Artificial Intelligence 191–192 (2012) 1–19

Contents lists available at SciVerse ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Applicability conditions for plans with loops: Computability results and algorithms

Siddharth Srivastava *, Neil Immerman, Shlomo Zilberstein

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

ARTICLE INFO

Article history:
Received 15 November 2010
Received in revised form 18 July 2012
Accepted 20 July 2012
Available online 25 July 2012

Keywords:
Automated planning
Plans with loops
Plan verification
Reachability in abacus programs
Generalized planning

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

1. Introduction

The problem of planning in AI is to compute a plan, or a procedure which can be executed by an agent to achieve a certain goal. This paper presents methods which can be used for the computation of compact plans that resemble computer programs with branches and loops.

In the classical formulation of AI planning, the agent's state is assumed to be completely observable, and effects of actions are assumed to be determined entirely by this state. Classical plans consist of linear sequences of actions which lead to a goal state from a particular initial state. Even in this restricted, deterministic formulation, the planning problem is PSPACE-complete [2] when the input is specified in the STRIPS framework [8]. More general formulations which allow the agent to possess only partial information about its current state, and its actions to be non-deterministic make the problem significantly harder [18]. Consequently, numerous approaches have been proposed for reusing sequences of actions computed for related problems [7,10] and for computing generalized plans which can be used to solve large classes of planning problems [19,14,26,24].

Approaches for generalized planning build extensively upon the power of including loops of actions for representing cyclic flows of control in plans. Not only are such constructs necessary when the input problem instances can be unbounded in size, but they also allow significant reductions in plan sizes for larger problems—particularly when contingent solutions are required in order to deal with partial observability [1,23]. Plans with loops therefore present two very appealing advantages: they can be more compact, and thus easier to synthesize, and they often solve many problem instances, offering greater generality.

* Corresponding author.
E-mail addresses: siddharth@cs.umass.edu (S. Srivastava), immerman@cs.umass.edu (N. Immerman), shlomo@cs.umass.edu (S. Zilberstein).

0004-3702/$ – see front matter
http://dx.doi.org/10.1016/j.artint.2012.07.005

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

In this paper, we present methods for computing the conditions under which a plan with a particular class of loops will terminate at a desired state. Our approach elaborates and builds upon the ideas presented in [22]. We further develop these ideas to identify more clearly the factors that make the problem of determining termination of plans with loops difficult. We also present new results for determining termination for a broader class of plans with loops and illustrate how our methods can be applied.

We first formulate the notion of plans with loops using the concept of *generalized planning problems* introduced in prior work [21,24]. Solutions to such problems are expressed as generalized plans. Generalized plans are rich control structures that include loops and parameterized or "lifted" actions whose arguments must be instantiated during execution. These notions are described in Section 2.

In spite of their expressiveness, a broad class of generalized plans can be easily translated into *abacus programs*—formal models of computation that use primitive actions, but are as powerful as Turing machines. Abacus programs have finite sets of non-negative registers, and actions that may increment or conditionally decrement these registers (Section 2.4). Abacus programs have been shown to have a close relationship with *numerical* planning problems. Helmert [11] showed that abacus programs can be reduced to a class of planning domains over numerical variables where the goal conditions do not use numerical variables, but action preconditions include comparisons of these variables with zero and action effects increment or decrement these variables. This leads to a negative result that the plan existence problem is undecidable for such planning domains due to the undecidability of the halting problem for abacus programs. In this work however, we present some positive results capturing classes of abacus programs for which the halting problem is decidable.

Our approach for computing applicability conditions for plans with loops is to first develop methods for computing the conditions under which a given abacus program will reach a desired state. This is referred to as the *reachability* problem of abacus programs. Undecidability of the halting problem in abacus programs implies that the reachability problem for abacus programs is also undecidable *in general*. However, we show that certain classes of abacus programs, categorized in terms of the graphical structure used to represent their control flow, do have solvable reachability problems. We develop methods for addressing the reachability problem of abacus programs in these classes.

These methods can be used to compute applicability conditions for a broad class of generalized plans by translating them into abacus programs. Furthermore, the fact that this translation preserves the structure of the control flow makes these methods applicable also in synthesis of "tractable" generalized plans: during synthesis, we can choose to permit only those control structures in generalized plans that would allow the computation of reachability conditions upon translation to abacus programs. Prior work describes one possible instantiation of this process in greater detail [25]. The fundamental nature of abacus programs also makes our methods more generally applicable to plans with loops that may not be expressed as generalized plans in our representation, but which have suitable translations into abacus programs.

The following section develops the formal framework for the rest of the paper and describes the connection between generalized plans and abacus programs. We develop methods for solving the reachability problem for abacus programs whose control flow only uses simple loops in Section 3. We then introduce a class of nested loops in Section 4 and develop methods for addressing the reachability problem for deterministic and non-deterministic abacus programs with this class of nested loops in Section 5. Finally, we conclude with a demonstration of the scope and efficiency of these methods.

2. Formal foundations

In this work we consider loops of actions whose every iteration, during any execution of the plan, will make measurable progress towards a goal. We call such necessarily terminating loops, *progressive*. For example, in the blocks-world, a loop of actions which in every iteration unstacks a block that is clear but not on the table, makes incremental progress towards the goal of having all blocks on the table. In contrast, a loop could also be used with actions that need to be repeated until they succeed. For example, in order to pick up a block using a slippery gripper, we need a loop that executes the pickup action until it succeeds. Plans with such loops are considered in strong cyclic planning [3] but are not the focus of this paper. Our motivation for considering only progressive loops is to facilitate the computation of plans with strong guarantees of termination and correctness in situations where the number of objects to be manipulated is unknown.

To clarify these notions, we begin the formal description of our approach with a brief summary of a recently proposed framework for generalized planning in which progressive loops turn out to be very useful. This is followed by a description of our representation for generalized plans (Section 2.2). The latter half of this section presents the formal definition of abacus programs (Section 2.4) and some conditions under which we can view generalized plans as abacus programs (Section 2.3).

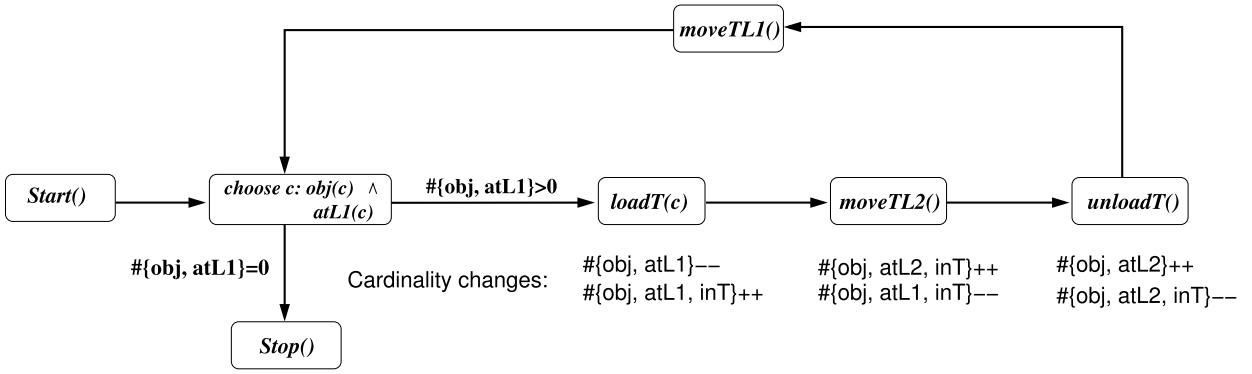

Fig. 1. A generalized plan for transporting objects from L1 to L2.

2.1. Generalized planning problems

Intuitively, a generalized planning problem consists of a domain schema, a set of initial states and a goal condition. A domain schema includes a predicate vocabulary (the set of predicate symbols that can be used in formulas; constants are represented as special unary predicates), a set of action operators and integrity constraints. We use first-order logic to represent domain schemas and generalized planning problems. This allows us to represent planning domains without referring to the specific objects that may occur in a particular generalized planning problem. Further, a generalized planning problem may include uncertainty about object quantities and properties. We refer the reader to prior work for details [21,24] and present the essentials below.

Definition 1 *(Domain schema).* A domain schema is a tuple $\mathcal{D} = \langle \mathcal{V}, \mathcal{A}, \mathcal{K} \rangle$ where $\mathcal{V}$ is a vocabulary, $\mathcal{A}$ is a set of actions expressed in first-order logic with transitive closure (FO(TC)), and $\mathcal{K}$ is an integrity constraint expressed in FO(TC).

Action representation. For each predicate p that action a_1 affects, the action operator for a_1 includes an expression of the following form, where p' denotes the predicate after action application:

$$p'(\bar{x}) \equiv \left[\neg p(\bar{x}) \wedge \Delta^{+}_{p,a}(\bar{x})\right] \vee \left[p(\bar{x}) \wedge \neg \Delta^{-}_{p,a}(\bar{x})\right]$$

Here $\Delta^{+}_{p,a}$ denotes the conditions under which predicate p is changed to true on application of action a, and $\Delta^{-}_{p,a}$ denotes the conditions under which it is changed to false. Intuitively, Eq. (1) states that p becomes true for a tuple iff either (a) it was false and action a changes it to true, or, (b) it was already true, and is not removed by action a. This representation is similar to frame axioms in situation calculus [15]. To compute the effect of an action on a given state, for each affected predicate p we evaluate the truth of the RHS of Eq. (1) on the given state.

We define a generalized planning problem as follows:

Definition 2 *(Generalized planning problem).* A generalized planning problem is a tuple $\langle \alpha, \mathcal{D}, \gamma \rangle$ where α is an FO(TC) formula describing the possible initial states, $\mathcal{D}$ is the domain schema, and γ is an FO(TC) formula specifying the goal states.

2.2. Generalized plans

We represent generalized plans using graphs. We provide a brief illustration of the main features of this representation here and refer the interested reader to prior work for further details [24].

Definition 3 *(Graph-based generalized plan).* A graph-based generalized plan $\Pi = \langle V, E, \ell, s, T \rangle$ is defined as a tuple where V and E are respectively, the vertices and edges of a finite connected, directed graph; ℓ is a function mapping nodes to actions and edges to conditions represented as linear inequalities; s is the start node and T a set of terminal nodes.

The edge conditions in graph-based generalized plans are represented as linear inequalities over the number of objects that satisfy certain properties (unary predicates). The interested reader is referred to [24] for details. Fig. 1 shows a simple generalized plan for a transport problem, where a single truck incrementally loads an object from location L1, drives to L2 and unloads the object. The predicate vocabulary in this example consists of the unary predicates $obj(x)$ denoting that x is an object to be transported; $atLi(x)$ denoting that x is at Li, where $i \in 1, 2$; and $inT(x)$, denoting that x is in the truck. The start and terminal nodes for this plan are labeled with dummy Start() and Stop() actions. Most edges have the default edge condition, "True". The two non-True edge conditions use the number of objects that satisfy the predicates obj and $atL1$ (denoted as $\#\{obj, atL1\}$). These two edge conditions depend on whether or not the cardinality of the set of objects at L1 is equal to 0.

The execution of a generalized plan begins at the start node and continues along edges whose conditions are satisfied by the world state resulting from the last action's application. Fig. 1 also lists the changes in cardinalities of certain predicate combinations. These changes are significant to the translation from plans to abacus programs and we will revisit them in the next section.

This example illustrates how a generalized plan may use choice actions to select arguments for subsequent actions. Choice actions select an object which satisfies a given formula in first-order logic, and assign it to a constant used in action update formulas. Intuitively, if multiple objects satisfy a formula used in a choice action, the generalized plan is considered to *solve* a problem iff all executions of the plan with all choices of the qualifying objects will solve the problem.

2.3. Cardinality changes in generalized plans

The generalized plan in Fig. 1 is annotated with the changes in cardinalities of various properties. The class of possible properties whose cardinalities are kept track of can be specific to a particular approach for generating generalized plans. We consider the special case where the space of possible properties is the powerset of all unary predicates in the domain. More precisely, we define the *role* of an element in a state to be the set of unary predicates that it satisfies (e.g., $\{obj, atL1\}$). The *role-count* of a role (e.g., $\#\{obj, atL1\}$) in a state denotes the cardinality of that role, or the number of elements that satisfy that role. Thus, in Fig. 1, the cardinality changes indicate that the $loadT(c)$ action decrements the number of objects at L1 by one and increments the number of objects at L1 and in the truck by one.

In the following development we will utilize two crucial aspects of cardinality changes that are demonstrated in Fig. 1:

1. Action branches in the plan (nodes with out-degree greater than 1) are distinguished by inequalities between a constant (zero) and the cardinalities of certain properties.
2. The changes due to actions on these cardinalities are deterministic. Every possible execution of a particular action node in the plan leads to the same change in the cardinality.

In fact, these aspects are fundamental to computation—as we will see in the next section, some form of such cardinality changes can be used to express *any* plan with loops.

In prior work we showed how generalized plans could be computed together with such cardinality changes in a wide class of domains [24]. This class includes all PDDL-like domains that use only unary predicates in their vocabulary and a particular class of domains with binary predicates, defined as extended-LL domains. In that work, action branches depend on comparisons between role-counts and the constant 1, while we use the constant 0 in this paper. The two representations are equivalent however and plans can be easily translated from one to the other [20]. Another direction of study, which we defer to future work, would be to identify the changes caused by each action node on a selected group of cardinalities, given an arbitrary generalized plan.

The main insight of this paper is that we can effectively determine the applicability of a generalized plan by looking only at the effect of each action on the cardinalities that determine action branches. Thus, we can reduce a generalized plan into a simpler structure whose actions increment or decrement non-negative integer valued variables, and whose action branches depend on the values of these variables. Such structures are known as *abacus programs* and are described formally in the next section.

Viewing generalized plans as abacus programs allows us to study more easily the conditions under which a particular sequence of action branches will be taken during an execution. This in turn allows us to compute the conditions under which the execution of a generalized plan will lead to a desired state in the plan. In most applications this state will be one that can only be reached when the world state satisfies a goal condition (recall that a generalized plan's edges are labeled with conditions on world states). In such configurations our methods will compute the conditions under which a generalized plan will terminate in a finite number of steps and achieve the goal. A domain can also be designed so that the non-occurrence of an undesirable property is included as a part of the goal formula—in which case, goal reachability will also ensure that unsafe situations do not occur.

2.4. Abacus programs

We now introduce the formal framework of abacus programs [13]. Abacus programs are finite automata whose states are labeled with actions that increment or decrement a fixed set of registers.

Definition 4 *(Abacus programs).* An abacus program $\langle \mathcal{R}, \mathcal{S}, s_0, s_h, \ell \rangle$ consists of a finite set of non-negative, integer-valued 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 $s_2 \in \mathcal{S}$.

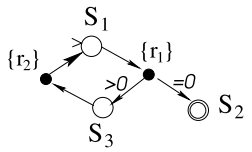

Fig. 2. A simple abacus machine for the program: `while` $(r_1 > 0)$ { $r_1 - -$; $r_2 + +$}.

We represent abacus programs as bipartite graphs with edges from nodes representing control states to nodes representing actions and vice-versa. In the rest of this paper, we will use the term "state" in connection with abacus programs to refer to a node that represents a control state and the term "action" to refer to a node that represents an action. "Node" will be used as a more general term, only in situations where the type of the label of the node is irrelevant to the property being discussed. States have at most one outgoing edge and actions have at most two outgoing edges; the two edges out of a decrement action are labeled $= 0$ and > 0 respectively (see Fig. 2).

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 state is reached. An abacus program *terminates* iff its execution reaches the halt state. The set of final register values in this case is called the *output* of the abacus program.

Abacus programs are equivalent to Minsky Machines [16], which are as powerful as Turing machines and thus have an undecidable halting problem:

Fact 1. The problem of determining whether an abacus program will reach the halt state starting with a given set of initial register values is undecidable.

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

As discussed in the previous section, our approach for determining the utility and applicability conditions of plans with loops is to view them as abacus programs. However, the abacus program framework is restrictive from this point of view: it does not include non-deterministic actions. In planning on the other hand, non-deterministic sensing actions are common. We need a way to effectively translate them into the abacus framework, without changing the loop structure. For this purpose, we extend the abacus program framework by adding the following non-deterministic form of action to Definition 4:

Definition 5 *(Non-deterministic abacus programs).* Non-deterministic abacus programs are abacus programs whose set of actions, *Act*, includes, in addition to the *Inc* and *Dec* actions, non-deterministic actions of the form:

- $NSet(s_1, s_2)$: non-deterministically go to $s_1 \in \mathcal{S}$ **or** go to $s_2 \in \mathcal{S}$

where $\mathcal{S}$ is the set of states of the abacus program.

A non-deterministic action thus has two outgoing edges in the graph representation. Either of these branches may be taken during execution. Although the original formulation of abacus programs is sufficient to capture any computation, these actions will allow us to conveniently translate plans with loops for non-deterministic domains into abacus programs.

3. Applicability conditions for deterministic simple-loop abacus programs

We now show that for any simple-loop abacus program, we can efficiently characterize the exact set of register values that lead not just to termination, but to any desired "goal" state defined by a given set of register values (Theorem 1). We only consider deterministic actions in this section; the case for simple loops with non-deterministic actions is analogous and can also be handled as a special case of the methods presented in Section 5.2 for a more general class of loops. Recall that a non-trivial strongly connected component is one which has more than one node.

We define simple-loop abacus programs as follows:

Definition 6 *(Simple-loop abacus programs).* A simple loop in a graph is a strongly connected component consisting of exactly one cycle. A *simple-loop* abacus program is one all of whose non-trivial strongly connected components are simple loops.

Let $S_1, a_1, \ldots, S_n, a_n, S_1$ be a simple loop (see Fig. 3). We denote register values at states 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 state S_1. Let $a(i)$ denote the index of the register potentially 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* state.

We use subscripts on vectors to project the corresponding registers, so that the initial count of action a_i's register can be represented as $\bar{R}^0_{a(i)}$. Let Δ^i denote the vector of changes in register values $R_1, \ldots, R_m$ for action a_i corresponding to its

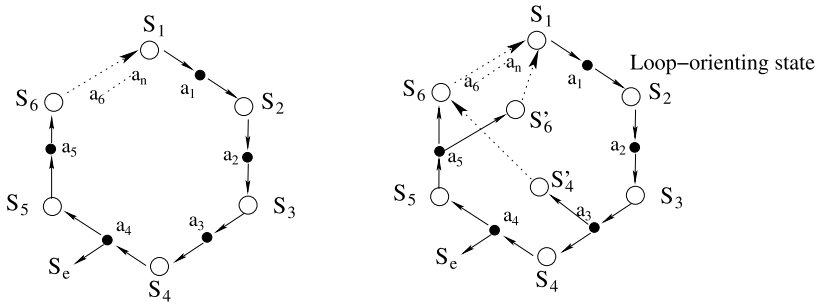

Fig. 3. A simple loop with (right) and without (left) shortcuts.

branch along the loop. For any action, this change vector has 0's in all dimensions except possibly for the register index that the action affects, where the change can be $+1, -1$, or 0 (corresponding to an "$= 0$" branch of a decrementing action). Let a *linear segment* of an abacus program be a distinct sequence of states and actions, $S_1, a_1, S_2, a_2, \ldots, a_{n-1}, S_n$, such that S_i has an edge to a_i and a_i has an edge to S_{i+1} in the program. 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 state combination:

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

Proof. We know $\bar{F} = \bar{R}^0 + \Delta^{1..n}$, *if* the linear segment is executed until S_n. However, we need to determine the conditions under which flow of control will not take a branch leading out of this linear segment. Since the sequence of actions is known, register values at each state S_i can be represented in terms of $\bar{R}^0$. More precisely, the register vector before action a_i (at S_{i-1}) is $\bar{R} + \Delta^{1..i-1}$. The condition for taking the > 0 branch of a_i can therefore be expressed as $(\bar{R} + \Delta^{1..i-1})_{a_i} > 0$. A conjunction of such expressions for each decrementing action in the given linear segment constitutes the necessary and sufficient conditions (by induction on the length of the linear segment). This conjunction can be computed in time linear in the length of the input segment. □

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

Proof. Consider the action a_4 in the left loop in Fig. 3. 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 $(\bar{R}^0 + \Delta^{1..3})_{a(4)} > 0$. For one full execution of the loop starting with $\bar{R}^0$ we require, for all $i \in \{1, \ldots, n\}$:

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

where $\circ$ is one of $\{>, =\}$ corresponding to 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 $\mathsf{LoopIneq}(\bar{R}^0)$. Formally, for any vector of register values $\bar{R}$ and a given simple loop *sl*, we define LoopIneq as follows:

$$\mathsf{LoopIneq}_{sl}(\bar{R}) = \bigwedge_{i=1}^{n} \left\{ \left(\bar{R} + \Delta^{1..i-1}\right)_{a(i)} \circ 0 \right\}$$

where in the ith inequality, $\circ$ is the inequality on the branch following action $a(i)$ that is in the loop. We omit the subscript *sl* where it is clear from the context. Let $\bar{R}^\ell = \bar{R}^0 + \ell \times \Delta^{1..n}$, the register vector after ℓ complete iterations. Thus, for executing the loop completely ℓ times, the required condition is $\mathsf{LoopIneq}(\bar{R}^0) \wedge \mathsf{LoopIneq}(\bar{R}^{\ell-1})$. This conjunction ensures that the conditions for execution of intermediate loop iterations hold, because the changes in register values due to actions are constant, and the expression for $\bar{R}^{\ell-1}$ is linear in them. 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:

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

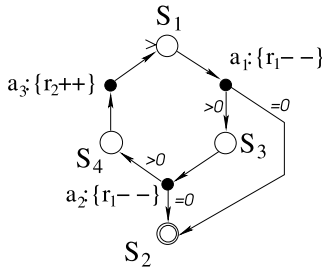

Fig. 4. An abacus program for integer division by 2.

The inequality for each action is of 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. □

An exit during the first iteration amounts to a linear segment of actions and is handled by Proposition 1. Instead of non-negative integers, each component of $\bar{F}$ may be an algebraic expression representing the register values which make a subsequent state in the abacus program reachable. These expressions may be derived from reachability computations for subsequent segments of the abacus program. More precisely, the precondition for reaching the goal by executing a segment Π_j of a simple loop abacus program will be expressed in terms of the register vector at the start of Π_j (say $\bar{R}^j$). Representing the precondition for reaching $\bar{F}$ by executing an abacus program Π starting with the register vector $\bar{R}^0$ as $pre_\Pi(\bar{R}, \bar{F})$, the precondition for reaching the goal by executing a segment Π_2 followed by Π_1 is computed as $pre_{\Pi_2}(\bar{R}^2, \bar{R}^1) \wedge pre_{\Pi_1}(\bar{R}^1, \bar{F})$. Here the final register vector for pre_{Π_2} is the vector of variables, $\bar{R}^1$. This process is similar to *regression* [17] but it applies to plans with loops of actions rather than acyclic plans. The following example illustrates these points.

Example 1. The abacus program shown in Fig. 4 can be used to divide the value of a register by 2. Suppose the initial register vector is $\bar{R}^0 = \langle r_1^0, r_2^0 \rangle$. The total change vector due to one iteration of the loop is $\Delta^{1..3} = \langle -2, +1 \rangle$. $\mathsf{LoopIneq}(\bar{R}^0)$ for this loop is $r_1^0 > 0 \wedge r_1^0 - 1 > 0$; $\mathsf{LoopIneq}(\bar{R}^{\ell-1}) \equiv r_1^0 + (\ell-1)\cdot(-2) > 0 \wedge r_1^0 + (\ell-1)\cdot(-2) - 1 > 0$.

To obtain conditions for reaching S_2 via the exit from action a_1 we include the condition that the value of r_1 must be zero before the last application of a_1: $r_1^0 + (\ell-1)\cdot(-2) = 0$. In general, this condition can be computed by treating the last, partial iteration of the loop required to reach an exit node (action a_1 in this case), as a linear segment in an abacus program.

Therefore, reachability conditions for S_2 via a_1 with at least one iteration of the simple loop are: $r_1^0 > 0 \wedge r_1^0 - 1 > 0 \wedge r_1^0 = 2(\ell-1)$, where ℓ represents the number of loop iterations. The final register vector at S_2 will be $\bar{F} = \langle 0, r_2^0 + \ell - 1 \rangle$ $(= \langle 0, r_2^0 + r_1^0/2 \rangle)$. The conditions for reaching S_2 via a_1 during the first iteration are: $r_1^0 = 0$; $\bar{F} = \langle 0, r_2^0 \rangle$. Therefore, the necessary and sufficient conditions for reaching S_2 with register vector $\bar{F}$ via a_1 are: $R_{a_1} \equiv \{\ell > 0 \wedge r_1^0 = 2(\ell-1) \wedge \bar{F} = \langle 0, r_2^0 + \ell - 1 \rangle\} \vee \{r_1^0 = 0 \wedge \bar{F} = \langle 0, r_2^0 \rangle\}$. If we include the condition that the final value of r_2 must be $r_1^0/2$, we get $r_2^0 = 0$. In other words, if $r_2^0 = 0$ and r_1^0 is even, then r_2 will be $r_1/2$ at S_2. Reachability conditions for S_2 via a_2 can be computed similarly and capture the case when r_1^0 is odd. Here, we get $R_{a_2} \equiv \{\ell > 0 \wedge r_1^0 = 2\ell + 1 \wedge \bar{F} = \langle 0, r_2^0 + \ell \rangle\} \vee \{r_1^0 = 1 \wedge \bar{F} = \langle 0, r_2^0 \rangle\}$. The complete reachability conditions for S_2 are $R_{a_1} \vee R_{a_2}$. If another segment of the program led to S_1, variables r_i^0 could be used as the components of the final register vector for precondition computation over that segment.

When used in combination with Proposition 1, the method described above produces the necessary and sufficient conditions for reaching any state and register value in an abacus program:

Theorem 1. *Let Π_A be a simple-loop abacus program. Let S be any state 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-state, but consisting only of linear paths and simple loops. One approach for carrying out this process is to first collapse every simple loop in the graph of the abacus program into a "super node". All edges ending or beginning at a node in a simple loop are changed to end or begin respectively, at the super node that replaced the simple loop. The resulting graph will be acyclic and we can compute all linear paths leading from the start state to the desired state. In each such path, we replace the super nodes with their corresponding simple loops. During this process, we re-attach the edges from neighbors of the super node to the original nodes that they were attached to. By Propositions 1 and 2, necessary and sufficient conditions for each of these segments can be computed. The disjunctive union of these conditions gives the

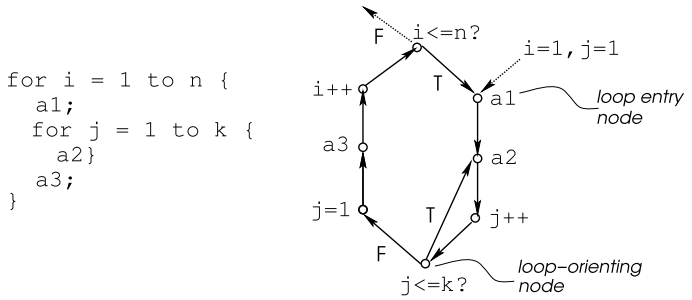

Fig. 5. Many "nested" loops encountered in programming can be viewed as simple loops with shortcuts by distinguishing the loop-orienting node from the loop entry node. In this example, the linear path from a_1 to $j < k$ can be treated as a linear segment, followed by a simple loop with shortcuts.

desired necessary and sufficient condition. In the worst case, the total size of the disjunction could be exponential in the number of nodes in the graph. □

4. Nested loops due to shortcuts

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

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

Definition 7 *(Complex loops).* Let $\mathcal{A}$ be an abacus program. A *complex* loop in $\mathcal{A}$ is a non-trivial strongly connected component that is not a simple loop.

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

Definition 8 *(Simple loop with shortcuts).* Let $\mathcal{A}$ be an abacus program. A simple loop with shortcuts in $\mathcal{A}$ is a strongly connected component $\mathcal{C}$ which includes a node n_0, designated the *loop-orienting node*, such that removing n_0 makes $\mathcal{C}$ acyclic.

We say that an abacus program has *only simple loops with shortcuts* if all its strongly connected components are simple loops with shortcuts.

Note that a loop-orienting node may be labeled with either an action or a state. Intuitively, such a simple loop with shortcuts consists of a simple loop with all elements, starting at the loop-orienting node, in increasing linear order. For any pair of nodes along the loop, a preceding b, a shortcut from a to b may be added; different shortcuts may overlap as long as this does not create cycles. (e.g., state S_2 can be designated the loop-orienting node in Fig. 3). The loop-orienting node does not have to be the node through which the flow of control enters a simple loop with shortcuts. Indeed, if we wish to determine the preconditions with respect to a node other than the loop-orienting node as the "entry" node for a given simple loop with shortcuts, we first find the preconditions with respect to the loop-orienting node as the entry node and then propagate these conditions back along the acyclic path(s) connecting the entry node to the loop-orienting node (by Theorem 1 as applied to an acyclic segment of the abacus program).

Simple loops with shortcuts form a very general class of complex loops: graph theoretically, this is exactly the class of strongly connected components with *cycle rank* 1 [6]. Many control flows that are typically understood as "nested" loops in programming can be represented as simple loops with shortcuts by choosing an appropriate loop-orienting node. Fig. 5 shows an example. Further, for abacus programs we show in Section 4.1 that this class of graphs is powerful enough to express any computation.

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

Definition 9 *(Loop decomposition).* Let $\mathcal{A}$ be an abacus program and let $\mathcal{C}$ a strongly connected component of $\mathcal{A}$ in the form of a simple loop with shortcuts, with the loop-orienting node n_0. The loop decomposition of $\mathcal{C}$ is defined as the set of all cycles of $\mathcal{C}$ beginning with n_0.

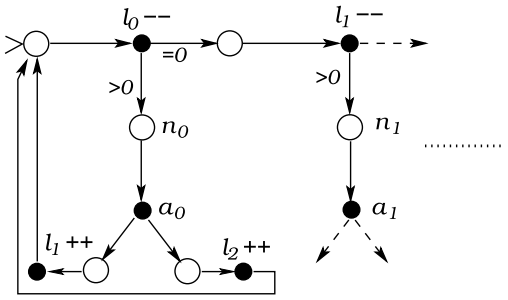

Fig. 6. Construction for translating a general abacus program into one with a simple loop with shortcuts.

In the worst case, the size of this decomposition can be exponential in the number of shortcuts. This construction proves useful because in a simple loop with shortcuts, every cycle must contain the loop-orienting node (this is immediate from Definition 8). Thus, the execution of a simple loop with shortcuts in an abacus program can be viewed as a sequence of complete executions of the simple loops in its decomposition. For instance, we can view the loop with shortcuts in Fig. 3 as consisting of 3 different simple loops. The order of execution of these loops, and whether a given loop will be executed at all, will depend on the results of actions a_3 and a_5.

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

Definition 10 *(Monotone simple loops with shortcuts).* Let the *net change* on a register due to a simple loop in an abacus program be the total change that will be caused on that register in one full execution of the loop. A simple loop with shortcuts in an abacus program is *monotone* iff for every register, the sign (positive or negative) of the net change, if any, on that register is the same for every simple loop in its decomposition.

In the next section we show that removing this restriction can significantly increase the power of abacus programs: any abacus program can be represented as a program consisting of a simple loop with possibly non-monotone shortcuts.

4.1. Relaxing monotonicity

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

Theorem 2. *Let Π_g be an abacus program with R_g, N_g and E_g as the sets of registers, nodes and edges respectively. Then there exists an equivalent abacus program, Π_S with $R_S(\supseteq R_g)$, $N_S(\supseteq N_g)$, and E_S as the sets of registers, nodes and edges respectively, such that*:

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

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

Construction of Π_S. Let $n_0 \to a_0$, $a_0 \to n_1$, $a_0 \to n_2$ be a set of edges corresponding to a single (decrementing) action a_0 in Π_g. We translate this sequence into a sequence beginning with the action decrementing l_0. The > 0 branch from this action represents the case that we were at state n_0. This branch will lead to the node for a_0; the two branches from a_0 lead to actions incrementing l_1 and l_2, corresponding to the branches that lead to n_1 and n_2. The construction is illustrated in Fig. 6. The translation is similar for incrementing actions. To get Π_S, we perform this construction for the edges corresponding to each action in Π_g in this manner and attach the each resulting graph to the $= 0$ branch of the last flag decrementing action, as shown in Fig. 6. The resulting abacus program Π_S consists of one simple loop with shortcuts.

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

By construction, executing an action on a register vector leads to a node n_i in Π_g iff executing that action on the extended register vector with all flag variables zero (note that the flag-testing action also decrements the only non-zero flag

to zero) leads to l_i, and subsequently, n_i in Π_S. By induction on path lengths, we therefore have the result that a node n_i is reachable from $\bar{R}_{init}$ in Π_g iff it is reachable from $\bar{R}'_{init}$ in Π_S. □

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

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

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

5. Applicability conditions for monotone simple loops with shortcuts

We now consider the problem of computing applicability conditions for monotone simple loops with shortcuts. We first present the more specific case of programs with deterministic actions. In the following section we present methods for computing reachability conditions for abacus programs with non-deterministic actions.

5.1. Deterministic monotone shortcuts

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

Our approach relies on the following observations:

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

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

Taking an initial register valuation as input, Algorithm 1 performs these computations. Let Π_A be an abacus program in the form of a simple loop with monotone shortcuts and only deterministic actions. Algorithm 1 works by identifying the unique loop ℓ whose $\mathsf{Loop1neq}_\ell$ is satisfied by the value $\bar{R}$ (initialized to $\bar{R^0}$) [steps 5–8], calculating the number of iterations which will be executed for that loop until $\mathsf{Loop1neq}_\ell$ gets violated [step 9], updating the register values to reflect the effect of those iterations [step 12] and identifying the next loop to be executed [the while loop, step 4].

The subroutine FindMaxIterations uses the inequalities in $\mathsf{Loop1neq}_\ell$ (see Proposition 2) to construct the vector equation $(\bar{R} + \ell_{\max}\Delta^\ell + \Delta^{1..i-1})_{a(i)} \circ 0$ for every action in loop ℓ. This system of equations consists of an inequality of the following form for every i corresponding to a decrementing action in the loop:

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

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 $\ell_{\max}$. Equality constraints either drop out (if the net change in their register's value due to the loop ℓ is zero and they are satisfied during the first iteration), or set $\ell_{\max} = 1$ (if the net change in their register's value is not zero, but it is satisfied during the first iteration). Equality constraints will be satisfied when FindMaxIterations is called because we know that $\mathsf{Loop1neq}_\ell$ was satisfied. Note that if there is any loop which does not decrease any register's value, it will never terminate. This will be reflected in our computation by an $\ell_{\max}$ value of ∞ [step 11]. Thus, we have:

Algorithm 1: Reachability for deterministic, monotone shortcuts

Input: Deterministic abacus program in the form of a simple loop with monotone shortcuts with loop-orienting node (say the state S_{start}), an initial register configuration $\bar{R}^0$
Output: Sequence of (loop id, #iterations) tuples and final value of $\bar{R}$ at S_{start}.

1 $\bar{R} \leftarrow \bar{R}^0$
2 Iterations $\leftarrow$ empty list
3 LoopList $\leftarrow$ simple loops in the loop decomposition
4 **while** *LoopList* $\neq \emptyset$ **do**
5 **if** *no* $\ell \in$ *LoopList satisfies* $\mathsf{LoopIneq}_\ell(\bar{R})$ **then**
6 Return Iterations
  end
7 $\ell \leftarrow$ id of loop for which $\mathsf{LoopIneq}_\ell(\bar{R})$ holds
8 Remove ℓ from LoopList
9 $\ell_{\max} \leftarrow \mathsf{FindMaxIterations}(\bar{R}, \ell)$
10 **if** $\ell_{\max} = \infty$ **then**
  Return "Non-terminating loop"
  end
11 Iterations.append($(\ell, \ell_{\max})$)
12 $\bar{R} \leftarrow \bar{R} + \ell_{\max}\Delta^\ell$
 end
13 Return Iterations, $\bar{R}$

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

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

Complexity analysis. Let b be the maximum number of branches in a loop in the decomposition of the given simple loop with shortcuts, and L the total number of simple loops in the decomposition. The most expensive operation in 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 $O(L^2 b)$. On the other hand, if such a program is directly applied on a problem instance and the program terminates, then the execution time for the program will be of the order of the largest input register value, which is unbounded.

5.2. Non-deterministic monotone shortcuts

We now consider the problem of computing applicability conditions for abacus programs whose simple loops have monotone shortcuts with non-deterministic actions. We presented methods for extending the approach of creating generalized plans with cardinality changes (summarized in Section 2) to this setting in prior work [23].

We will find that the accuracy of the reachability conditions that we compute is determined by *order independence* (Definition 11), or the extent to which the execution of different loops in the decomposition of a simple loop with shortcuts can be rearranged without significantly affecting the overall outcomes. The methods discussed in this section can also be applied to settings with only deterministic actions—yet, simple loops with shortcuts in most such situations demonstrate order *dependence*. Therefore, reachability conditions obtained in this manner will tend to be subsumed by those computed using methods from Section 5.1.

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

$$k_1 + \cdots + k_m = l \tag{1}$$

Determining final register values. We denote the ith loop in the decomposition of the given simple loop with shortcuts as $loop_i$. The final register values after the $l = \sum_{i=1}^{m} k_i$ complete iterations (provided that these iterations are indeed executed

without exiting the simple loop with shortcuts) can be obtained by adding the changes due to each simple loop, with Δ^{loop_i} denoting the change vector due to $loop_i$:

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

Cumulative branch conditions. For computing sufficient conditions on the achievable register values after $k_1, \ldots, k_m$ complete iterations of the given loops, our approach is to treat each loop as a simple loop and determine the preconditions for executing it. Note that every required condition for a loop's complete iteration stems from a comparison of a register's value with zero. We consider the case where the conditions required for staying in the loop are always > 0 and discuss the situation with equality constraints in the following section ("Accuracy of the Computed Conditions"). Thus, we want to determine the lowest possible value of each register during the $k_1, \ldots, k_m$ iterations of loops $1, \ldots, m$, and constrain that value to be greater than zero.

Let $\mathcal{R}^+$, $\mathcal{R}^-$ be the sets of registers undergoing *net* non-negative and negative changes respectively, by any loop. The sequence of actions in an iteration of a simple loop may first decrease a register and then increase it. Through this process, the net decrease in a register due to one full iteration of a simple loop may be smaller than the greatest decrease that it underwent due to a an initial segment of the loop. We denote the change due to an initial segment (w.r.t. the loop-orienting node) of a simple loop on a register as a *partial change* due to that loop on that register. Let δ_j^i be the greatest partial negative change caused on R_j by $loop_i$. Let $\min(j) = \arg\min_x\{\delta_j^x\colon\ x \in \{1, \ldots, m\}\}$.

For $R_j \in \mathcal{R}^+$, the lowest possible value is $R_j^0 + \delta_j^{\min(j)}$, since the value of R_j can only increase after the first iteration. The required constraint on $R_j \in \mathcal{R}^+$ therefore is $R_j^0 + \delta_j^{\min(j)} \geqslant 0$ (we require "$\geqslant 0$" because the condition "> 0" on an edge refers to the register value before a decrement takes place).

We now compute a lower bound on the least value of R_j that can be achieved with $k_1, \ldots, k_m$ iterations of loops $1, \ldots, m$ respectively.

Lemma 1. *In any execution of $k_1, \ldots, k_m$ iterations of loops $1, \ldots, m$, the value of register R_j can never fall below $R_j^0 + \sum_{i=1}^{m} k_i \Delta_j^{loop_i} + \delta_j^{\hat{\jmath}} - \Delta_j^{loop_{\hat{\jmath}}}$, where $\hat{\jmath} = \arg\min_x\{\delta_j^x - \Delta_j^{loop_x}\colon\ x \in \{1, \ldots, m\}\}$.*

Proof. Suppose the last loop to be executed is $loop_x$. If $\delta_j^x \neq \Delta_j^x$, then the least possible value of R_j during the last execution of $loop_x$ is given by first computing the value of R_j after execution of all iterations of all the required loops, and then subtracting from it the effect of one complete iteration of $loop_x$, and adding δ_j^x, the greatest partial negative change of $loop_x$:

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

To obtain the lowest value of this expression over all possible choices for the last loop, we need to minimize this expression w.r.t. x. In most cases encountered in planning, this can be done effectively by choosing the loop which minimizes δ_j^x and using that loop for x (this method was used by Srivastava et al. [22]). In this paper, we use the more general approach by selecting the last loop, $\hat{\jmath}$, as follows:

$$\hat{\jmath} = \arg\min_x\left\{\delta_j^x - \Delta_j^{loop_x}\colon\ x \in \{1, \ldots, m\}\right\}$$

This minimization requires the same number of comparison operations as the minimization over δ_j^x alone.

Let $R_j^{lb} = R_j^0 + \sum_{i=1}^{m} k_i \Delta_j^{loop_i} + \delta_j^{\hat{\jmath}} - \Delta_j^{loop_{\hat{\jmath}}}$. Our claim is that this expression is a lower bound on the possible values of R_j in any execution of the given loops and their iteration counts. Suppose this is not true. Then, a strictly lower value of R_j must be achieved during an execution of some loop, $loop_q$, which is not the last loop to be executed. This is not possible however, because $R_j \in \mathcal{R}^-$ and every successive loop iteration can only decrease its value. □

Now that we can compute the minimum possible values of all registers, we can state the required constraints as:

$$\forall R_j \in R^-\left\{R_j^0 + \sum_{i=0}^{m} k_i \Delta_j^{loop_i} + \delta_j^{\hat{\jmath}} - \Delta_j^{loop_{\hat{\jmath}}} \geqslant 0\right\} \tag{3*}$$

$$\forall R_j \in R^+\left\{R_j^0 + \delta_j^{\min(j)} \geqslant 0\right\} \tag{4*}$$

Together with Eqs. (1)–(2), these inequalities provide sufficient conditions binding reachable register values with the number of loop iterations and the initial register values. However, the process for deriving them assumed that for every j, $loop_{\hat{\jmath}}$ and $loop_{\min(j)}$ will be executed at least once. We can make these constraints more accurate by using a disjunctive

formulation for selecting the loop causing the greatest negative change among those that are executed at least once. For register R_j, let $0\hat{\jmath}, \ldots, m\hat{\jmath}$ be the ordering of loops in increasing order of the values $\delta_j^x - \Delta_j^{loop_x}$. We will use this ordering for writing the constraints for registers in $\mathcal{R}^-$. Similarly, let $0j, \ldots, mj$ be the ordering of loops in increasing values of δ_j^x, with the intended purpose of writing constraints for registers in $\mathcal{R}^+$. In each of the following constraints, we will use $k_{i<x} = 0$ to denote the constraints $\{k_i = 0\colon\ i < x\}$, where the ordering is the one being used in that constraint. We can now write disjunctions of constraints corresponding to the first loop in these orderings that is executed at least once, as follows:

$$\forall R_j \in R^- \bigvee_{x=0\hat{\jmath},\ldots,m\hat{\jmath}} \left\{ k_{i<x} = 0; k_x \neq 0; R_j^0 + \sum_{x \leqslant i \leqslant m\hat{\jmath}} k_i \Delta_j^{loop_i} + \delta_j^x - \Delta_j^{loop_x} \geqslant 0 \right\} \tag{3}$$

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

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

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

The following example illustrates the computation of conditions (3) and (4).

Example 2. Suppose the decomposition of an abacus program in the form of a simple loop with shortcuts consists of two loops. A single iteration of $loop_1$ first decrements R_1 by 5 (i.e., the "first" five actions starting from the loop-orienting node are decrements) and then increments it by 1. A single iteration of $loop_2$ first decrements R_1 by 3 and then increments it by 2. Effects on register R_2 are as follows. A single iteration of $loop_1$ first decrements R_2 by 2 and then increments it by 3; $loop_2$ first decrements R_2 by 1 and then increments it by 2.

Conditions (1) and (2) are easily computed. We need to compute condition (3) for R_1 since it undergoes a net decrement. In this example, the greatest partial negative changes (δ_1^x) are $-5, -3$ for $x = 1, 2$ respectively; the net changes, $\Delta_1^{loop_x}$ are $-4, -1$ for $x = 1, 2$ respectively. The expressions $\delta_j^x - \Delta_j^{loop_x}$ evaluate to $-1, -2$ for $x = 1, 2$ respectively, and therefore the ordering of loops 1 and 2 in increasing order of this value is $\{2, 1\}$. Consequently, the lowest possible value of R_1 will occur when $loop_2$ is executed last, by Lemma 1. Thus, we first write the conditions when $loop_2$ is executed last: $k_2 > 0$ and $R_1^0 - 4k_1 - k_2 + 1 - 3 = R_1^0 - 4k_1 - k_2 - 2 \geqslant 0$. If $loop_2$ is never executed, we have $k_2 = 0$, $k_1 > 0$ and $R_1^0 - 4k_1 + 4 - 5 = R_1^0 - 4k_1 - 1 > 0$. The disjunction corresponding to condition (3) therefore is:

$$\left(k_2 \neq 0 \wedge R_1^0 - 4k_1 - k_2 - 2 \geqslant 0\right) \vee \left(k_2 = 0 \wedge k_1 \neq 0 \wedge R_1^0 - 4k_1 - 1 \geqslant 0\right)$$

Condition (4) is computed by ordering the loops in increasing order of δ_2^x, which takes the values -2, -1 for $x = 1, 2$ respectively. Thus the desired condition (4) is:

$$\left(k_1 \neq 0 \wedge R_2^0 - 2 \geqslant 0\right) \vee \left(k_1 = 0 \wedge k_2 \neq 0 \wedge R_2^0 - 1 \geqslant 0\right)$$

We could also use conditions (3*) and (4*) to compute a more conservative (not complete) condition for executing k_1 and k_2 iterations of loops 1 and 2:

$$R_1^0 - 4k_1 - k_2 - 2 \geqslant 0 \wedge R_2^0 - 2 \geqslant 0$$

These conditions do not use the loop orderings but miss only a small number of initial register values which would also have allowed the required iterations of both loops.

5.2.1. Accuracy of the computed conditions

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

Definition 11 *(Order independence).* A simple loop with shortcuts is order independent if for every initial valuation of the registers at S_{start}, the set of register values possible at S_{start} after any number of iterations does not depend on the order in which those iterations are taken.

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

A simple loop with shortcuts will have to be order dependent if one of the following holds: (1) the lowest value achievable by a register during its execution depends on the order in which shortcuts are taken. In this case, possible lowest values will impose different constraints for each ordering; or, (2) a *non-spurious* equality condition has to be satisfied to stay in

a loop. In the latter case, the non-deterministic branch leading to the shortcut that has the equality condition will have to be taken at the precise iteration when equality is satisfied. In fact, the disjunction of these two conditions is necessary and sufficient for a loop to be order dependent.

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

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

A naive approach of even expressing the necessary conditions for an order dependent loop can be exponential in the number of shortcuts, even while considering just a single iteration of each loop. We can now see the computation of $\hat{j}$ as handling a very specific kind of order dependence, when the lowest value of a register only depends on the last iteration to be executed.

Example 3. Consider loops l_1, l_2 in the decomposition of a simple loop with monotone shortcuts in an abacus program. 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.

Loops with non-spurious equality constraints are thus special cases of order dependent loops. Although we did not encounter any loops with non-spurious equality constraints in any of the test problems we considered, conditions (1)–(4) can be extended to include equality conditions for the first and last iteration of each loop. Because the registers increase or decrease monotonically, this will make (1)–(4) sufficient (but not necessary) conditions for situations where equality branches are required to stay in the loop. Unfortunately, in the worst case this can also make (1)–(4) unsatisfiable. We can now present two results capturing the accuracy of the conditions (1)–(4).

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

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

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

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

In other words, if we don't have order independence, the conditions (1)–(4) are sufficient, but not necessary. In adversarial formulations however, if the next simple loop to be executed depends on non-deterministic actions, then we require exactly the conditions (1)–(4) which ensure that all the stipulated iterations of all the loops will be executed. In Section 6 we present several examples of this scenario. This leads to the main result of this section, which is analogous to Theorem 1 for simple loops.

Theorem 4. *Let Π be an abacus program, all of whose strongly connected components are simple loops with monotone shortcuts. Let S be any state in the program, and $\bar{F}$ a vector of register values. We can then compute a disjunction of linear constraints on the initial*

Table 1
Timing results for computing preconditions.

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

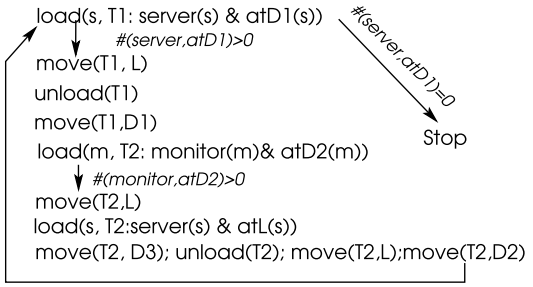

Fig. 7. Solution plan for the transport problem.

register values for reaching S with the register values $\bar{F}$. If all simple loops with shortcuts in Π are order independent, the obtained precondition is necessary and sufficient.

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

Semantics of the computed conditions. Since we are working in the setting where non-deterministic actions are allowed, the variable k_i may implicitly capture the number of times particular outcomes of non-deterministic actions present in $loop_i$ must occur during its k_i iterations. This may appear to be measuring an inherently unpredictable property (non-determinism) and seem to mitigate the utility of the computed preconditions. However, as we will see in Section 6, non-deterministic abacus actions may stand for sensing actions; while we may not be able to predict the outcome of each sensing action, it may still be possible to know how many times a certain outcome is possible, which is all that we need to use the conditions above. In addition, if k_i's are used as parameters, the conditions above capture their tolerable values under which a desired register value may be achieved.

In this section we addressed the problem of determining when a program can reach a certain state with a given register vector by deriving constraints between the initial and final register values for a given abacus program. In order to achieve these results, we used the concept of order independence to summarily deal with a collection of simple loops and the number of times each had to be executed.

These methods could also be applied to deterministic programs but the methods we proposed in Section 5.1 will be more accurate in general. This is because simple loops with shortcuts that are created by deterministic actions are highly order dependent: they include non-spurious equality conditions due to which the order of execution of loops is determined exactly by the initial register values.

6. 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 (references are included with the descriptions below). Existing approaches solve different subsets of these problems, but almost uniformly without termination guarantees [14,1].

Our system takes as input an abacus program or a generalized plan with cardinality changes marked for each action. For every strongly connected component, it first determines if it is a simple loop. If not, it determines whether the component is a simple loop with shortcuts. In order to do so it searches for a loop-orienting node, removal of which would make the entire component acyclic. If no such node exists, or if the shortcuts are found to be non-monotone then the input cannot be handled using our methods and failure is reported. Reachability conditions are constructed for simple loops and simple loops with monotone shortcuts as described in the previous sections. Table 1 shows timing results for 10 different plans.

Plan representation. Figs. 7, 8 and 9 show solution plans for some of the test problems. In order to make the plans easy to read, we show only actions. The default flow of control continues line by line (semicolons are used as linebreaks). 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 to continue executing

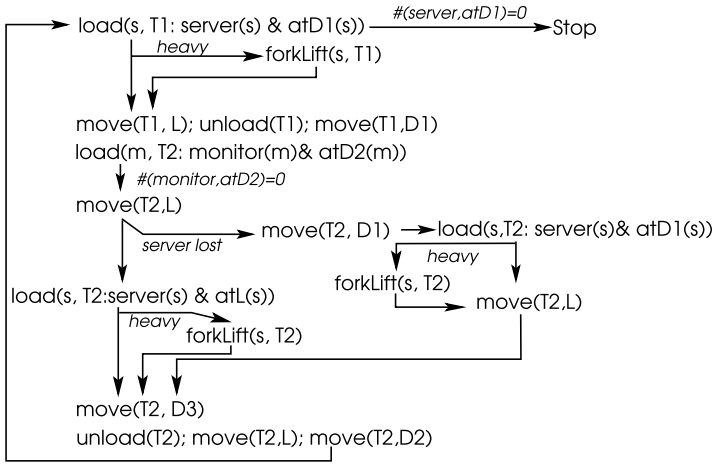

Fig. 8. Solution plan for the conditional version of transport.

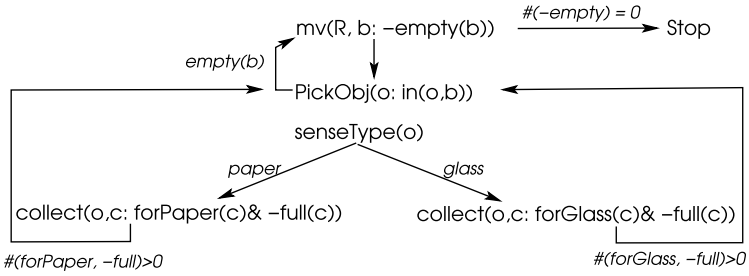

Fig. 9. Solution plan for the recycling problem.

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 nodes are indicated with the action "Stop".

Transport. In the transport problem [21] two trucks have to deliver sets of packages through a "Y"-shaped roadmap. Locations D1, D2 and D3 are present at the three terminal points of the Y; location L is at the intersection of its prongs. Initially, an unknown number of servers and monitors are present at D1 and D2 respectively; trucks T1 (capacity 1) and T2 (capacity 2) are also at D1 and D2 respectively. The goal is to deliver all objects to D3, but only in pairs with one of each kind.

The problem is modeled using the predicates $\{server, monitor, atD_i, inT_i, atL, T1, T2\}$. As discussed in the previous section, role-counts in this representation can be treated as register values and actions as abacus actions on these roles. The plan shown in Fig. 7 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 [23], 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. 8 shows a solution plan found by merging together plans which encountered and dealt with different non-deterministic action outcomes [23]. 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:

$$s_3^f = m_3^f = \sum_{i=0}^{7} k_i, \qquad m_2^f = m_2^0 - \sum_{i=0}^{7} k_i = 0, \qquad s_1^f = s_1^0 - \sum_{i=0}^{7} k_i - k_0 - k_5 - k_6 - k_7 = 0$$

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.

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. 9), with the start state at *PickObj*. *senseType* is a sensing action, and the collect actions decrement the available capacity of each container, represented as the role-count of $\{forX, \neg full\}$ where X is paper or glass. Let e, fg, fp, p, 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:

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

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

Accumulator. The accumulator problem [14] 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 $k \geqslant 1$, 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

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

This implies that $l = k - 1 \geqslant 0$ 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 2.5 GHz AMD dual core system. Problems Hall-A, Prize-A(5) and Prize-A(7) [1] 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 [1] 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 [21], 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.

7. 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 [3] 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 [14] 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 [26] identifies loops from example traces but does not address the problem of preconditions or termination of its learned plans. Bonet et al. [1] 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 [7]. Approaches for *regression* [17,9] in planning directly address the problem of computing preconditions of acyclic plan segments. However, there has been no concerted effort towards finding preconditions of plans with loops. Static analysis of programs deals with similar problems of finding program preconditions [5]. However, these methods typically work with the weaker notion of *partial correctness* [12], where a program is guaranteed to provide correct results *if* it terminates. Methods like Terminator [4] specifically attempt to prove termination of loops, but do not provide precise preconditions or the number of iterations required for termination.

8. Conclusions and future work

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

We established tractability results of reachability analysis for several classes of plans or programs with such actions. For simple loops of actions, this problem admits very efficient algorithms; slight extensions to this class of loops (i.e., simple loops with shortcuts), however, were found to be general enough to capture the full power of Turing machines and therefore had an undecidable reachability problem (Theorem 2) *in general*. On the other hand, the property of *monotonicity* in this case does permit development algorithms for determining reachability, with their accuracy depending upon the notion of order dependence (Proposition 4). Order dependence itself is not very restrictive in non-deterministic situations from an adversarial point of view, where the exact sequence of non-deterministic outcomes of actions cannot be predicted, and we need to plan for the worst case.

These results contribute to the understanding of the factors that make these problems difficult: when order dependence cannot be overcome by conservative approximations, and when the property of monotonicity does not hold. Although non-monotone simple loops with shortcuts have an undecidable reachability problem in the worst case, in some cases the problems of reachability, and at the least termination, can be answered. Further identification of tractable classes of non-monotone simple loops with shortcuts is left for future work. Computation and expression of order dependent preconditions are also important directions for future work on pushing the theoretical limits of solvability of these problems.

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

Acknowledgements

We thank the anonymous reviewers for their detailed and helpful comments. Support for this work was provided in part by the National Science Foundation under grants IIS-0915071, CCF-0830174 and CCF-1115448.

References

[1] B. Bonet, H. Palacios, H. Geffner, Automatic derivation of memoryless policies and finite-state controllers using classical planners, in: Proc. of the 19th International Conference on Automated Planning and Scheduling, 2009, pp. 34–41.
[2] T. Bylander, The computational complexity of propositional STRIPS planning, Artificial Intelligence 69 (1–2) (1994) 165–204.
[3] A. Cimatti, M. Pistore, M. Roveri, P. Traverso, Weak, strong, and strong cyclic planning via symbolic model checking, Artificial Intelligence 147 (1–2) (2003) 35–84.
[4] B. Cook, A. Podelski, A. Rybalchenko, Termination proofs for systems code, in: Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006, pp. 415–426.
[5] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Communications of the ACM 18 (1975) 453–457.
[6] L.C. Eggan, Transition graphs and the star-height of regular events, Michigan Mathematical Journal 10 (1963) 385–397.
[7] R. Fikes, P. Hart, N. Nilsson, Learning and executing generalized robot plans, Technical report, AI Center, SRI International, 1972.
[8] R.E. Fikes, N.J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, Technical report, AI Center, SRI International, 1971.
[9] C. Fritz, S.A. McIlraith, Monitoring plan optimality during execution, in: Proc. of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS), 2007, pp. 144–151.
[10] K.J. Hammond, CHEF: A model of case based planning, in: Proc. of the 13th National Conference on Artificial Intelligence, 1986, pp. 267–271.
[11] M. Helmert, Decidability and undecidability results for planning with numerical state variables, in: Proc. of the Sixth International Conference on Artificial Intelligence Planning and Scheduling, 2002, pp. 44–53.
[12] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (1969) 576–580.
[13] J. Lambek, How to program an infinite abacus, Canadian Mathematical Bulletin 4 (3) (1961) 295–302.
[14] H.J. Levesque, Planning with loops, in: Proc. of the 19th International Joint Conference on Artificial Intelligence, 2005, pp. 509–515.
[15] H.J. Levesque, F. Pirri, R. Reiter, Foundations for the situation calculus, Electronic Transactions on Artificial Intelligence 2 (1998) 159–178.
[16] M.L. Minsky, Computation: Finite and Infinite Machines, Prentice-Hall, Upper Saddle River, NJ, USA, 1967.
[17] R. Reiter, Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, The MIT Press, Massachusetts, MA, 2001.

[18] J. Rintanen, Complexity of planning with partial observability, in: Proc. of the 14th International Conference on Automated Planning and Scheduling, 2004, pp. 345–354.
[19] J.W. Shavlik, Acquiring recursive and iterative concepts with explanation-based learning, Machine Learning 5 (1990) 39–70.
[20] S. Srivastava, Foundations and Applications of Generalized Planning, PhD dissertation, Department of Computer Science, University of Massachusetts Amherst, 2010.
[21] S. Srivastava, N. Immerman, S. Zilberstein, Learning generalized plans using abstract counting, in: Proc. of the 23rd National Conference on AI, 2008, pp. 991–997.
[22] S. Srivastava, N. Immerman, S. Zilberstein, Computing applicability conditions for plans with loops, in: Proc. of the 20th International Conference on Automated Planning and Scheduling, 2010, pp. 161–168.
[23] S. Srivastava, N. Immerman, S. Zilberstein, Merging example plans into generalized plans for non-deterministic environments, in: Proc. of the 9th International Conference on Autonomous Agents and Multiagent Systems, 2010, pp. 1341–1348.
[24] S. Srivastava, N. Immerman, S. Zilberstein, A new representation and associated algorithms for generalized planning, Artificial Intelligence 175 (2) (2011) 615–647.
[25] S. Srivastava, N. Immerman, S. Zilberstein, T. Zhang, Directed search for generalized plans using classical planners, in: Proc. of the International Conference on Automated Planning and Scheduling, Freiburg, Germany, 2011, pp. 226–233.
[26] E. Winner, M. Veloso, LoopDISTILL: Learning domain-specific planners from example plans, in: Workshop on AI Planning and Learning, in conjunction with ICAPS, 2007.