branch will be taken in ℓ subsequent loop iterations iff $(\bar{R}^0 + k \cdot \Delta^{1..n} + \Delta^{1..3})_{a(4)} > 0$, and similar inequalities hold for every branching action, for *all* $k \in \{0, \ldots, \ell - 1\}$. More precisely, for one full execution of the loop starting with $\bar{R}^0$ we require, for all $i \in \{1, \ldots, n\}$:

$$(\bar{R}^0 + \Delta^{1..i-1})_{a(i)} \circ 0$$

where $\circ$ is one of $\{>, =\}$ depending on the branch that lies in the loop; (this set of inequalities can be simplified by removing constraints that are subsumed by others). Since the only variable term in this set of inequalities is $\bar{R}^0$, we represent them as $\mathsf{LoopIneq}(\bar{R}^0)$. Let $\bar{R}^\ell = \bar{R}^0 + \ell \times \Delta^{1..n}$, the register vector after ℓ complete iterations. Thus, for executing the loop completely ℓ times, the required conditions are $\mathsf{LoopIneq}(\bar{R}^0) \wedge \mathsf{LoopIneq}(\bar{R}^{\ell-1})$. These two sets of conditions ensure that the conditions for execution of intermediate loop iterations hold, because the changes in register values due to actions are constant, and the expression for $\bar{R}^{\ell-1}$ is linear in them. Note that these conditions are necessary and sufficient since there is no other way of executing a complete iteration of the loop except by undergoing all the register changes and satisfying all the branch conditions.

Hence, the necessary and sufficient conditions for achieving the given register-value after ℓ complete iterations are:

$$C(\bar{R}^0, \bar{F}, \ell) \equiv \mathsf{LoopIneq}(\bar{R}^0) \wedge \mathsf{LoopIneq}(\bar{R}^{\ell-1}) \wedge (\bar{F} = \bar{R}^\ell).$$

Each loop inequality is constant size because it concerns a single register. The total length of all the inequalities is $O(n)$ and as described above they can be computed in a total of $O(n)$ time. □

Note that an exit during the first iteration amounts to a linear segment of actions and is handled by Prop. 1. Further, the vector $\bar{F}$ can include symbolic expressions. Initial values R^0 can be computed using $R^\ell = F$; these expressions for R^0 can be used as target values for subsequent applications of Prop. 2. Therefore, when used in combination with Prop. 1, the method described above produces the necessary and sufficient conditions for reaching any node and register value in an abacus program:

Theorem 1. *Let Π_A be a simple-loop abacus program. Let S be any node in the program, and $\bar{F}$ a vector of register values. We can then compute a disjunction of linear constraints on the initial register values that is a necessary and sufficient condition for reaching S with the register values $\bar{F}$.*

Proof. Since Π_A is acyclic except for simple loops, it can be decomposed into a set of segments starting at the common start-node, but consisting only of linear paths and simple loops (This may require duplication of nodes following a node where different branches of the plan merge. Thus, in the worst case, the

total size of the disjunction could be exponential.). By Prop. 1 and 2, necessary and sufficient conditions for each of these segments can be computed. The disjunctive union of these conditions gives the overall necessary and sufficient condition. □

4. Nested Loops Due to Shortcuts

Due to the undecidability of the halting problem for abacus programs, it is impossible to find preconditions of abacus programs with arbitrarily nested loops. The previous section demonstrates, however, that structurally restricted classes of abacus programs admit efficient applicability tests.

In this section, we show that methods developed in the previous section *can* be extended to a class of graphs representing nested loops obtained by adding unidirectional paths, or shortcuts to a simple loop. We first define the general class of non-simple loops as follows:

Definition 4. (Complex Loops) *A* complex *loop in a graph is a non-trivial strongly connected component that is not a simple loop.*

In particular, we will be interested in a special class of complex loops, i.e., those obtained by adding "shortcuts" in a simple loop:

Definition 5. (Simple loop with shortcuts) *A simple loop with shortcuts is a strongly connected component $\mathcal{C}$ which includes a node S_0, designated the* start node*, such that removing S_0 makes $\mathcal{C}$ acyclic.*

Intuitively, such a simple loop with shortcuts consists of a simple loop with all elements, starting at the start node, in increasing linear order. For any pair of nodes along the loop, a preceeding b, a shortcut from a to b may be added; different shortcuts may overlap as long as this does not create cycles. (e.g., node S_2 can be designated the start node in Fig. 2).

Simple loops with shortcuts form a very general class of complex loops: graph theoretically, this is exactly the class of strongly connected components with *cycle rank* (Eggan, 1963) 1. Many control flows that are typically understood as "nested" loops in programming can be represented as simple loops with shortcuts. In the case of abacus programs, we show in Section 6 that this class of graphs is powerful enough to express any computation.

The advantage of this class of loops is that we can decompose them into simple loops; in the definition below, a cycle has no repeated nodes other than the start and end nodes.

Definition 6. *(Loop Decomposition) Let $\mathcal{K}$ be a graph in the form of a simple loop with shortcuts with start node S_0. The loop decomposition of $\mathcal{K}$ is defined as the set of all cycles of $\mathcal{K}$ beginning with S_0.*

In the worst case, the size of this decomposition can be exponential in the number of shortcuts. This construction proves useful because in a simple loop with shortcuts, every cycle must contain the start node (this is immediate from

Def. 5). Thus, the execution of a simple loop with shortcuts can be viewed as a sequence of complete executions of the simple loops in its decomposition. For instance, we can view the loop with shortcuts in Fig. 2 as consisting of 3 different simple loops. The order of execution of these loops, and whether a given loop will be executed at all, will depend on the results of actions a_3 and a_5.

We now define a special class of simple loops with shortcuts for abacus programs. In the next section we present efficient methods for finding preconditions of such programs.

Definition 7. (Monotone simple loops with shortcuts) *A simple loop with shortcuts in an abacus program is monotone iff the sign (positive or negative) of the net change, if any, in a register's value is the same for every simple loop in its decomposition.*

In Section 6 we show that removing this restriction can signficiantly increase the power of abacus programs: any abacus program can be represented as a program consisting of a simple loop with possibly nonmonotone shortcuts.

5. Applicability Conditions for Monotone Simple Loops with Shortcuts

We now consider the problem of computing applicability conditions for monotone simple loops with shortcuts. We first present the more general case of programs which may include non-deterministic actions. We also categorize the conditions under which these methods provide accurate results. Although the methods for non-deterministic programs can also be applied to deterministic programs, we present more accurate methods for deterministic programs in Section 5.2

5.1. Non-deterministic Monotone Shortcuts

We first consider the the problem of computing applicability conditions for abacus programs whose simple loops have monotone shortcuts and include non-deterministic actions. We will find that the accuracy of the resulting conditions is determined by *order independence* (Def. 8), or the extent to which the execution of different loops in a decomposition can be rearranged without significantly affecting the overall outcomes. The situation where we have only deterministic actions can be seen as one of the extreme cases of order *dependence*. While the quality of methods presented in this section will suffer in such cases, we present specialised methods for handling them in the following section (5.2).

Suppose an abacus program Π is a simple loop with shortcuts which can be decomposed into m simple loops with the start node S_{start}. We consider the case of l complete iterations of Π counted at its start node, with $k_1, \ldots, k_m$ representing the number of times loops $1, \ldots, m$ are executed, respectively. The final, partial iteration and the loop exit can be along any of the simple loops and can be handled as a linear program segment. Then,

$$k_1 + \ldots k_m = l. \tag{1}$$

Determining Final Register Values. We denote the loop created by taking the ith shortcut as $loop_i$. The final register values after the $l = \sum_{i=1}^{m} k_i$ complete iterations can be obtained by adding the changes due to each simple loop, with Δ^{loop_i} denoting the change vector due to $loop_i$:

$$\bar{F} = \bar{R}^0 + \sum_{i=1}^{m} k_i \Delta^{loop_i} \tag{2}$$

Cumulative Branch Conditions. For computing sufficient conditions on the achievable register values after $k_1, \ldots, k_m$ complete iterations of the given loops, our approach is to treat each loop as a simple loop and determine its preconditions. Note that every required condition for a loop's complete iteration stems from a comparison of a register's value with zero. We therefore want to determine the lowest possible value of each register during the $k_1, \ldots k_m$ iterations of loops $1, \ldots, m$, and constrain that value to be greater than zero.

Let $\mathcal{R}^+, \mathcal{R}^-$ be the sets of registers undergoing *net* non-negative and negative changes respectively, by any loop. Consider an $R_j \in R^-$. Loops with net zero or negative effects on R_j may include both incrementing and decrementing actions. Let δ_j^i be the greatest partial negative change caused on R_j by $loop_i$. Let $min(j) = argmin_x\{\delta_j^x : x \in \{1, \ldots, m\}\}$.

For $R_j \in R^+$, the lowest possible value is $R_j^0 + \delta_j^{min(j)}$, since the value of R_j can only increase after the first iteration. The required constraint on $R_j \in \mathcal{R}^+$ therefore is $R_j^0 + \delta_j^{min(j)} \geq 0$ ("$\geq$" because "$>$" must hold *before* the decrement).

The following lemma shows that in order to compute the least value of a register in $\mathcal{R}^-$ over all possible executions with $k_1, \ldots k_m$ iterations of loops $1, \ldots, m$ respectively, we only need to consider which loop is executed last:

Lemma 1. *Suppose* $\mathrm{loop}_1, \ldots, \mathrm{loop}_m$ *form the decomposition of a monotone simple loop with shortcuts. Then there exists an execution ordering such that the lowest value (over all possible orderings with k_i iterations of* loop_i *for all i) of $R_j \in \mathcal{R}^-$ is achieved in the last loop to be executed.*

Proof. Suppose this is not the case and the lowest possible value of R_j is achieved in the "middle" of an execution ordering, i.e., during the execution of $loop_x$, following which other loop iterations will be executed.

We can change the execution ordering so that this iteration of $loop_x$ occurs at the end. In this new ordering, because of monotonicity, we either get a lower value of R_j during the last iteration of $loop_x$ or the same one. □

We now compute the lowest value of R_j achieved during the last iteration, after $k_1, \ldots, k_m$ iterations of loops $1, \ldots, m$, with loop x being executed the last. If $\delta_j^x \neq \Delta_j^x$, then this may occur after a partial execution of the last iteration of $loop_x$. We obtain this value by first computing the value of R_j after execution of all iterations of all the required loops, and then subtracting from it the effect of one complete iteration of $loop_x$, and adding δ_j^x, the greatest partial negative change of $loop_x$:

$$R_j^0 + \sum_{i=1}^{m} k_i \Delta_j^i - \Delta_j^{loop_x} + \delta_j^x$$

To obtain the lowest value of this expression over all possible choices for the last loop, we need to minimize this expression w.r.t x. In most cases encountered in planning, this can be done effectively by choosing the loop which minimizes δ_j^x and using that loop for x (this method was used by Srivastava et al. (2010a)). In this paper, we use the more general approach by selecting the last loop, $\widehat{\jmath}$, as follows:

$$\widehat{\jmath} = argmin_x\{\delta_j^x - \Delta_j^{loop_x} : x \in \{1, \ldots, m\}\}$$

This minimization $\widehat{\jmath}$ requires the same number of comparison operations the minimization over δ_j^x alone. Now that we can compute the minimum possible values of all registers, we can state the required constraints as:

$$\forall R_j \in R^- \big\{R_j^0 + \sum_{i=0}^{m} k_i \Delta_j^{loop_i} + \delta_j^{\widehat{\jmath}} - \Delta_j^{loop_{\widehat{\jmath}}} \geq 0\big\} \tag{3*}$$

$$\forall R_j \in R^+ \big\{R_j^0 + \delta_{min(j)} \geq 0\big\} \tag{4*}$$

Together with Eqs. (1-2), these inequalities provide sufficient conditions binding reachable register values with the number of loop iterations and the initial register values. However, the process for deriving them assumed that for every j, $loop_{\widehat{\jmath}}$ and $loop_{min(j)}$ will be executed at least once. We can make these constraints more accurate by using a disjunctive formulation for selecting the loop causing the greatest negative change among those that are executed at least once. For register R_j, let $0\widehat{\jmath}, \ldots m\widehat{\jmath}$ be the ordering of loops in decreasing order of the values $\delta_j^x - \Delta_j^x$. We will use this ordering for writing the constraints for registers in $\mathcal{R}^-$. Similarly, let $0j, \ldots, mj$ be the ordering of loops in decreasing values of δ_j^x, with the intended purpose of writing constraints for registers in $\mathcal{R}^+$. In each of the following constraints, we will use $k_{i<x} = 0$ to denote the constraints $\{k_i = 0 : i < x\}$, where the ordering is the one being used in that constraint. We can now write disjunctions of constraints corresponding to the first loop in these orderings that is executed at least once, as follows:

$$\begin{aligned} \forall R_j \in R^- \bigvee_{x=0\widehat{\jmath},\ldots,m\widehat{\jmath}} \big\{&k_{i<x} = 0; k_x \neq 0; \\ &R_j^0 + \sum_{x \leq i \leq m\widehat{\jmath}} k_i \Delta_j^{loop_i} + \delta_j^x - \Delta_j^{loop_x} \geq 0\big\} \end{aligned} \tag{3}$$

$$\forall R_j \in R^+ \bigvee_{x=0_j,\ldots,m_j} \big\{k_{i<x} = 0; k_x \neq 0; R_j^0 + \delta_j^x \geq 0\big\} \tag{4}$$

Constraints 3 & 4 are derived from 3* and 4* by replacing the argmins $\widehat{\jmath}$ and *min(j)* by the variable x, which iterates over loops in the order $0\widehat{\jmath}, \ldots, m\widehat{\jmath}$ for registers in $\mathcal{R}^-$ and in the order $0j, \ldots, mj$ for registers in $\mathcal{R}^+$.

Constraint 3 is tighter than 3* only when changing the loop that executes last will have an impact on the lowest value of at least one register. Otherwise, $\delta_x - \Delta_j^{loop_x}$ will be the same for every loop for each register R_j, representing the situation where the lowest achievable value of register R_j is independent of which loop's execution occurs last.

Accuracy of the Computed Conditions. Note that these conditions do not deal with equality conditions that may have to be satisfied for staying in a loop. Equality conditions are very constraining, and may constrain the execution of a loop corresponding to a shortcut to occur exactly once, when the equality condition holds. However, conditions (1-4) can be extended to include equality conditions for the first and last iteration of each loop. This will make (1-4) sufficient conditions for situations where equality branches are required to stay in the loop (in our experience this is rare in planning domains). However, adding these constraints may also make (1-4) unsatisfiable if the same register is used in two different equality constraints corresponding to two different loops caused by shortcuts.

In order to discuss when conditions (1-4) are accurate we first define order independence:

Definition 8. (Order Independence) *A simple loop with shortcuts is order independent if for every initial valuation of the registers at S_{start}, the set of register-values possible at S_{start} after any number of iterations does not depend on the order in which those iterations are taken.*

An equality constraint in a loop is considered *spurious*, if no loop created by the shortcuts changes the register on which equality is required. During the execution of the loop, the truth of such conditions will not change. Consequently, such equality conditions do not introduce order dependence. In practice, these conditions can be translated into conditions on register values just prior to entering the loop.

A simple loop with shortcuts will have to be order dependent if one of the following holds: (1) the lowest value achievable by a register during its execution depends on the order in which shortcuts are taken. In this case, possible lowest values will impose different constraints for each ordering; or, (2) a *non-spurious* equality condition has to be satisfied to stay in a loop. In the latter case, the non-deterministic branch leading to the shortcut that has the equality condition will have to be taken at the precise iteration when equality is satisfied. In fact, the disjunction of these two conditions is necessary and sufficient for a loop to be order dependent.

Proposition 3. *A simple loop with shortcuts is order-dependent iff either (1) the lowest value achievable by a register during its execution depends on the order in which shortcuts are taken or (2) a non-spurious equality condition has to be satisfied to continue a loop iteration.*

Proof. Sufficiency of the condition was discussed above. If the loop is order dependent, then there is a register value that is reachable only via a "good"

subset of the possible orderings of shortcuts. Consider an ordering with the same number of iterations of these shortcuts, not belonging to this subset. During the execution of this sequence, there must be a first step after which a loop iteration that could be completed in the good subset, cannot be completed in the chosen ordering. This has to be either because an inequality > 0 is not satisfied before a decrement, which implies (1) holds, or because $R_j = 0$ is required to continue the iteration; this must have been possible in the good loop orderings, but $R_j > 0$ must hold here, which implies case (2) holds. □

A naive approach of even expressing the necessary conditions for an order dependent loop can be exponential in the number of shortcuts, even while considering just a single iteration of each loop. Deriving better representations for such conditions is an important direction for future work. However, we can now see the computation of $\widehat{\jmath}$ as handling a very specific kind of order dependence, when the lowest value of a register only depends on the last iteration to be executed.

Example 1. *Consider loops l_1, l_2 created by shortcuts in a larger loop. l_1 increases R_1 by 5 and R_2 by 1. l_2 first decreases R_1 by 4 and then increases it by 5. l_1, l_2 are monotone shortcuts but their combination is order dependent: at S_{start} with $R_1 = 1$, l_2 cannot be executed completely before executing l_1. Expressing precise preconditions for reachable register values thus requires a specification of the order in which the shortcuts have to be taken.*

We can now present two results capturing the accuracy of the conditions (1-4).

Proposition 4. *If Π is an order independent simple loop with monotone shortcuts, then Eqs. (1-4) provide necessary and sufficient conditions on the initial and achievable register values.*

Proof. By construction, the inequalities ensure that none of the register values drops to zero, so that if a register value satisfies the inequalities, then it will be reachable. This proves that the conditions are sufficient. Suppose that a register value $\bar{F}$ is reachable from $\bar{R}^0$, after $k_0, \ldots k_m$ iterations of $loop_0, \ldots, loop_m$ respectively. Eq. (2) cannot be violated, because the changes caused due to the loops are fixed; Eq. (1) will be satisfied trivially. If $\bar{R}^0, k_0, \ldots, k_m$ don't satisfy Eqs. (3-4), the lowest value achieved during the loop iterations will fall below zero because the loop is order independent. Therefore, (1-4) must be satisfied. □

Proposition 5. *If Π is a simple loop with monotone shortcuts, then Eqs. (1-4), together with constraints required for equality branches during the first and last iterations of the shortcuts containing them give sufficient conditions on the possible final register values in terms of their initial values.*

Proof. By construction, conditions (1-4) and the equality constraints ensure that every branch required to complete k_i iterations of loop i will be satisfied. □

In other words, if we don't have order independence, the conditions (1-4) are sufficient, but not necessary. In adversarial formulations however, if the next simple loop to be executed depends on non-deterministic actions, then we require exactly the conditions (1-4) which ensure that all the stipulated iterations of all the loops will be executed. In Section 8 we present several examples of this scenario.

This leads to the main result of this section, which is analogous to Theorem 1 for simple loops:

Theorem 2. *Let* Π *be an abacus program, all of whose strongly connected components are simple loops with monotone shortcuts. Let* S *be any node in the program, and* $\bar{F}$ *a vector of register values. We can then compute a disjunction of linear constraints on the initial register values for reaching* S *with the register values* $\bar{F}$*. If all simple loops with shortcuts in* Π *are order independent, the obtained precondition is necessary and sufficient.*

Proof. Similar to the proof by decomposition for Theorem 1, using propositions 4 and 5. □

Semantics of the Computed Conditions. Since we are working in the setting where non-deterministic actions are allowed, the variable k_i may implicitly capture the number of times particular outcomes of non-deterministic actions present in $loop_i$ must occur during its k_i iterations. This may appear to be measuring an inherently unpredictable property (non-determinism) and seem to mitigate the utility of the computed preconditions. However, as we will see in Section 8, non-deterministic abacus actions may stand for sensing actions; while we may not be able to predict the outcome of each sensing action, it may still be possible to know how many times a certain outcome is possible, which is all that we need to use the conditions above. In addition, if k_i's are used as parameters, the conditions above capture their tolerable values under which a desired register value may be achieved.

5.2. Deterministic Monotone Shortcuts

In the previous section we addressed the problem of determining when a program can reach a certain state with a given register vector by deriving constraints between the initial and final register values for a given abacus program. In order to achieve these results, we used the concept of order independence to summarily deal with a collection of simple loops and the number of times each had to be executed. In deterministic programs however, the set of simple loops obtained by decomposition exhibit two complementary properties. On one hand, they are highly order *dependent*, in that every subsequent loop to be executed will be precisely determined by the initial register value and the iterations executed so far. On the other hand, as we will see below, the exact number of iterations of each loop that will be executed can be computed easily. A naive approach for constructing applicability tests in this case would be to apply the conditions developed in the previous section directly to this setting.

From this point of view, deterministic actions can be seen as introducing an extreme form of order dependence with extensive use of non-spurious equality conditions. However, we can utilize determinism in this setting to develop a more precise, but less general applicability test.

We now address the problem of determining if a program will terminate with a given register vector by designing an algorithm which takes as input an initial register vector, and provides a yes/no answer. More precisely, the algorithm will efficiently compute the final register vector for the given initial register vector. Without loss of generality, we consider this problem in the setting where we have a single simple loop with shortcuts and the start node for the program is the start node of this loop.

Our approach relies on the following observations:

1. Because of monotonicity, if a loop is executed for a certain number of iterations and then exited, flow of control will never return to that loop.
2. For any given configuration of register values at the start node, at most one of the simple loops in the given loop's decomposition may be completely executable. This is because if multiple simple loops can be executed starting from a given register value configuration, then at some action node in the program, it should be possible for the control to flow along more than one outgoing edge. However, this is impossible because every action which has multiple outcomes (a decrementing action) has exactly two branches, whose conditions are always mutually inconsistent.

As a consequence of the second observation, given such an abacus program and an initial register vector, we can compute the first loop which will be executed and the number of iterations for which it will be executed (the precise method for computing this is described below); we can then remove this loop from consideration because of the first observation and repeat the process. This can be continued until no loop can be executed completely. When this process terminates, we get the sequence of loops and the number of iterations of each that *must* be executed before exiting the given simple loop with shortcuts.

Taking an initial register valuation as input, Alg. 1 perfoms these computations. Let Π_A be an abacus program in the form of a simple loop with monotone shortcuts and only deterministic actions. Alg. 1 works by identifying the unique loop ℓ whose $\mathsf{LoopIneq}_\ell$ is satisfied by the value $\bar{R}$ (initialized to $\bar{R^0}$) [steps 5-8], calculating the number of iterations which will be executed for that loop until $\mathsf{LoopIneq}_\ell$ gets violated [step 9], updating the register values to reflect the effect of those iterations [step 12] and identifying the next loop to be executed [the while loop, step 4].

The subroutine FindMaxIterations uses the inequalities in $\mathsf{LoopIneq}_\ell$ (see Prop. 2) to construct the vector equation $(\bar{R} + \ell_{max}\Delta^\ell + \Delta^{1..i-1})_{a(i)} \circ 0$ for every action in loop l. This system of equations consists of an inequality of the following form for every i corresponding to a decrementing action in the loop:

$$\ell_{max} < (\bar{R}_{a(i)} + \Delta^{1..i-1}_{a(i)})/\Delta^\ell_{a(i)}$$

Algorithm 1: Reachability for deterministic, monotone shortcuts

Input: Deterministic abacus program in the form of a simple loop with monotone shortcuts with start node S_{start}, an initial register configuration $\bar{R}^0$

Output: Sequence of (loop id, #iterations) tuples and final value of $\bar{R}$ at S_{start}.

```
1  R̄ ← R̄⁰
2  Iterations ← empty list
3  LoopList ← simple loops created by shortcuts
4  while LoopList ≠ ∅ do
5  |   if no ℓ ∈ LoopList satisfies LoopIneq_ℓ(R̄) then
6  |   |   Return Iterations
   |   end
7  |   ℓ ← id of loop for which LoopIneq_ℓ(R̄) holds
8  |   Remove ℓ from LoopList
9  |   ℓ_max ← FindMaxIterations(R̄, ℓ)
10 |   if ℓ_max = ∞ then
   |   |   Return "Non-terminating loop"
   |   end
11 |   Iterations.append((ℓ, ℓ_max))
12 |   R̄ ← R̄ + ℓ_max Δ^ℓ
   end
13 Return Iterations, R̄
```

Since $\bar{R}$ is always known during the computation, the floor of minimum of the RHS of these equations for all i yield the largest possible value of ℓ_{max}. Equality constraints either drop out (if the net change in their register's value due to the loop ℓ is zero and they are satisfied during the first iteration), or set $\ell_{max} = 1$ (if the net change in their register's value is not zero, but it is satisfied during the first iteration). Equality constraints will be satisfied when FindMaxIterations is called because we know that $\mathsf{LoopIneq}_\ell$ was satisfied. Note that if there is any loop which does not decrease any register's value, it will never terminate. This will be reflected in our computation by an ℓ_{max} value of ∞ [step 11]. Thus, we have:

Theorem 3. *Given a deterministic abacus program Π in the form of a simple loop with monotone shortcuts and start node S, and an initial register vector $\bar{R}^0$, Alg. 1 returns the number of times each simple loop in Π's decomposition will be executed as well as the register vector at S after all these iterations.*

Depending on the rest of the abacus program, the final register vector ouput by Alg. 1 can be used as the initial register vector for determining the reachability of a subsequent state with a desired register vector.

Complexity Analysis. Let b be the maximum number of branches in a loop in the decomposition of the given simple loop with shortcuts, and L the total number of simple loops in the decomposition. The most expensive operation in

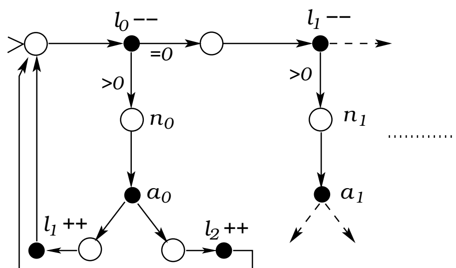


Figure 3: Construction for translating a general abacus program into one with a simple loop with shortcuts.

this algorithm is step 5, where $\bar{R}$ is tested on every loop's inequality (these loop inequalities only need to be constructed once). Step 5 is executed in $O(Lb)$ time and step 9 in $O(b)$ time. The entire loop may be executed at most L times, resulting in a total execution time of $O(L^2b)$. On the other hand, if such a program is directly applied on a problem instance and the program terminates, then the execution time for the program will be of the order of the largest input register value, which is unbounded.

6. Relaxing Monotonocity

We now consider the problem of computing the preconditions of an abacus program with simple loops with shortcuts that need not be monotone. As noted earlier, in terms of computational expressiveness this class is very powerful. We show below that any abacus program can effectively be represented as a program consisting of one simple loop with shortcuts.

Theorem 4. *Let Π_g be an abacus program with R_g, N_g and E_g as the sets of registers, nodes and edges respectively. Then there exists an equivalent abacus program, Π_S with $R_S(\supseteq R_g)$, $N_S(\supseteq N_g)$, and E_S as the sets of registers, nodes and edges respectively, such that:*

1. *Π_S consists of one simple loop with shortcuts.*
2. *Execution of Π_g with an initial register vector $\bar{R}_{init}$ is equivalent to that of Π_S with an initial vector $\bar{R}'_{init}$: a node $n \in N_g$ is reachable with a register vector $\bar{R}_f$ in Π_g iff it is reachable in Π_S with a register vector $\bar{R}'_f$ which matches $\bar{R}_f$ on all the registers from R_g.*

Proof. In order to construct Π_S, we add a new flag register l_i for each $n_i \in N_g$. The values of these flag registers will never rise above 1; at any stage during execution, at most one of the flag registers will be non-zero. We will use these flags to translate edges from Π_g into a set of "case statements" starting with a common, new start node.

Construction of Π_S. Let $n_0 \rightarrow a_0$, $a_0 \rightarrow n_1$, $a_0 \rightarrow n_2$ be a set of edges corresponding to a single (decrementing) action-node a_0 in Π_g. We translate this sequence into a sequence beginning with the action node decrementing l_0. The > 0 branch from this action represents the case that we were at state n_0. This branch will lead to the node for a_0; the two branches from a_0 lead to actions incrementing l_1 and l_2, corresponding to the branches that lead to n_1 and n_2. The construction is illustrated in Fig. 3. The translation is similar for incrementing actions. To get Π_S, we perform this construction for the edges corresponding to each action in Π_g in this manner and attach the each resulting graph to the $= 0$ branch of the last flag decrementing action, as shown in Fig. 3. The resulting abacus program Π_S consists of one simple loop with shortcuts with the new start node as the common start node for the shortcuts.

Computation of $\bar{R}'_{init}$. The initial values for all the original registers R_g are the same as those in $\bar{R}_{init}$; the flag register corresponding to Π_g's start node is initialized as 1 and all the other flag registers are initialized as 0.

By construction, executing an action-node on a register vector leads to a node n_i in Π_g iff executing that action-node on the extended register vector with all flag variables zero (note that the flag-testing action also decrements the only non-zero flag to zero) leads to l_i, and subsequently, n_i in Π_S. By induction on path lengths, we therefore have the result that a node n_i is reachable from $\bar{R}_{init}$ in Π_g iff it is reachable from $\bar{R}'_{init}$ in Π_S. □

Simple loops with non-monotone shorcuts are therefore sufficient to capture the power of Turing machines:

Corollary 1. *The class of abacus programs whose strongly connected components are simple loops with shortcuts is Turing-complete.*

Removing the condition of monotonicity therefore makes the problem of computing preconditions of abacus programs with simple loops with shortcuts undecidable. Currently, there are no intermediate characterizations of simple loops with shortcuts that bridge the gap between monotone shortcuts, where this paper demonstrates the existence of efficient methods for finding preconditions, and non-monotone loops where the problem becomes undecidable. An important direction for future work is to identify useful, yet tractable generalizations of the notion of monotonicity where preconditions can be computed.

7. Transforming Plans into Abacus Programs

The previous sections presented methods for finding preconditions for various classes of abacus programs. Abacus programs can express any computation, including plans with PDDL actions. However, a translation of such plans into abacus programs is unlikely to employ only the kind of loops discussed above. But, if planning actions can be treated as actions that increment or decrement counters, the techniques developed above can be directly applied. We have been developing an approach for planning with abstract states which can yield such a

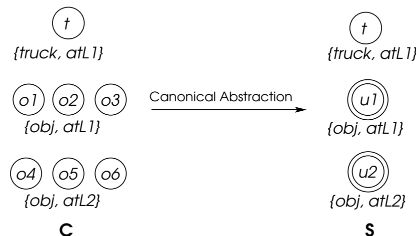


Figure 4: A concrete state and its abstraction in a unary formulation of the transport domain.

framework automatically (Srivastava et al., 2010c). In this section, we illustrate the relevant concepts of this approach with an example.

Consider a restricted version of the transport domain with one truck of capacity one and two locations, formulated using only unary predicates. The vocabulary $\mathcal{V}$ of this domain consists of the unary predicates *obj, atL1, atL2, inT* and the constant *truck*. *obj(x)* signifies that x is an object which can be loaded into the truck; *atLi(x)* and *inT(x)* denote that x is at location Li and in the truck, respectively.

The left part of Fig. 4 shows a state in this domain, represented as a logical structure (C). C's universe has 7 elements ($\{t, o1, o2, o3, o4, o5, o6\}$) which are drawn with circles and annotated with the set of predicates that they satisfy and the constants, if any, that they represent. The structure C therefore represents a state with 3 objects at each location and the truck at $L1$. For the purpose of this illustration, we only need the single action *loadT(c)*, which loads crate c into the truck.

Overview of canonical abstraction. We now illustrate how state abstraction can reduce actions on concrete states like C to actions which only increment or decrement certain counters. The right part of Fig. 4 shows structure S, the result of applying a state abstraction technique called *canonical abstraction* (Sagiv et al., 2002) on C. We first define the *role* of an element in a structure as the set of unary predicates that it satisfies and the set of constants that it represents.

Canonical abstraction collapses all elements of a concrete structure that have the same role into a single collective element called a *summary element* of that role. As opposed to singleton elements drawn with circles, we will use double circles to represent summary elements in diagrams. Thus, element u_1 for instance is a summary element which represents all objects with the role $\{$*obj, atL1*$\}$ in C (viz., $o1, o2, o3$). A more formal description of canonical abstraction which also works for structures with predicates of higher arities is presented by Sagiv et al. (2002); an overview of this approach in the context of AI planning was presented in previous work (Srivastava et al., 2010c).

Abstract structures represent sets of concrete structures. A summary element of a certain role in an abstract structure denotes that there is at least one element with that role in all concrete structures represented by the abstract

structure. In this way, abstract structures like S represent unbounded collections of concrete structures by introducing uncertainty about the number of elements of each role: S represents the set of all concrete states with exactly one truck, at least on object at L1, and at least one object at L2.

Choice operations on abstract structures. For the purpose of applying planning actions however, we need to extend this technique with a method for selecting individual action arguments from the summary elements representing them. The "drawing-out" mechanism developed in our prior work (Srivastava et al., 2010c) accomplishes this. This mechanism, illustrated in Fig. 5, introduces choice actions which take an abstract structure S and a role r as arguments. Choice actions assign a new, "operand" constant to a single member of the set of all elements with role r represented by S.

A choice operation for selecting a concrete element represented by a summary element has two possible outcomes due to the imprecision in quantities of objects represented by summary elements. This is shown in Fig. 5 in structures S_2 and S_3. These two cases correspond to whether or not the original summary element represents exactly one element. For the case where it does, the summary element is replaced by a singleton and this singleton element is assigned the operand constant (S_3); otherwise, a new element is "drawn-out" from the summary element and assigned the operand constant (S_2), corresponding to the case where the original summary element represented more than one concrete elements. Therefore, *the outcomes possible on application of a choice operation on an abstract structure can be categorized in terms of inequalities between the cardinality of summary elements, or role-counts in the initial abstract structure, and the constant 1.*

Measuring action effects via changes in role-counts. The top row in Fig. 5 shows the changes in each role-count caused due to action application. In this example, the action *loadT* decreases the role-count of the role $\{$*obj, atL1, inT*$\}$ by 1 and makes a corresponding increase in the role-count of $\{$*obj, atL1*$\}$. The set of actions in this domain are therefore similar in function to abacus actions: increments and decrements in role-counts occur in unit amounts; if an action may have multiple outcomes, they are categorized in terms of the comparison of a role-count with the constant 1.

In previous work (Srivastava et al., 2010c) we identified a class of domains called extended-LL domains[1] where actions always take such forms after canonical abstraction. We summarize the relevant properties of extended-LL domains as follows:

Theorem 5. (Extended-LL Domains) *(Srivastava et al., 2010c) Let S_0 be a canonically abstracted state in an extended-LL domain, and let $\{S_1, \ldots, S_n\} = a(S_0)$ be the possible results of applying action a on S_1. Then:*

[1] As a special case, all representations of PDDL planning domains using only unary predicates qualify as extended-LL domains.

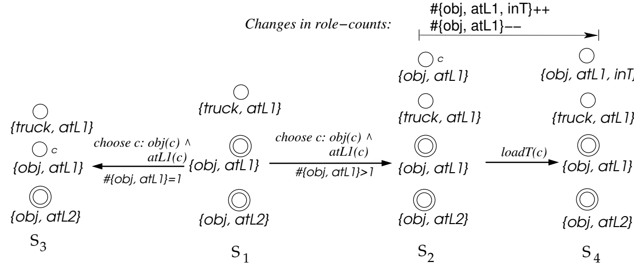


Figure 5: A sequence of actions in a unary representation of transport domain.

1. *For each $S_i \in a(S_0)$, the changes in role-counts due to action a_1 can be listed using a set of increment tuples $\{(R_1^+, c_1), \ldots (R_k^+, c_k)\}$ denoting that role-count of R_i^+ is increased by c_i and a set of roles that are decremented by 1, say $\{R_1^-, \ldots R_m^-\}$.*
2. *The conditions under which a particular result S_i will occur are conjunctions of inequalities between role-counts in S_0 and the constant 1.*

Moreover, the role-count changes caused due to actions can be efficiently computed given the action description and the abstract structure on which the action is applied. Actions on abstract structures in extended-LL domains can therefore be easily translated into sequences of abacus actions, and vice-versa. In addition to the transport domain discussed here, further examples of such domains are presented in the Section 8.

Lemma 2. *Let $S_1 \xrightarrow{a_1} S_2$ be an action operation in an extended-LL domain, where S_2 represents only one of the possible outcomes of action a_1 when applied on S_1.*

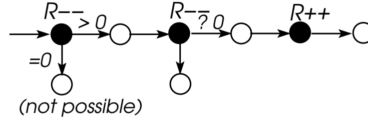
This operation can be translated into a linear abacus program Π_a whose start node is labeled S_1 and terminal node is labeled S_2.

Proof. Using Theorem 5, the changes in role-counts due to any action a on an abstract structure S_0 in an extended-LL domain can be listed as a set of increments: $\{(R_1^+, c_1), \ldots (R_k^+, c_k)\}$ and a set of roles decremented by 1 $\{R_1^-, \ldots R_m^-\}$.

Starting with an initial node for Π_a labeled S_0, we first add sequences of c_i abacus action nodes for each role R_i that needs to be incremented, with new intermediate nodes. Let the final state-node obtained after this operation be ns_I.

We then need to add abacus operations simulating role decrements. First, for each decrement to be conducted we identify the branch ($R_i^- > 1$ or $R_i^- = 1$) that was taken in the given action operation (this is possible because of the second part of Theorem 5). Starting at ns_I, we add sequences of abacus operations corresponding to each decrement operation. Since abacus actions can only conduct comparisons with zero, each of these sequences consists of the following 3 actions as shown in the figure below: one decrement operation for

the role, an extra decrement to conduct a comparison with zero instead of with 1, and finally, an increment operation to reverse the extra decrement.



Here, the comparison operation after the second decrement is the the operation identified in the given example. Chaining such role-decrementing sequences of operations one after the other, starting at ns_I gives us a linear abacus program simulating $S_1 \xrightarrow{a_1} S_2$. □

A plan with extended-LL domain actions can therefore be converted into an Abacus program without changing its structural complexity (its loop structure). The method for computing preconditions of loops of abacus actions can therefore be used for plans with simple loops in extended-LL domains.

A similar structure preserving translation can be used to translate abacus actions into sequences of extended-LL domain actions which use roles as registers. Note that in extended-LL domains, an action which increases a role-count also necessarily decrements some other role-count. Therefore, in order to simulate abacus actions that increment a register without a corresponding decrement, we add an extra role R_∞ from which all the action simulating increments can delete objects. Thus, we have:

Lemma 3. *Linear segments of abacus programs can be simulated by linear segments of programs in extended-LL domains and vice versa.*

Corollary 2. *Plans with extended-LL domain actions can simulate Abacus programs without increasing the loop complexity and vice versa.*

Note however, that plans in extended-LL domains tend to be more compact since a single action can update many role-counts, with increments larger than 1.

Theorem 6. *Plans with extended-LL domain actions are Turing complete.*

Extended-LL domains thus represent a powerful class of planning domains. Their action operations, however, are fundamentally simple and can be analyzed along the lines developed in the previous sections.

The next section shows a range of problems which can be represented in the form of extended-LL domains, and whose actions can be treated as abacus actions. As a result, preconditions and termination guarantees of a wide range of plans with loops in these domains can be computed very efficiently. We also demonstrate our approach on plans with complex loops created by nondeterministic sensing actions.

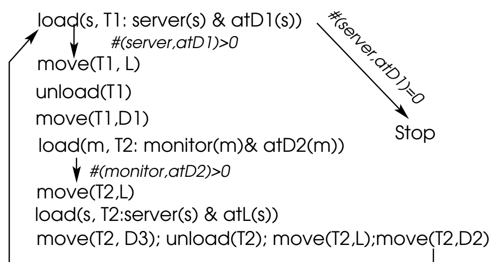


Figure 6: Solution plan for the transport problem

8. Example Plans and Preconditions

We implemented the algorithm for finding preconditions for simple loops and order independent nested loops due to shortcuts, and applied it to various plans with loops that have been discussed in the literature. Existing approaches solve different subsets of these problems, but almost uniformly without computing plan preconditions or termination guarantees. For nested loops, our implementation takes a node in a strongly connected component as an input and computes an appropriate start node. It then decomposes the component into independent simple loops and computes the preconditions. Table 1 shows timing results for 10 different plans.

Plan Representation. Figs. 6, 7 and 8 show solution plans for some of the test problems. In order to make the plans easy to read, we show only action nodes. The default flow of control continues line by line (semi-colons are used as line-breaks). Edges are shown when an action may have multiple outcomes and are labeled with the conditions that must hold *prior* to action application for that edge to be taken (as with abacus programs). Only the edges required by the plan are drawn; the preconditions must ensure that these edges are always taken. For clarity, in some cases we label only one of the outcomes of an action, and the others are assumed to have the complement of that label. Actions are written as "ActionName(args:argument-formula(args))". Any object satisfying an action's argument formula may be chosen for executing the plan. The desired halt states are indicated with the action "Stop".

Transport. In the transport problem (Srivastava et al., 2008) two trucks have to deliver sets of packages through a "Y"-shaped roadmap. Locations D1, D2 and D3 are present at the three terminal points of the Y; location L is at the intersection of its prongs. Initially, an unknown number of servers and monitors are present at D1 and D2 respectively; trucks T1 (capacity 1) and T2 (capacity 2) are also at D1 and D2 respectively. The goal is to deliver all objects to D3, but only in pairs with one of each kind.

The problem is modeled using the predicates $\{server,\ monitor,\ atD_i,\ inT_i,\ atL,\ T1,\ T2\}$. As discussed in the previous section, role-counts in this representation can be treated as register values and actions as abacus actions on these

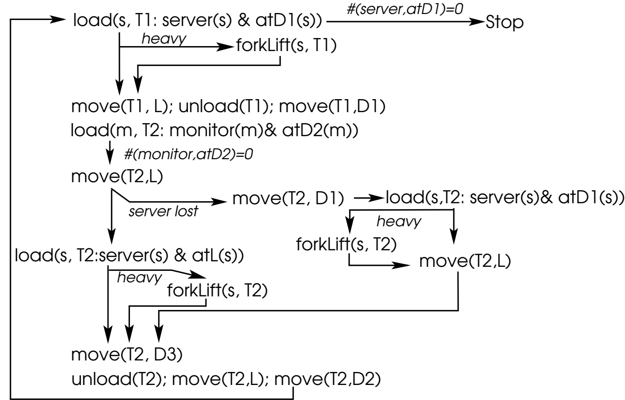


Figure 7: Solution plan for the conditional version of transport

roles. The plan shown in Fig. 6 first moves a server from D1 to L using T1. T2 picks up a monitor at D2, moves to L, picks up the server left by T1 and transports both to D3. The first action, *load*, uses as its arguments an object s (satisfying $server(s) \wedge atD1(s)$), and the constant T1 representing the truck T1. It decrements the count of the role $\{server,\ atD1\}$ and consequently has two outcomes depending on its value. Note that the second load action in the plan also has two outcomes, but only the one used in the plan is shown. In order to reach the Stop state with the goal condition, we require that final values of $s_1 = \#\{server,\ atD1\}$ and $m_2 = \#\{monitor,\ atD2\}$ be zero. Let $s_3 = \#\{server,\ atD3\}$ and $m_3 = \#\{monitor,\ atD3\}$. The changes caused due to one iteration of the loop are $+1$ for m_3, s_3 and -1 for s_1, m_1. Using the method developed in proposition 2, the necessary and sufficient condition for reaching the goal after l iterations of the loop is that there should be equal numbers of objects of both types initially: $m_2^0 = l = s_1^0$.

Transport Conditional. In the conditional version of the transport problem, objects left at L may get lost, and servers may be heavy, in which case the forkLift action has to be used instead of the load action. Fig. 7 shows a solution plan found by merging togther plans which encountered and dealt with different non-deterministic action outcomes (Srivastava et al., 2010b). If a server is not found when T2 reaches L, the plan proceeds by moving T2 to D1, loading a server, and then proceeding to D3. Note that the shortcut for the "server lost" has a sub-branch, corresponding to the server being heavy. The plan can be decomposed into 8 simple loops. Of these, 4, which use the "server lost" branch use one extra server (loops $0, 5, 6$ and 7 in the inequality below). Let role-counts s_2, m_2, s_3, m_3 be as in the previous problem. Then, the obtained applicability conditions are:
$s_3^f = m_3^f = \sum_{i=0}^{7} k_i; \quad m_2^f = m_2^0 - \sum_{i=0}^{7} k_i = 0$
$s_1^f = s_1^0 - \sum_{i=0}^{7} k_i - k_0 - k_5 - k_6 - k_7 = 0$
These conditions show that every possible loop decrements the role-counts s

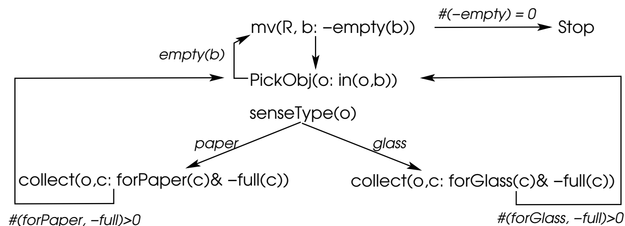


Figure 8: Solution plan for the recycling problem

and m; however, in order to have all objects at D3 the conditions now require extra servers to be kept at D1, amounting to the number of times a server was lost.

Recycling. In this problem a recycling agent must inspect a set of bins, and from each bin, collect paper and glass objects in their respective containers. The solution plan includes nested loops due to shortcuts (Fig. 8), with the start node at *PickObj*. *senseType* is a sensing action, and the collect actions decrement the available capacity of each container, represented as the role-count of $\{forX, \neg full\}$ where X is paper or glass. Let *e, fg, fp, p, g* denote the role-counts of non-empty bins, glass container capacity, paper container capacity, paper objects and glass objects respectively. Let l_1 denote the number of iterations of the topmost loop, l_2 of the paper loop and l_3 of the glass loop. The applicability conditions are:

$$e^f = e^0 - l_1 = 0, \quad fp^f = fp^0 - l_2 \geq 0,$$
$$p^f = p^0 + l_2, \quad fg^f = fg^0 - l_3 \geq 0, \quad g^f = g^0 + l_3.$$

Note that the non-negativity constraints guarantee termination of all the loops.

Accumulator. The accumulator problem (Levesque, 2005) consists of two accumulators and two actions: *incr_acc(i)* increments register i by one and *test_acc*(), tests if the given accumulator's value matches an input k. Given the goal $acc(2) = 2k - 1$ where k is the input, KPLANNER computes the following plan: *incr_acc(1);* ***repeat*** {*incr_acc(1); incr_acc(2); incr_acc(2)*}***until*** *test_acc(1); incr_acc(2)*. Although the plan is correct for all $k \geq 1$, KPLANNER can only determine that it will work for a user-provided range of values. This problem can be modeled directly using registers for accumulators and asserting the goal condition on the final values after l iterations of the loop (even though there are no decrement operations). We get

$$acc(1) = l + 1; \; acc(2) = 2l + 1 = 2k - 1.$$

This implies that $l = k - 1 \geq 0$ iterations are required to reach the goal.

Problem	Time (s)	Problem	Time(s)
Accumulator	0.01	Prize-A(7)	0.02
Corner-A	0.00	Recycling	0.02
Diagonal	0.01	Striped Tower	0.02
Hall-A	0.01	Transport	0.01
Prize-A(5)	0.01	Transport (conditional)	0.06

Table 1: Timing results for computing preconditions

Further Test Problems and Discussion. We tested our algorithms with many other plans with loops. Table 1 shows a summary of the timing results. The runs were conducted on a 2.5GHz AMD dual core system. Problems Hall-A, Prize-A(5) and Prize-A(7) (Bonet et al., 2009) concern grid world navigation tasks. In Hall-A the agent must traverse a quadrilateral arrangement of corridors of rooms; the prize problems require a complete grid traversal of $5 \times n$ and $7 \times n$ grids, respectively. Note that at least one of the dimensions in the representation of each of these problems is taken to be *unknown* and *unbounded*. Our implementation computed correct preconditions for plans with simple loops for solving these problems. In Hall-A, for instance, it correctly determined that the numbers of rooms in each corridor can be arbitrary and independent of the other corridors. The Diagonal problem is a more general version of the Corner problem (Bonet et al., 2009) where the agent must start at an unknown position in a rectangular grid, reach the north-east corner and then reach the southwest corner by repeatedly moving one step west and one step south. In this case, our method correctly determines that the grid must be square for the plan to succeed. In Striped Tower (Srivastava et al., 2008), our approach correctly determines that an equal number of blocks of each color is needed in order to create a tower of blocks of alternating colors. In all the problems, termination of loops is guaranteed by non-negativity constraints such as those above.

9. Related Work

Although various approaches have studied the utility and generation of plans with loops, very few provide any guarantees of termination or progress for their solutions. Approaches for cyclic and strong cyclic planning (Cimatti et al., 2003) attempt to generate plans with loops for achieving temporally extended goals and for handling actions which may fail. Loops in strong cyclic plans are assumed to be *static*, with the same likelihood of a loop exit in every iteration. The structure of these plans is such that it is always *possible*–in the sense of graph connectivity–to exit all loops and reach the goal; termination is therefore guaranteed if this can be assumed to occur eventually. Among more recent work, Kplanner (Levesque, 2005) attempts to find plans with loops that generalize a single numeric planning parameter. It guarantees that the obtained solutions will work in a user-specified interval of values of this parameter. Distill (Winner and Veloso, 2007) identifies loops from example traces but does

not address the problem of preconditions or termination of its learned plans. Bonet et al. (2009) derive plans for problems with fixed sizes, but the controller representation that they use can be seen to work across many problem instances. They also do not address the problem of determining the problem instances on which their plans will work, or terminate.

Finding preconditions of linear segments of plans has been well studied in the planning literature. Triangle tables (Fikes et al., 1972) can be viewed as a compilation of plan segments and their applicability conditions. However, there has been no concerted effort to find preconditions of plans with loops. Static analysis of programs deals with similar problems of finding program preconditions. However, these methods typically work with the weaker notion of *partial correctness*, where a program is guaranteed to provide correct results *if* it terminates. Methods like Terminator (Cook et al., 2006) specifically attempt to prove termination of loops, but do not provide precise preconditions or the number of iterations required for termination.

10. Conclusions and Future Work

In this paper we presented an approach for formulating and studying the problem of determining when a certain loop of actions can be guaranteed to (a) terminate, and (b) lead to a desired result. We showed how this problem can be studied effectively as the problem of reachability of desired states in the context of primitive actions that can only increase, decrease or non-determinitically change the value of some counters. Although this approach is the first to address this problem comprehensively, it is very efficient and scalable for commonly encountered loops of actions in planning. In addition to finding preconditions of computed plans, it can also be used as a component in the synthesis of plans with safe loops.

We presented several results about the trade-offs between computational expressiveness of different classes of plans or programs with such actions and the tractability of answering the reachability problem. For simple loops of actions, this problem permits very efficient algorithms; slight extensions to this class of loops (viz. simple loops with shortcuts), however, were found to be general enough to capture the full power of Turing machines and therefore had an undecidable reachability problem (Theorem 4) *in general*. On the other hand, the property of *monotonicty* in this case does permit algorithms for determining reachability, with their accuracy depending upon the notion of order dependence (Prop. 4). Order dependence itself is not very restrictive in non-deterministic situations from an adversarial point of view, where the exact sequence of non-deterministic outcomes of actions cannot be predicted, and we need to plan for the worst case.

The presented work increases our understanding of the factors that make these problems difficult: when order dependence cannot be overcome by conservative approximations, and when the property of monotonicity does not hold. Although non-monotone simple loops with shortcuts have an undecidable reachability problem in the worst case, in some cases in this setting also, the problems

of reachability, and at the least termination, can be answered. A greater identification of tractable classes of non-monotone simple loops with shortcuts is left for future work. Computation and expression of order-dependent preconditions are also important directions for future work on pushing the theoretical limits of solvability of these problems.

We showed one approach for interpreting planning actions as abacus actions in this paper. The underlying methods for determining reachability in abacus programs however, can be used whenever actions can be interpreted as incrementing or decrementing counters. Development of more general reductions, for instance by using description logic to construct roles in planning problems is also an important direction for future work.

Acknowledgments

Support for this work was provided in part by the National Science Foundation under grants IIS-0915071, CCF-0541018, and CCF-0830174.

Bonet, B., Palacios, H., Geffner, H., 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In: Proc. of the 19th International Conference on Artificial Intelligence Planning and Scheduling. pp. 34–41.

Bylander, T., 1994. The computational complexity of propositional strips planning. Artificial Intelligence 69 (1-2), 165–204.

Cimatti, A., Pistore, M., Roveri, M., Traverso, P., 2003. Weak, strong, and strong cyclic planning via symbolic model checking. Artificial Intelligence 147 (1-2), 35–84.

Cook, B., Podelski, A., Rybalchenko, A., 2006. Termination proofs for systems code. In: Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 415–426.

Eggan, L. C., 1963. Transition graphs and the star-height of regular events. Michigan Mathematical Journal 10, 385–397.

Fikes, R., Hart, P., Nilsson, N., 1972. Learning and executing generalized robot plans. Tech. rep., AI Center, SRI International.

Hammond, K. J., 1986. CHEF: A model of case based planning. In: Proc. of the 13th National Conference on Artificial Intelligence. pp. 267–271.

Lambek, J., 1961. How to program an infinite abacus. Canadian Mathematical Bulletin 4 (3).

Levesque, H. J., 2005. Planning with loops. In: Proc. of the 19th International Joint Conference on Artificial Intelligence. pp. 509–515.

Minsky, M. L., 1967. Computation: finite and infinite machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Rintanen, J., 2004. Complexity of planning with partial observability. In: Proc. of the 14th International Conference on Automated Planning and Scheduling. pp. 345–354.

Sagiv, M., Reps, T., Wilhelm, R., 2002. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24 (3), 217–298.

Shavlik, J. W., 1990. Acquiring recursive and iterative concepts with explanation-based learning. Machine Learning 5, 39–70.

Srivastava, S., Immerman, N., Zilberstein, S., 2008. Learning generalized plans using abstract counting. In: Proc. of the 23rd National Conference on AI. pp. 991–997.

Srivastava, S., Immerman, N., Zilberstein, S., 2010a. Computing applicability conditions for plans with loops. In: Proc. of the 20th International Conference on Automated Planning and Scheduling. pp. 161–168.

Srivastava, S., Immerman, N., Zilberstein, S., 2010b. Merging example plans into generalized plans for non-deterministic environments. In: Proc. of the 9th International Conference on Autonomous Agents and Multiagent Systems. pp. 1341–1348.

Srivastava, S., Immerman, N., Zilberstein, S., 2010c. A new representation and associated algorithms for generalized planning. Artificial Intelligence, To appear.

Winner, E., Veloso, M., 2007. LoopDISTILL: Learning domain-specific planners from example plans. In: Workshop on AI Planning and Learning, in conjunction with ICAPS.