Computing Applicability Conditions for Plans with Loops: New Results[⋆]

Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein

Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003
{siddharth,immerman,shlomo}@cs.umass.edu

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. We first develop these methods for abacus programs, and then show that plans in a wide variety of domains can be treated as abacus programs. 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.

1 Introduction

Recent work in planning has highlighted the benefits of using loops in plan representation [1–3]. Plans with loops 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 to determine the general conditions under which they terminate and achieve the 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 paper presents methods for efficiently determining the conditions under which plans with some classes of simple and nested loops can solve a problem instance. These methods can also be used to determine the utility of adding a loop during plan generation. We initially assume that planning actions come from a simple, but powerful class of action operators, which can only increment

[⋆] Parts of this article appeared in the proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2010.

or decrement a finite set of registers by unit amounts. Then we show that many interesting planning problems can be directly 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. That is, finding closed-form applicability conditions, or preconditions for such plans 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. Finally, we show how a recently proposed approach for finding plans with loops can be interpreted as generating abacus programs of this very class. This method can be used to translate plans with simple and nested loops in many planning domains into abacus programs, thus allowing applicability conditions to be computed for a broad range of planning problems.

We start with a formal description of abacus programs. This is followed by a formal analysis of the problem of finding preconditions of abacus programs with simple loops and a class of nested loops. We then show how plans with loops can be translated into abacus programs, and conclude with a demonstration of the scope and efficiency of these methods.

2 Abacus Programs

Abacus programs [4] 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 $\ell : \mathcal{S} \setminus \{s_h\} \mapsto \mathrm{Act}$. 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 states to actions and from actions to states. In order to distinguish abacus program states from states in planning, we will refer to a state in the graph of an abacus program as a "node". The two edges out of a decrement action are labeled $= 0$ and > 0 respectively (see Fig. 1).

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. Abacus programs are equivalent to Minsky Machines [5], 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, for some abacus programs halting *is* decidable, depending on the complexity of the loops. A *simple loop* is a cycle. A *simple-loop* abacus program is one all of whose non-trivial strongly connected components are simple loops. In the next section we 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 with a desired set of register values (Th. 1).

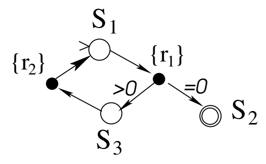

Fig. 1. A simple abacus machine for the program: `while` $(r_1 > 0)$ { $r_1 - -; r_2 + +$}

2.1 Applicability Conditions for 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, $\bar{R}^0 = \langle R^0_1, R^0_2, \ldots, R^0_m\rangle$ 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 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. (Sketch) 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. This can be done by computing the register values at each node S_i in terms of $\bar{R}^0$, and stating the inequality required for the desired branch of the next action.

Proposition 2. *Suppose $S_1, a_1, \ldots, S_n, a_n, S_1$ is a simple loop of an abacus program. Then in $O(n)$ time we can compute a set of linear constraints, $C(\bar{R}^0, \bar{F}, l)$, that are satisfied by initial and final register tuples ($\bar{R}^0$ and $\bar{F}$), and the 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

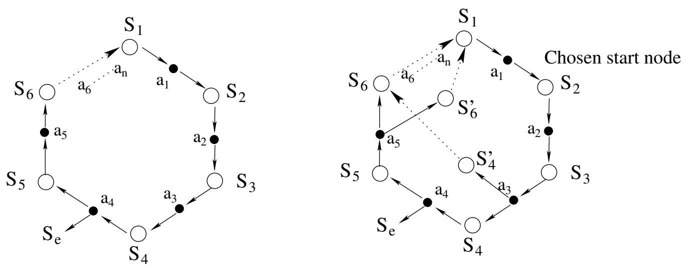

Fig. 2. A simple loop with (right) and without (left) shortcuts

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* $k \in \{0, \dots, l-1\}$. More precisely, for one full execution of the loop starting with $\bar{R}^0$ we require, for all $i \in \{1, \dots, n\}$:

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

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 $\bar{R}^l = \bar{R}^0 + l \times \Delta^{1..n}$, the register vector after l complete iterations. Thus, for executing the loop completely l times, the required conditions are $\mathsf{LoopIneq}(\bar{R}^0) \wedge \mathsf{LoopIneq}(\bar{R}^{l-1})$. 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 $\bar{R}^{l-1}$ 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: $C(\bar{R}^0, \bar{F}, l) \equiv \mathsf{LoopIneq}(\bar{R}^0) \wedge \mathsf{LoopIneq}(\bar{R}^{l-1}) \wedge (\bar{F} = \bar{R}^l)$. 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 $R^l = F$; 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 outlined 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). 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.

2.2 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. Characterizing the precise boundary between decidability and undecidability of abacus programs in terms of their structural complexity is an important open problem.

In this section, we show that methods developed in the previous section *can* be extended to a class of nested loops caused due to non-deterministic actions. Non-deterministic actions are common in planning but do not exist in the original definition of abacus domains. In the representation of Def. 1, we define a non-deterministic action in an abacus program $NSet(r, s_1, s_2)$ as follows:

– $NSet(r, s_1, s_2)$: set r to 0 and goto $s_1 \in S$ **or** set r to 1 and goto $s_2 \in S$.

We assume that the register r is new, or unused 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, the inclusion of non-deterministic actions allows us to conveniently express sensing actions encountered in partially observable planning domains.

Definition 2. (Complex Loops) *A* complex *loop in a graph is a non-trivial strongly connected component that is not a simple loop.*

Definition 3. (Shortcuts) *A* shortcut *in a simple loop is a linear segment of nodes (without branches) starting with a branch caused due to a* non-deterministic *action in the loop and ending at any subsequent node in the loop in the direction of the loop, but not after a* designated *start node. The origin of every shortcut must occur at or after the start node.*

In Fig. 2, S_2 can be treated as a start node. The definition above constrains shortcuts to originate from a branch of a non-deterministic action; shortcuts beginning with branches of deterministic, register-decrementing actions are considered in sections 2.5 and 2.5.

In terms of graph structure, simple loops with shortcuts categorize the class of graphs with *cycle rank* [6] one. This class of graphs captures many common control flows, including those with doubly nested loops and nested `for` loops such as:

`for i=1 to n do {for j=1 to k do {xyz}}`. Actions which create shortcuts in such loops can be easily transformed into non-deterministic actions followed by actions with the original conditions.

2.3 Applicability Conditions for Monotone Shortcuts

In the rest of this paper we consider a special class of simple loops with shortcuts, where the shortcuts are *monotone*:

Definition 4. (Monotone Shortcuts) *The shortcuts of a simple loop are* monotone *if the sign (positive or negative) of the net change, if any, in a register's value is the same due to every simple loop created by the shortcuts.*

For ease of exposition we require that the start nodes of all shortcuts in a simple loop occur either at the common start node, or before the end node of any other shortcut, making shortcuts non-composable (i.e., only one shortcut can be taken in every iteration). Non-composability allows us to easily count the simple loops caused due to shortcuts independently. For instance, we can view the loop with shortcuts in Fig. 2 as consisting of 3 different simple loops. Which loop is taken during execution will depend on the results of non-deterministic actions a_3 and a_5. Additionally, we will only consider the case where non-deterministic actions occur on the outer, simple loop. Composable shortcuts and branches caused due to non-deterministic actions on shortcuts can be handled similarly by considering all possible completions of the loop independently, as simple loops. However, this may result in exponentially many simple loops in the worst case.

Suppose an abacus program Π is a simple loop with m monotone shortcuts and a chosen 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 shortcuts $1, \ldots, m$ are taken, respectively. The final, partial iteration and the loop exit can be along any of the shortcuts, or the outer simple loop, and can be handled as a linear program segment. Let k_0 be the number of times the underlying simple loop is executed without taking any shortcuts. Then,

$$k_0 + k_1 + \ldots k_m = l. \tag{1}$$

Determining Final Register Values We denote the loop created by taking the i^{th} shortcut as $loop_i$, and the original simple loop taken when none of the shortcuts are taken as $loop_0$. The final register values after the $l = \sum_{i=0}^{m} k_i$ complete iterations can be obtained by adding the changes due to each simple loop, with Δ^{loop_i} denoting the change vector due to $loop_i$:

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

Cumulative Branch Conditions For computing sufficient conditions on the achievable register values after $k_0, \ldots, k_m$ complete iterations of the given loops, the 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 $k_0, k_1, \dots k_m$ iterations of loops $0, \dots, m$, and constrain that value to be greater than zero. For every register R_j, we first identify the index of simple loop which can cause the greatest negative change in a single, partial iteration starting at S_{start}, as $min(j)$, and the value of this change as $\delta_{min(j)}$. For readability we will use $\widehat{\jmath}$ to denote $min(j)$.

Let R^+ and R^- be the sets of registers undergoing *net* positive and negative changes respectively, by any loop. For $R_j \in R^+$, the lowest possible value is $R_j^0 + \delta_{\widehat{\jmath}}$. The required constraint on R_j is simply $R_j^0 + \delta_{\widehat{\jmath}} \geq 0$ ("$\geq$" because "$>$" must hold *before* the decrement), since the value of R_j can only increase after the first iteration. For $R_j \in R^-$, the lowest possible value is $R_j^0 + \sum_{i \neq \widehat{\jmath}} k_i \Delta^{loop_i} + (k_{\widehat{\jmath}} - 1)\Delta^{loop_{\widehat{\jmath}}} + \delta_{\widehat{\jmath}}$, achieved when $loop_{\widehat{\jmath}}$ is executed at the end, after all the iterations of the other loops. Therefore, we get:

$$\forall R_j \in R^- \{ R_j^0 + \sum_{i=0}^{m} k_i \Delta^{loop_i} + \delta_{\widehat{\jmath}} - \Delta^{loop_{\widehat{\jmath}}} \geq 0 \}$$
$$\forall R_j \in R^+ \{ R_j^0 + \delta_{\widehat{\jmath}} \geq 0 \}$$

These conditions can be extended to include equality conditions for the first and last iteration of each loop. 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, $loop_{\widehat{\jmath}}$ will be executed at least once. In [7], we show how these constraints can be made more accurate by using a disjunctive formulation for selecting the loop causing the greatest negative change among those that are executed at least once. The accuracy of the resulting conditions can be analyzed in terms of order independence:

Definition 5. (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 the shortcuts are taken.*

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.*

We refer the reader to [7] for details on these results; we conclude this section by noting that the conditions developed above capture the tolerable values of k_i under which a desired combination of register values may be achieved.

2.4 Relaxing Monotonocity

Although non-deterministic actions make it easier to express plans with sensing actions, they significantly add to the power of abacus programs consisting of simple loops with shortcuts. Specifically, we show below that reachability in an abacus program consisting of a simple loop with non-monotone shortcuts is at least as hard as the problem of reachability in a vector addition system [8]. Vector addition systems are not as powerful as Turing machines, but still have a hard reachability problem. Although it has been proved that reachability in vector addition systems is decidable, known algorithms require non-primitive-recursive space [9]. We use the formalization of vector addition systems from [8]:

Definition 6. (Vector addition systems) *An n-dimensional vector addition system (VAS) is a pair (x, W) where $x \in \mathbb{N}^n$ is called the start point and W is a finite subset of $\mathbb{Z}^n$. The* reachability set *of the VAS (x, W) is the set of all z, $z = x + v_1 + \cdots + v_j$ where each $v_i \in W$ and all the intermediate sums, $x + v_1 + \cdots + v_i$, $1 \leq i \leq j$ are non-negative.*

Proposition 3. *Determining the set of register values reachable at the start node of an abacus program consisting of a simple loop with shortcuts is at least as hard as the problem of determining the reachable register values in a vector addition system.*

Proof. Suppose (x, W) is an n-dimensional VAS. We construct an abacus program with n registers, one for each dimension in the given VAS. We can then construct a simple loop with shortcuts so that each of the simple loops created corresponds to a unique $w \in W$. The shortcut corresponding to w_i would consist of sequences of incrementing/decrementing actions for each dimension in w_i (see Fig. 3). For decrementing actions, the zero-branches lead to an exit from the loop to a trap state. In the resulting abacus program, a given configuration of register values is reachable at the start state iff there exists an ordering of the simple loops created by shortcuts which leads to it. In other words, every reachable register-value configuration corresponds to a sum of the w_i's with none of the intermediate values being negative.

2.5 Simple Loops with Shortcuts due to Deterministic Actions

We now discuss simple loops with shortcuts in settings with deterministic actions.

Monotone Shortcuts Let Π_A be an abacus program in the form of a simple loop with shortcuts originating at deterministic (decrementing) actions. Taking an initial register valuation as input, Alg. 1 computes a sequence of tuples representing the order in which the simple loops created by shortcuts will be taken, and the number of times each such loop will be executed in this ordering. Such a sequence is sufficient to calculate all the reachable register values during an execution of the given program.

Algorithm 1: Reachability for deterministic, monotone shortcuts

```
Input: Deterministic abacus program in the form of a simple loop with
       monotone shortcuts, an initial register configuration $\bar{R}^0$
Output: Sequence of (loop id, #iterations) tuples.
1  $\bar{R} \leftarrow \bar{R}^0$
2  Iterations ← empty list
3  LoopList ← simple loops created by shortcuts
4  while LoopList ≠ ∅ do
5      if no $l \in$ LoopList satisfies $\mathsf{LoopIneq}_l(\bar{R})$ then
6          Return Iterations
       end
7      $l \leftarrow$ id of loop for which $\mathsf{LoopIneq}_l(\bar{R})$ holds
8      Remove $l$ from LoopList
9      $l_{max} \leftarrow$ FindMaxIterations($\bar{R}, l$)
10     Iterations.append(($l, l_{max}$))
11     $\bar{R} \leftarrow \bar{R} + l_{max}\Delta^l$
   end
12 Return Iterations
```

Alg. 1 relies on the following observations:

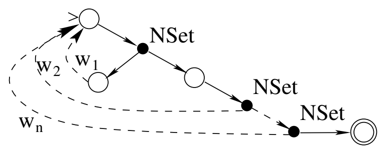

Fig. 3. Reduction of a vector addition system to a non-deterministic abacus program.

1. Because all the shortcuts are monotone, if a loop is executed for a certain number of iterations and then exited, the 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 created by shortcuts 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*.

The overall algorithm works by identifying the unique loop l whose $\mathsf{LoopIneq}_l$ 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 $\mathsf{LoopIneq}_l$ gets violated [step 9], updating the register values to reflect the effect of those iterations [step 11] and identifying the next loop to be executed [the while loop, step 4].

The subroutine FindMaxIterations uses the inequalities in $\mathsf{LoopIneq}_l$ (see prop. 2) to construct the vector equation $(\bar{R} + l_{max}\Delta^l + \Delta^{1..i-1})_{a(i)} \circ 0$ 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:

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

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 $l_{max} = 1$ (if the net change in their register's value is not zero, but it is satisfied during the first iteration). 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 ∞.

Let b be the maximum number of branches in a loop created due to the shortcut, and L the total number of simple loops generated due to the shortcuts. The most expensive operation in 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 register value.

Non-monotone Shortcuts and Linear Hybrid Automata Currently, the complexity and decidability of the problem of reachability for abacus programs consisting of simple loops with non-monotone, deterministic shortcuts is unknown. In general, reachability problems for abacus programs can be easily represented as reachability problems for linear hybrid automata (LHA) [10]. While a hybrid system is a model of computation which combines discrete state transitions with continuous flows of real-valued variables within each state, in linear hybrid automata, the flows of these variables are constrained by linear expressions. Numerous implementations of approximate and partially decidable algorithms have been developed for model checking linear hybrid systems. Deterministic abacus programs with simple loops with shortcuts can be easily represented as particularly simplified linear hybrid systems, with register changes occurring as "jump" transitions between discrete states. We have obtained promising results using LHA analysis tools on these translated representations. A detailed analysis of their applicability, as well as the impact of monotonicity and our restrictions based on graph structure on LHA verification algorithms is left for future work.

The following table summarizes known results about determining the set of states reachable from a given initial state for abacus programs with simple loops with shortcuts:

	Deterministic	Non-deterministic
Monotone Shortcuts	Alg. 1	Eq.(1-4) [7](order-indep.)
Non-monotone Shortcuts	unknown	VAS$\preceq$

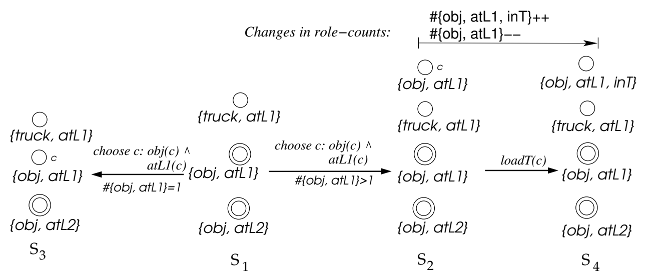

Fig. 4. A sequence of actions in a unary representation of transport domain. Predicate *object* is abbreviated as *obj.*

3 Transforming Plans into Abacus Programs

In the previous section we showed how to find preconditions for a class of abacus programs. Abacus programs can express any computation, including plans with PDDL [11] 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 recently developed an approach to accomplish that called ARANDA [12].

We illustrate the relevant concepts of ARANDA with an example. ARANDA uses *canonical abstraction* [13] to create abstract states by collecting elements satisfying the same sets of unary predicates into *summary elements.* The set of predicates satisfied by an element is called the element's *role.* Consider a simplified transport domain where objects need to be moved from L1 to L2 by a single truck of capacity one. The vocabulary for this domain consists of unary predicates {*atL1, atL2, inT, object, truck*}. Fig. 4 shows a sequence of actions applied on the initial abstract state S_1. Summary elements are drawn in the figure using double circles; S_1 has two summary elements, with roles {*object, atL2*} and {*object, atL1*}. A summary element of a certain role indicates that there may be *one or more* elements of that role. Singleton elements (such as the truck with the role {*truck, atL1*}) are drawn using single circles, and indicate that there is *exactly one* element of that role. S_1 thus represents a situation with unknown numbers of objects at L1 and L2, and exactly one truck, at L1.

Planning actions in this framework become actions that increment or decrement *role-counts*, or the number of elements satisfying certain role(s). Action *loadT(x)* in Fig. 4 loads object x into the truck. For such actions which require arguments, ARANDA "draws-out" a representative element of a role from its summary element if the role is not represented by a singleton. This results in two cases: either the drawn out element was the only one with its role, or there are other elements which have this role. This is illustrated by the *choose* action in Fig. 4, which has two possible outcomes corresponding to the number of elements, or the role-count of the role {*object, atL1*}. Note that the intermediate states S_2 and S_3 after choice and before action application do not differ in any predicates–the drawn out element is marked with a constant–and thus have the same role-counts. The combination of *choose* and *loadT* on the other hand is exactly like an abacus action application except that this combined operation conducts a comparison with 1 instead of 0 during decrementing, and also increments another register.

We have characterized a class of domains called *extended-LL* domains where the outcomes of any action application resulting in multiple outcomes depend on whether or not a role-count was greater than or equal to one. Examples of such domains are linked lists, blocks-world scenarios, and domains with only unary predicates as in Fig. 4. Formal results relating extended-LL domain plans to abacus programs are presented in [7].

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. We also demonstrate our approach on plans with complex loops created by on-deterministic sensing actions.

4 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.

Transport In the fully observable version of 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. In the partially observable formulation, 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.

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 roles. Fig. 5 shows the main loop in a solution plan found by merging plans for handling various non-deterministic outcomes with the base plan which solved the fully observable formulation [14]. The base plan first moves a server from D1 to L using T1; T2 then picks up a monitor at D2, moves to L, picks up the server left by T1 and transports both to D3. In other words, in the fully-observable version of this problem, outcomes labeled "heavy", and "server lost" in Fig. 5 are never taken, and the solution plan is a simple loop without the segments created by these branches. The computed preconditions for this version of the

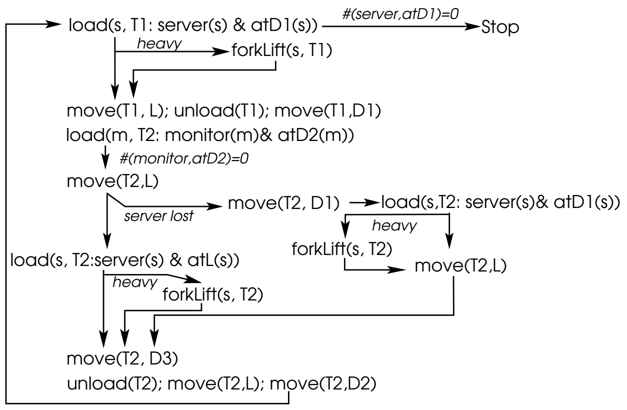
load(s, T1: server(s) & atD1(s))
#(server,atD1)=0
Stop
heavy
forkLift(s, T1)
move(T1, L); unload(T1); move(T1,D1)
load(m, T2: monitor(m)& atD2(m))
#(monitor,atD2)=0
move(T2,L)
server lost
move(T2, D1)
load(s,T2: server(s)& atD1(s))
heavy
forkLift(s, T2)
move(T2,L)
load(s, T2:server(s) & atL(s))
heavy
forkLift(s, T2)
move(T2, D3)
unload(T2); move(T2,L); move(T2,D2)

Fig. 5. Solution plan for the conditional version of transport

plan (i.e., assuming that those outcomes can never occur) correctly state that we need to have equal numbers of monitors and servers to reach the goal.

In the partially observable version, 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. The computed preconditions show that every possible loop decrements the role-counts of servers and monitors at D_1 and D_2 respectively; 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.

Accumulator The accumulator problem [1] 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.

Further Test Problems and Discussion We tested our algorithms with many other plans with loops, for the planning problems: Hall-A, Prize-A(7 & 5), Corner-A [3], Diagonal [7], Recycling, and Striped Tower [12] in addition to those discussed above. The times taken for computing preconditions were uniformly less than 0.06 seconds (see [7] for more details); the runs were conducted on a 2.5GHz AMD dual core system. At least one of the quantities 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 all the plans, termination of loops was proved by non-negativity constraints such as those above.

5 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 [15] 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. Among more recent work, KPLANNER [1] 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 [2] 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 [16] can be viewed as a compilation of plan segments and their applicability conditions. However, there has been no concerted effort for finding preconditions of plans with loops. Static analysis of programs deals with similar problems in proving program correctness. 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 [17] attempt to prove termination, but do not provide precise preconditions or the number of iterations required for termination. The focus of such approaches lies in searching for an instance of the set of possible initial states on which a given program may fail. On the other hand, we want to find termination conditions and also preconditions when possible, as opposed to just verifying the correctness of the given plan.

6 Conclusions and Future Work

We presented a formal approach for finding preconditions of plans with a restricted form of loops. We also presented a characterization of the aspects of complex loops, which make it difficult to find their preconditions. While the presented approach is the first to address this problem, it is also very efficient and scalable. In addition to finding preconditions of computed plans, it can also be used as a component in the synthesis of plans with safe loops.

A greater understanding of the impact of a plan's structural complexity on the hardness of evaluating its preconditions is a natural question for future research. The scope of the presented approach could also be extended by combining it with approaches for symbolic computation of preconditions of action sequences. The connections with hybrid systems discussed in Section 2.5 also present promising directions 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.

References

1. Levesque, H.J.: Planning with loops. In: Proc. of IJCAI. (2005) 509–515
2. Winner, E., Veloso, M.: LoopDISTILL: Learning domain-specific planners from example plans. In: Workshop on AI Planning and Learning, ICAPS. (2007)
3. Bonet, B., Palacios, H., Geffner, H.: Automatic derivation of memoryless policies and finite-state controllers using classical planners. In: Proc. of ICAPS. (2009)
4. Lambek, J.: How to program an infinite abacus. Canadian Mathematical Bulletin **4**(3) (1961)
5. Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1967)
6. Eggan, L.C.: Transition graphs and the star-height of regular events. Michigan Mathematical Journal **10** (1963) 385–397
7. Srivastava, S., Immerman, N., Zilberstein, S.: Computing applicability conditions for plans with loops. In: Proc. of ICAPS. (2010)
8. Hopcroft, J., Pansiot, J.J.: On the reachability problem for 5-dimensional vector addition systems. Theoretical Computer Science **8**(2) (1979) 135 – 159
9. Kosaraju, S.R.: Decidability of reachability in vector addition systems (preliminary version). In: STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing. (1982) 267–281
10. Alur, R., Henzinger, T.A., Ho, P.H.: Automatic symbolic verification of embedded systems. IEEE Transactions on Software Engineering **22** (1996) 181–201
11. Fox, M., Long, D.: PDDL2.1: An extension to pddl for expressing temporal planning domains. Journal of Artificial Intelligence Research **20** (2003) 2003
12. Srivastava, S., Immerman, N., Zilberstein, S.: Learning generalized plans using abstract counting. In: Proc. of AAAI. (2008) 991–997
13. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems **24**(3) (2002) 217–298
14. Srivastava, S., Immerman, N., Zilberstein, S.: Merging example plans into generalized plans for non-deterministic environments. In: Proc. of AAMAS. (2010)
15. Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Weak, strong, and strong cyclic planning via symbolic model checking. Artif. Intell. **147**(1-2) (2003) 35–84
16. Fikes, R., Hart, P., Nilsson, N.: Learning and executing generalized robot plans. Technical report, AI Center, SRI International (1972)
17. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. (2006)