Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)

# Computing Applicability Conditions for Plans with Loops

**Siddharth Srivastava** and **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. 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 (Levesque 2005; Winner & Veloso 2007; Bonet, Palacios, & Geffner 2009). 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. 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. 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 (Lambek 1961) 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* $\ell : \mathcal{S} \setminus \{s_h\} \mapsto$ 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. 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:

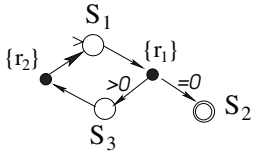


Figure 1: A simple abacus machine for the program: while $(r_1 > 0)$ { $r_1 - -; r_2 + +$}

**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 defined by a given set of register values (Theorem 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, $\bar{R}^0 = \langle R_1^0, R_2^0, \ldots, R_m^0 \rangle$ 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 $\Delta^i$ 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.* (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 using this expression to state the required inequality for following the required 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, $C(\bar{R}^0, \bar{F}, l)$, that are satisfied by initial and final register tuples, $\bar{R}^0, \bar{F}$,*

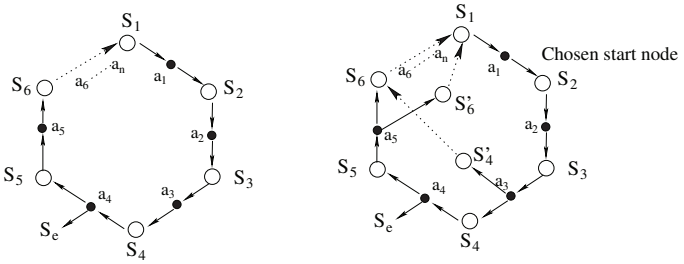


Figure 2: A simple loop with (right) and without (left) shortcuts

*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* $k \in \{0, \ldots, l-1\}$. 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 $\bar{R}^l = \bar{R}^0 + l \times \Delta^{1..n}$, 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 $\Pi_A$ 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 $\Pi_A$ 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. □

## 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 treat a powerful class of nested loops (encountered in partially observable planning) as a set of independent simple loops.

**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, but not after a* designated *start node. The start node must precede all of the loop's shortcuts (e.g., node $S_2$ in Fig.2).*

Simple loops with shortcuts form a very general class of complex loops. 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.

## 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 $\Pi$ is a simple loop with $m$ monotone shortcuts and a chosen start node $S_{start}$. We consider the case of $l$ complete iterations of $\Pi$ 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 $\Delta^{loop_i}$ 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 $k_0, k_1, \ldots k_m$ iterations of loops $0, \ldots, m$, 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. This leads to the following inequalities:

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

$$\forall R_j \in R^+ \big\{ R_j^0 + \delta_{\widehat{\jmath}} \geq 0 \big\}$$

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. 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 $0_j, \ldots, m_j$ be the ordering of loops in decreasing order of negative change values caused by an initial segment of the loop starting at $S_{start}$. We use $k_{i<x} = 0$ as an abbreviation for $\forall i < x : k_i = 0$. We can then write a disjunction of constraints corresponding to the first loop in $loop_{0_j}, \ldots loop_{m_j}$ which has non-zero iterations:

$$\forall R_j \in R^- \bigvee_{x=0_j,\ldots,m_j} \big\{ k_{i<x} = 0; k_x \neq 0; \\ R_j^0 + \sum_{i \neq x} k_i \Delta^{loop_i} + \delta_x - \Delta^{loop_x} \geq 0 \big\} \quad (3)$$

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

Constraints 3 & 4 are derived from the unnumbered constraints above by replacing $\widehat{\jmath}$ with $x$, which iterates over loops in the order $0_j..m_j$, specific to register $R_j$; $\delta_x$ represents the greatest negative change in loop $x$ for role $j$.

**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 and not over-constraining, we first define 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.*

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.

**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 $\Pi$ 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 $\Pi$ 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. □

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

**Theorem 2.** *Let $\Pi$ 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 $\Pi$ 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** In the result and the conditions constructed above, the $k_i$ variables, which count the number of times a non-deterministic action effect occurs, 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 the next section, 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 for the conditions above. In addition, if the $k_i$ are used as parameters, the sufficient conditions above capture their tolerable values under which a desired register value may be achieved.

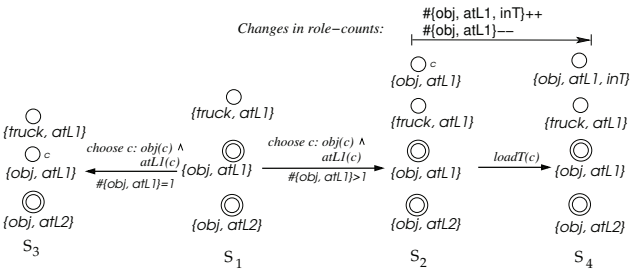


Figure 3: A sequence of actions in a unary representation of transport domain. Predicate *object* is abbreviated as *obj*.

## 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 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 (Srivastava, Immerman, & Zilberstein 2008).

We illustrate the relevant concepts of ARANDA with an example. ARANDA uses *canonical abstraction* (Sagiv, Reps, & Wilhelm 2002) 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. 3 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. The abstract state $S_1$ 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. 3 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. 3, 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. 3. Assuming this fact about extended-LL domains, we can state the following lemmas. The term *linear* in these results refers to plans or programs consisting of a linear sequence of actions.

**Lemma 1.** *Let $S_1 \xrightarrow{a_1} S_2$ be an action operation in an extended-LL domain, where $S_2$ is one of the possible results of $a_1$. This operation can be translated into a linear abacus program $\Pi_a$ whose start node is labeled $S_1$ and terminal node is labeled $S_2$.*

*Proof. (Sketch)* We create registers for every role-count that is changed. Increments and decrements are conducted via a sequence of abacus operators, with increments first. If $a_1$ increments a role-count or makes no change in role-counts, the translation is straightforward (using an extra register to be incremented in the latter case). In order to translate a decrementing operation on the role-count of a role $R$, we make two decrements on the register $R$, a comparison with 0, and finally an increment operation to reverse the extra decrement. Note that before the first action application the role $R$ has at least one element (otherwise $R$ would not have been present in $S_1$). □

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 simple loops of abacus actions (Prop. 2) can thus be used for plans with simple loops in extended-LL domains. A similar translation can be used to translate abacus actions into sequences of extended-LL domain actions, leading to the following result:

**Lemma 2.** *Linear segments of abacus programs can be simulated by linear segments of programs in extended-LL domains and vice versa.*

As a result, the class of abacus programs is equivalent to the class of plans with extended-LL domain actions:

**Corollary 1.** *Plans with extended-LL domain actions can simulate abacus programs without increasing the loop complexity of the program, and vice versa.*

Consequently, we have:

**Theorem 3.** *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.

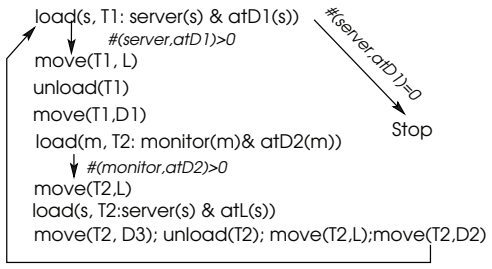


Figure 4: Solution plan for the transport problem

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 non-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. Table 1 shows timing results for 10 different plans.

**Plan Representation** Figs. 4, 5 and 6 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

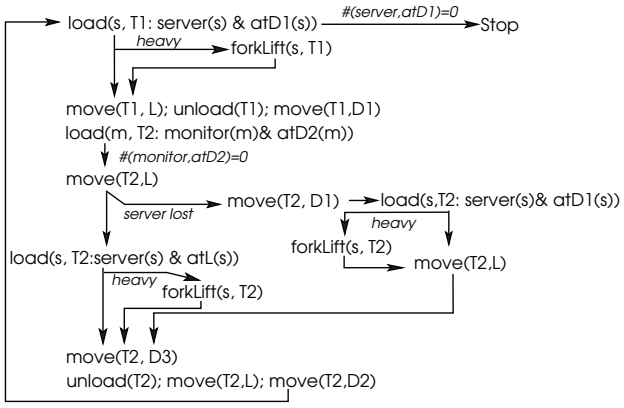


Figure 5: Solution plan for the conditional version of transport

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 roles. The plan shown in Fig. 4 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. 5 shows a solution plan found by merging togther plans which encountered and dealt with different non-deterministic action outcomes (Srivastava, Immerman, & Zilberstein 2010). 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 $s$ 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.

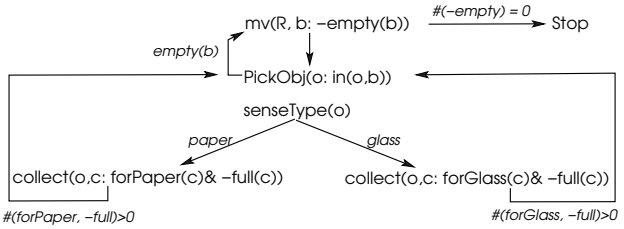


Figure 6: Solution plan for the recycling problem

| 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

**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. 6), 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, ¬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.

**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, Palacios, & Geffner 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, Palacios, & Geffner 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, Immerman, & Zilberstein 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.

## 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 (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 & 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, Hart, & Nilsson 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, Podelski, & Rybalchenko 2006) specifically attempt to prove termination of loops, but do not provide precise preconditions or the number of iterations required for termination.

## 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.

## Acknowledgments

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

## References

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. of ICAPS*, 34–41.

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

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

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

Lambek, J. 1961. How to program an infinite abacus. *Canadian Mathematical Bulletin* 4(3):295–302.

Levesque, H. J. 2005. Planning with loops. In *Proc. of IJCAI*, 509–515.

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

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

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *Proc. of AAAI*, 991–997.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2010. Merging example plans into generalized plans for non-deterministic environments. In *Proc. of AAMAS* (to appear).

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