Finding Plans with Branches, Loops and Preconditions

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

We present a new approach for finding conditional plans with loops and branches for planning in situations with uncertainty in state properties as well as in object quantities. We use a state abstraction technique from static analysis of programs to builds such plans incrementally using generalizations of input example plans generated by classical planners. Preconditions of the resulting plans with loops are computed by analyzing the changes in the counts of objects of different types across each loop. The scope and scalability of this approach are demonstrated using experimental results on common benchmark domains.

Introduction

Over the history of AI Planning, the focus of research has been on efficiently finding linear sequences of actions that take a specific problem state to a goal state. Recent work on more expressive representations like plans with loops (Levesque 2005; Winner & Veloso 2007) and high level domain knowledge (Baier, Fritz, & McIlraith 2007) has shown the potential of vast improvements in the scope and performance of planning techniques. As an example, consider a recycling robot that must pick up objects from a set of bins, perform a sensing action to determine recyclability, and store them in appropriate containers. Using tree-structured conditional plans–a common representation in conditional planning (Hoffmann & Brafman 2005; Bryce, Kambhampati, & Smith 2006)–makes the solutions exponential in the number of objects and limits these approaches to small "toy" problems. In contrast, algorithm-like plans that may include loops offer a compact solution to the recycling problem, which consists of a single loop with conditional branches that depend on the type of object. While the benefits of such representations have become clear, there has been relatively slower progress in finding such plans and more importantly, in proving their correctness or finding their preconditions.

In this paper, we present approaches for computing program-like plans that work in multiple situations, and for determining when such plans will work. This area of research inherently involves problems in both verification and planning; in particular, the work presented in this paper addresses two broad problems: (1) Verification of plans with program-like structure with loops and branches (by determining their preconditions), and (2) Generation of plans with such structure for solving multiple problem instances that could vary in the numbers of objects.

Our approach finds plans with branches and "nested" loops, and their preconditions by generalizing and combining multiple, small plans, generated rapidly using a classical planner. In order to do so, we use state abstraction techniques developed for a system for static analysis of programs, TVLA (Sagiv, Reps, & Wilhelm 2002), in the previously unexplored direction of learning generalized plans with preconditions. More precisely, we use these techniques to recognize loop invariants while generalizing example classical plans, and to determine the right positions for merging generalizations of different example plans. Finally, we use a novel algorithm for finding preconditions for some classes of plans with nested loops and branches. To our knowledge, this is the first approach for computing and analyzing plans with such loop and branch structures.

The notion of using abstract states or belief states (Bonet & Geffner 2000) to represent sets of real world states is central to our approach. We begin by summarizing the relevant aspects of the abstraction mechanism we use and how we adapt it for action application in planning problems in the following section. The interested reader is referred to the TVLA system (Sagiv, Reps, & Wilhelm 2002) and the authors' prior work for further details on the framework (Srivastava, Immerman, & Zilberstein 2008b; 2008a). We also introduce our mechanism of sensing actions and observations in the following section. An overview of the overall approach is presented in the section on finding conditional plans with loops. This section also summarizes an algorithm for generalizing example plans by finding loops (Srivastava, Immerman, & Zilberstein 2008b) and presents new methods for merging segments of generalized example plans together. Finally, we present our new approach for finding preconditions of plans with branches and loops, and some results from an implementation of the algorithms presented in the paper. A section with proofs of the main results is included at the end.

Formal Model

We represent states of a domain as traditional (two-valued) logical structures over a domain-specific vocabulary of predicates. A state thus consists of a universe of objects, and for every predicate, a set of object-tuples satisfying it. Domains may include first-order *integrity constraints* that must be satisfied in all instances of the domain. We use the terms "state" and "structure" interchangeably.

Each action is specified as a first-order formula defining its precondition, and a set of update formulas defining the new value of each predicate. Equation 1 shows the update formula for predicate p_i where Δ_i^+ (Δ_i^-) specify when $p_i(\bar{x})$ will be changed to true (false) by the action.

$$p'_i(\bar{x}) := (\neg p_i(\bar{x}) \wedge \Delta_i^+) \quad \vee \quad (p_i(\bar{x}) \wedge \neg \Delta_i^-) \qquad (1)$$

This first order representation of planning is very standard from a logical point of view and can be easily translated to

frame axioms for actions and to successor state axioms in the situation calculus. However, instead of using theorem proving to derive the effects of an action, we use the much more efficient method of formula evaluation on structures.

Example The recycling problem can be modeled using the following vocabulary: $\mathcal{V} = \{\mathit{bin}^1, \mathit{visited}^1, \mathit{object}^1, \mathit{collected}^1, \mathit{empty}^1, \mathit{container}^1, \mathit{forPaper}^1, \mathit{forGlass}^1, \mathit{in}^2, \mathit{isPaper}^1, \mathit{isGlass}^1, \mathit{robotAt}^1\}$. An example structure, S, can be described as follows: the universe, $|S| = \{b, o, c_1, c_2\}$, $\mathit{bin}^S = \{b\}$, $\mathit{object}^S = \{o\}$, $\mathit{container}^S = \{c_1, c_2\}$, $\mathit{forPaper}^S = \{c_1\}$, $\mathit{forGlass}^S = \{c_2\}$, $\mathit{in}^S = \{(o, b)\}$, $\mathit{isPaper}^S = \{o\}$, $\mathit{robotAt}^S = \{b\}$, $\mathit{visited}^S = \{b\}$. We omit the predicates not satisfied by any tuples.

Integrity constraints for the recycling domain would include among others the formula $\forall uvw(\mathit{in}(u,v) \wedge \mathit{in}(u,w) \rightarrow (v = w \wedge (\mathit{bin}(v) \vee \mathit{container}(v))))$ meaning that each object can be in at most one bin or container.

To keep the presentation of the running example very simple, we assume here the artificial integrity constraint that no bin contains more than one object. The goal condition is that all bins are empty: $\forall x(\mathit{bin}(x) \rightarrow \mathit{empty}(x))$.

The precondition and updates for the action $\mathit{collect}(o, c)$ are:

$$\begin{aligned}
(\mathit{isGlass}(o) \quad &\leftrightarrow \quad \mathit{forGlass}(c)) \wedge \mathit{container}(c) \wedge \\
&\qquad \exists b(\mathit{bin}(b) \wedge \mathit{in}(o,b) \wedge \mathit{robotAt}(b)) \\
\mathit{in}'(u,v) \quad &:= \quad (\mathit{in}(u,v) \wedge u \neq o) \vee \\
&\qquad (\neg \mathit{in}(u,v) \wedge u = o \wedge v = c) \\
\mathit{empty}'(u) \quad &:= \quad \mathit{empty}(u) \vee \mathit{in}(o,u) \\
\mathit{collected}'(u) \quad &:= \quad \mathit{collected}(u) \vee o = u
\end{aligned}$$

State Abstraction Using 3-valued Logic

We represent belief states as in Srivastava, Immerman, & Zilberstein (2008b), which in turn is based on the abstraction methodology of TVLA (Three Valued Logic Analyzer), a system for the static analysis of programs (Sagiv, Reps, & Wilhelm 2002). We represent potentially infinite sets of similar concrete structures using an (abstract) 3-valued structure, where the truth value of a tuple being in a relation may be 1 (present), 0 (not present), or $\frac{1}{2}$ (perhaps present). The universe of an abstract structure may include *summary elements*, each of which denotes an arbitrary non-zero number of objects. We draw summary elements using double circles; relations with truth value $\frac{1}{2}$ are drawn using dotted edges, those with truth value 1 are drawn using solid edges and those with truth value 0 are not drawn.

For example, in Fig. 1 the abstract structure S_a contains two summary elements, b, p. Intuitively, S_a represents (or "embeds")[1] any concrete structure that contains one or more non-empty bins, (since *empty* is not written it is false), one or more paper objects, and one glass object. Since concrete structures must satisfy the integrity constrains, we know that each bin contains exactly one object and no object is in more than one bin. Two structures represented by S_a are drawn at the top of Fig. 1. The set of all concrete states represented by S_a is denoted $\gamma(S_a)$. Recall that all states of a domain are required to satisfy the integrity constraints, $\mathcal{I}$. Thus, $\gamma(S_a) = \{S \,|\, S_a \sqsupseteq S; S \text{ concrete}; S \models \mathcal{I}\}$.

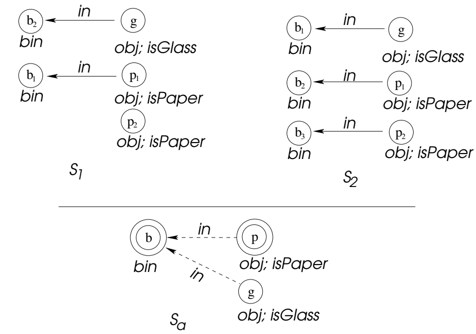


Figure 1: Abstraction for representing belief states

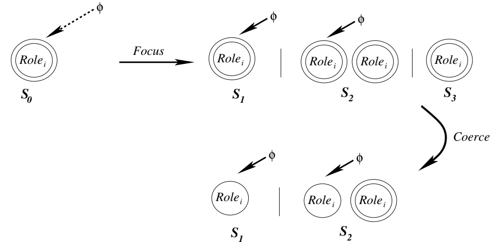


Figure 2: Focus and coerce.

Given a domain, we choose a set, A, of unary predicates to be the *abstraction predicates*. (All the unary predicates in our examples are abstraction predicates.) We define the *role* of an element of a structure to be the set of abstraction predicates it satisfies. In Fig. 1, the role of p is $\{\mathit{obj}, \mathit{isPaper}\}$.

The *canonical abstraction* of a concrete structure, $S^{\#}$, is the least general abstract structure S that represents $S^{\#}$ and has definite truth values for each abstraction predicate (Sagiv, Reps, & Wilhelm 2002). This is computed simply by collapsing all elements of each role to one element of that role. The collapsed element is a summary element if there were multiple elements with that role in $S^{\#}$. Truth values of tuples involving summary elements in S are the most specific generalizations of the truth values of tuples they represent in $S^{\#}$. (In Fig. 1 S_a is the canonical abstraction of S_1, and of S_2.) Maintaining a set of abstract structures is an efficient way to model belief states with uncertainty in object quantities. Note that even though they typically represent infinite collections of concrete states, each canonical abstract structure contains at most 2^a elements where $a = |A|$, the number of abstraction predicates.

Action Application on Belief States

Since we represent belief states using three-valued structures, we can safely apply the (first-order) definitions of the action operators directly to the current belief state to derive the new belief state after the action has been applied. For action, a, and abstract or concrete structure, T, let $\tau_a(T)$ denote the result of applying action a to T.

Fact 1 *If S represents $S^{\#}$ then $\tau_a(S)$ represents $\tau_a(S^{\#})$.*

[1] Formally we say that structure S *represents* structure T (equivalently, T is *embeddable* in S), $S \sqsupseteq T$, iff there is an onto function f from the universe of T onto the universe of S such that for any relation symbol R^k, and any elements, $t_1, \ldots, t_k$ of T, the truth value of $R(f(t_1), \ldots, f(t_k))$ in S, generalizes the truth value of $R(t_1, \ldots, t_k)$ in T ($\frac{1}{2}$ generalizes anything whereas 0 and 1 only generalize themselves).

(Sagiv, Reps, & Wilhelm 2002).

Fact 1 should give the reader an idea of the power and generality of the TVLA abstraction methodology. However, to make this useful, we have to make sure that the belief states stay as precise as possible as we repeatedly apply actions, i.e., we want to maintain definite truth values (0,1) whenever possible. We sketch this process here (see Srivastava, Immerman, & Zilberstein(2008) for details).

While the abstraction is convenient for succinctly representing a large set of possible concrete structures, the designers of TVLA have observed that before an action is applied, it is useful to view the arguments of the action in more detail. They thus introduced the *focus* operation: given an abstract structure, S, and a formula, ϕ, with at most one free variable, focus(S, ϕ) produces a set of structures $S_1, \ldots, S_k$ that represent the same set of concrete structures as S, i.e., $\gamma(S) = \gamma(S_1) \cup \cdots \cup \gamma(S_k)$, but such that the truth value of ϕ is definite in S_i, $i = 1, \ldots k$.

Given an action a, we automatically generate a set of relevant focus formulas, $\phi_1, \ldots, \phi_t$ and focus with respect to all of these. We then apply τ_a to the relevant structures, thus preserving precision. We use the TVLA function *coerce* to refine or remove any structures that do not satisfy the integrity constraints. Finally, we canonically abstract the result structures to return to the standard, abstract representation, no longer focusing on $\phi_1, \ldots, \phi_t$.

In Fig. 2, a simple example of focus is shown, where we are focusing on the formula $\phi(x)$ whose meaning might be that x is the unique argument on which action a will be applied. On the top line, structure S_0 is shown consisting of a single summary element where ϕ has truth value $\frac{1}{2}$. When we focus on ϕ the result is the three structures on the right representing the situations where ϕ has definite truth values and holds for all, some, and none of the elements of the universe, respectively. In the lower line, in the presence of the integrity constraint saying that ϕ must hold for a unique element of the universe, coerce removes S_3 and refines S_1 and S_2. This bottom line shows how we use focus and coerce to *draw-out* action arguments from their summary elements.

Observation Model and Sensing Actions Conditional plans deal with uncertainty in predicates in the agent's belief state using *observation* or *sensing actions* (Bonet & Geffner 2000; Hoffmann & Brafman 2005). In our formulation, sensing actions consist of preconditions and action updates like regular actions. However, the action-specific focus formula for a sensing action is the formula that the action needs to sense. The action specific focus operation for sensing actions thus takes an abstract state and returns a set of more precise belief states corresponding to the different possible definite truth values of the formula being sensed. For instance, the recycling domain has only one sensing action applicable on a drawn-out chosen bin marked with the new (not in the domain's vocabulary) abstraction predicate *chosen*: *senseType()*, with the focus formula $\exists x(chosen(x) \wedge in(o, x))$. When applied to an abstract structure S_a, it returns versions of S_a with different possible combinations of definite truth values for tuples $(_, b)$ being in the *in* relation, where b satisfies *chosen*.

In addition to uncertainty about predicates, we assume that the agent gets limited information about object quantities after each action: it can only determine whether there are zero, exactly one, or more than one objects of each role.

Plan Representation and Execution

We represent conditional plans similar to finite state controllers, using directed graphs whose nodes are labeled with abstract structures and edges are labeled with actions. Edge labels may also include conditions (with the default condition True) under which they may be taken. Execution begins at one of the pre-defined *start* nodes whose structure embeds the agent's initial belief state. At any stage during the plan execution a program-counter (initialized with the start node) labels the active node. The labels of outgoing edges from each node represent the next possible actions. At each step in plan execution one of these actions (say a) for the active node (say n) whose preconditions are satisfied is executed. A neighboring node (connected to n by an edge labeled a) whose structure embeds the resulting belief state becomes the new active node. At any stage, if the next action cannot be carried out, or if a valid node embedding the result state cannot be found, the plan execution ends. A conditional plan **solves** a concrete state $S^{\#}$ if every allowed execution of the plan-steps on $S^{\#}$ starting at an allowed start node ends at a state satisfying the goal; the plan solves a belief state S if it solves every $S^{\#} \in \gamma(S)$ from which the goal is reachable.

Finding Conditional Plans with Loops

Given a set of domain-specific actions, integrity constraints, a goal formula, and an initial belief state S_{init}, our objective is to find a conditional plan solving the initial belief state S_{init}. Alg. 1 provides an overview of our approach. Its input is an initial set of concrete (linear) example plans, and for each plan in this set, a concrete member of S_{init} that it solves.

In the recycling problem for instance, an input example plan could use the sensing actions determining each object's type, but may only work when the type is found to be "paper" (Fig. 3(a)). Such example plans can be provided from prior experience. Alternatively, given an abstract structure S_0 representing initial states, they can be generated by existing *classical* planners as follows: (a) create a concrete member state $S_0^{\#} \in \gamma(S_0)$ with specific truth values for the unobserved predicates. The number of universe elements in $S_0^{\#}$ corresponding to a summary element in S_0 can vary; in this paper we used a heuristic process to add at least six elements in $S_0^{\#}$ for every summary element in S_0. (b) make the appropriate sensing actions for the unobserved predicates as prerequisites for actions that use those predicates (c) solve this problem instance using a classical planner like FF (Hoffmann & Nebel 2001).

The first `while` loop in Alg. 1 incrementally processes example plans from *EgPlans*. In this loop, each example plan is first generalized using the technique developed by (Srivastava, Immerman, & Zilberstein 2008b), resulting in a generalized trace *t*, possibly with loops (Fig. 3(a,b,c)). This process is summarized below. Following generalization, the *Merge* algorithm adds segments of the generalized trace t

Algorithm 1: Generalizing and merging examples

```
Input: EgPlans = {π1 : S1, π2 : S2, ...}
Output: Plan Π
Π ← ∅
while there is a πi : Si ∈ EgPlans do
    Remove πi : Si from EgPlans
    tracei ← generalize(πi, Si)
    Merge(Π, tracei)
    if EgPlans = ∅ and proactiveMode then
        looseEnds = getUnhandledStrucs(Π)
        while looseEnds ≠ ∅ do
            Remove S0 ∈ looseEnds
            π0 ←invokeClassicalPlanner(S0)
            EgPlans ← EgPlans ∪(π0 : S0)
return Π
```

to relevant points in the existing plan Π (initialized with an empty graph) while minimizing new edges (Fig. 3(d,e)). When all members of *EgPlans* have been processed, Alg. 1 can invoke a classical planner to generate new, directed example plans under a "proactive" mode. In order to do this, the domain knowledge should be sufficient to provide possible effects of actions that were not dealt with in the example plans. In this mode, Alg. 1 computes abstract structures that are not solved by Π using *getUnhandledStrucs* as described in the following section. Concrete states for each of these abstract structures are then created, and solved by invoking a classical planner as described above to create additional example plans which are added to *EgPlans*. Partial solutions to these instances are often sufficient (see the results section).

Generalizing Example Plans

The *generalize* subroutine finds loops in abstract traces of sample plans (Srivastava, Immerman, & Zilberstein 2008b). For clarity, we summarize this process and some key results about its analysis in this section. We also provide a detailed example incorporating our new sensing actions. The input to *generalize* is represented as a pair $(\pi, S_0^{\#})$, where $\pi = (a_1, \ldots, a_n)$ is a solution plan for the concrete structure $S_0^{\#}$. The algorithm proceeds as follows: first, π is modified to be applicable to abstract states by replacing its actions' arguments by their roles in the corresponding concrete states, giving us π'. π' is then applied to an abstraction S_0 of $S_0^{\#}$, keeping only that abstract structure S_i at each step which embeds the state $S_i^{\#}$ obtained by π at that step (this is called "tracing"). Repeated abstract structures in this trace indicate that certain state properties have recurred. With an appropriate abstraction, this means that the same actions can be applied again, and is taken as a cue for recognizing a loop. The loop is formed by merging the two abstract structures in the trace. This process is recursively applied on the remainder of the trace after the loop. Finally, the trace with multiple loops is returned.

Note that the original tracing process described above rejected any structure S_i that was not consistent with the result $S_i^{\#}$ in the concrete example. For the purpose of this paper, these rejected structures are included in the trace as open-ended nodes with no following actions, and are extracted by *getUnhandledStrucs* as a compact representation of situations that were not handled.

Example Fig. 3(a) shows a plan segment that collects one object of type paper, moves to the next bin and finds a glass object. $S_0^{\#}$ is a concrete structure in which more than 2 objects each of type paper and glass have been collected, and two bins remain to be visited. Two of the actions in this example, *gotoNextBin* and *senseType*, can have multiple abstract results due to the focus operations described earlier. When applied on an abstract structure with an unknown number of unvisited bins, the two results of the *gotoNextBin* action correspond to whether or not the next bin is the last unvisited bin, as per the drawing-out operation described earlier (Fig. 2). The *senseType* action uses the focus operation to enumerate the different possibilities for the type of the object being sensed. Dotted edges in Fig. 3 represent results of these actions that did not occur in the execution of the given example plan on $S_0^{\#}$.

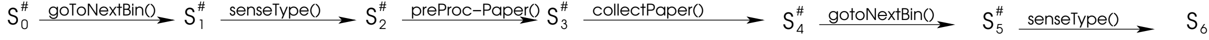
$S_0^{\#}$'s canonical abstraction, S_0, is identical to S_4, the abstract result of collecting another object of type paper. This is recognized during tracing (Fig. 3(b)) and a loop is formed by attaching the "*collectPaper()*" edge to S_0 (Fig. 3(c)). The following action edge (*gotoNextBin()*) from $S_4^{\#}$ however, is not merged with the edge between S_0 and S_1 because $S_5^{\#}$ and its abstraction S_5 do not have any elements with the role of "unvisited bins", thus differing from S_1.

In a fairly general setting ("extended-LL" domains), exact effects of plans with simple loops are determined by easy-to-compute linear functions on *role-counts*, or, the number of elements with a certain role in a structure. This is done by determining the conditions because of which a particular action branch occurs, and then translating these conditions into conditions on the start structure. For instance, the result of *gotoNextBin* on S_0 depends on the number of unvisited bins. Fig. 3(b) shows these conditions, together with automatically computed changes in the counts of various roles caused due to the actions. Expressions for net change in the role-count across any loop determines if the loop makes progress towards the goal state.

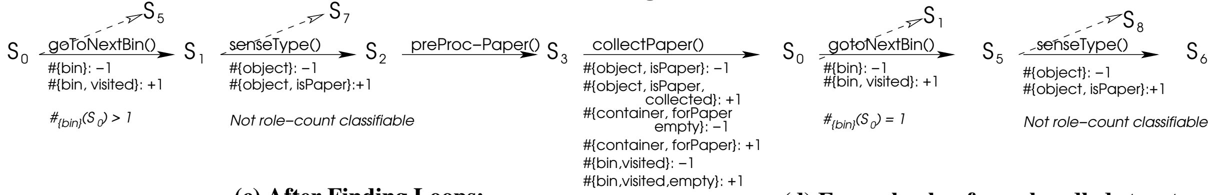
Intuitively, extended-LL domains are those where the unary predicates of a state are sufficient to determine truth values of predicates of higher arities involving the drawn-out objects in that state. The exact relationships between unary and higher-arity predicates may still differ across different states. This class of domains captures many interesting planning problems including the ones discussed in this paper. For completeness, we repeat the definition of extended-LL domains below. A formula φ is *role-specific in S* if there exists a role r such that $\varphi(x) \implies r(x)$ in S.

Definition 1 *(Extended-LL domains)* An *Extended-LL domain with start structure* S_{start} is a domain-schema such that every action a with focus formulas $\{\psi_{a_1}, \ldots, \psi_{a_n}\}$ satisfies the following conditions: if S is reachable via action updates from S_{start} then $\forall i, j$, we have ψ_{a_i} role-specific and either $\psi_{a_i} \equiv \psi_{a_j}$ or $\psi_{a_i} \implies \neg\psi_{a_j}$ in S.

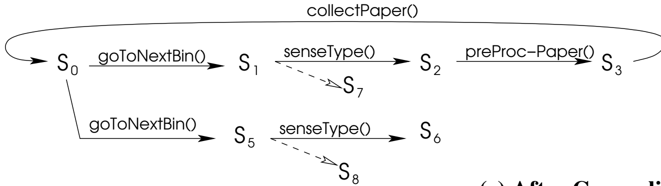
(a) Example plan execution:



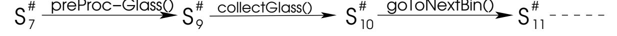
(b) After Tracing:



(c) After Finding Loops:



(d) Example plan for unhandled structure:



(e) After Generalization and Merge:

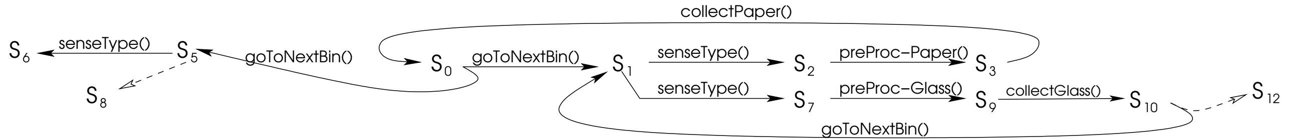


Figure 3: Generalization and merging process in the recycling domain. Dotted edges represent results that did not occur in the example.

We conclude this section with a summary of the methods of analysis of plans with loops presented by Srivastava, Immerman, & Zilberstein (a detailed proof of this fact is included in the appendix):

Fact 2 *Given a plan with simple loops over an extended-LL domain, and a structure node S in the plan, we can compute a set of linear inequalities whose solutions are exactly the achievable role-counts at S. Each of these inequalities either of the form $r_k^0 + l \cdot \delta_k \circ C$, or $r_k^f = l \cdot \delta_k + C$ where r_k^0 represents the role-count of role r_k upon entering the loop; r_k^f is the role-count of r_k at S; δ_k is the (automatically determined) net change in r_k due to the loop; l is the number of iterations of the loop; $\circ$ is $<$ or $=$; and C is a known constant.*

If initial role-counts and numbers of loop iterations are left as variables, these inequalities give the preconditions for reaching a state with a desired role-count, and can be computed in time linear in the number of actions in the plan.

Further, action branches in these domains are determined by linear inequalities on role-counts, and the effect of an action on the role-count of a structure S is determined by a linear function of the initial role-counts. The effect of a loop on role-counts indicates whether or not the loop makes progress towards the goal.

Merging New Segments Using Open Contexts

Merge (Alg. 2) is a greedy algorithm for combining different example plans with sensing actions using abstract structures in generalized traces as representations of possible states, or contexts in plan execution. Given an example trace t_i and an existing plan Π, *Merge* uses *findMergePoint* to find the earliest structure in t_i that is embeddable in a structure in

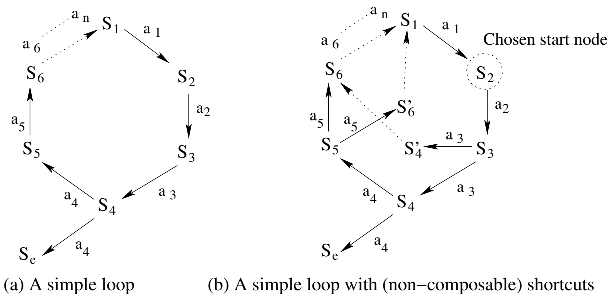
Algorithm 2: Merge

Input: Existing plan Π, eg trace t_i
Output: Extension of Π
if $\Pi = \emptyset$ **then**
  $\Pi \leftarrow t_i$
  return Π
repeat
  $\text{mp}_\Pi, \text{mp}_t \leftarrow \text{findMergePoint}(\Pi, t_i, bp_\Pi, bp_t)$
  if *mp_Π found and not first iteration* **then**
    attachEdges(Π, t_i, bp_t, mp_t, mp_Π, bp_Π)
  if *mp_Π found* **then**
    $\text{bp}_\Pi, \text{bp}_t \leftarrow \text{findBranchPoint}(\Pi, t_i, mp_\Pi, mp_t)$
until *new bp_Π or mp_Π not found*
return Π

Π. In the current implementation, in order to provide accurate expressions of loop effects, structures within loops in t_i are not considered during this search; those within loops in Π are allowed (see the following section on analysis for details). If successful, *findMergePoint* returns mp_Π and mp_t, the nodes on Π and t_i corresponding to these structures. A successful search indicates that the example trace's actions can be successfully executed starting at mp_Π .

However, these actions may not be different from those following mp_Π in Π. In order to to minimize the new edges added to Π, after finding the merge points, *Merge* conducts a search for a branch point using the procedure *findBranchPoint*.

findBranchPoint traverses the edges of t_i and Π starting from the last known merge points mp_t and mp_Π, and re-



(a) A simple loop (b) A simple loop with (non–composable) shortcuts

Figure 4: Simple loops and shortcuts

turns the first pair of subsequent nodes where t_i and Π are not consistent: i.e., either a pair of nodes such that none of the successor actions in Π match any of the successor actions in t_i, or, a pair of nodes n_t, n_Π such that the structure in Π (at n_Π) does not embed the structure in the trace (at n_t). This gives us a branch point, or a situation where the trace behaved differently from the existing plan. In general, the search for subsequent merge points can range over all nodes in Π. However, we bias this search towards finding those merge points for which we can find preconditions as described in the next section. In the current implementation this is done using a heuristic of first searching in the list of nodes in Π that were added after the last branch point in Π, and then searching in the list of all non-ancestors of the last branch point. The list of non-ancestors is obtained by running BFS on Π with its edges inverted, and taking the complement of the obtained set of reachable nodes.

The overall merge algorithm works by attaching nodes and edges from the branch point to the merge point (bp_t, mp_t) in t_i between bp_Π and mp_Π in Π. If a branch point on Π coincides with the next merge point on Π, the *Merge* algorithm introduces a new loop (Fig. 3(d,e)).

Given a generalized plan Π with Π_E edges and a new trace t with t_n nodes, the merge algorithm runs in time $O(\Pi_E \cdot t_n)$.

Analysis of Loop Effects and Preconditions

In this section we illustrate how to find conditions under which the execution of certain kinds of nested loops can be guaranteed to end at a given loop node with given values of role-counts.

We define a *simple* loop as a cycle of nodes, and a *complex* loop as a strongly connected component that is not a simple loop. A *shortcut* in a simple loop is a linear sequence of actions (no branches) starting with a branch caused due to a sensing action in the loop and ending at any subsequent node in the loop that is not after a *chosen* start node. The start node can be any node, but is common to all of a loop's shortcuts (Fig.4).

Simple loops with shortcuts form a very general class– many cases of "nested" loops can be translated into such loops without changing their loop variables or their limits. For instance, perhaps the most common "nested" loop in programming, `for i=1 to n do {for j=1 to k do {xyz}}`, can be turned into a single loop over i with an *if* statement (a branch) resetting j to 1 and incrementing i when $j = k$ is reached. Loops of such kind of any depth, all doubly nested loops and many other so called "nested" configurations can be translated in this way.

For ease in exposition we require that the start nodes of all shortcuts in a simple loop occur at the start node, or otherwise, before the end node of any other shortcut, making shortcuts non-composable in any single iteration of the underlying simple loop. Non-composability allows us to easily count the simple loops caused due to shortcuts independently while computing their overall effects. For instance, we can view the loop in Fig. 4(b) as consisting of 3 different simple loops. Which loop is taken during execution will depend on the results of sensing actions a_3 and a_5.

In the recycling problem for example, (Fig. 3(e)), we get two loops oriented oriented around S_1 as the start node.

Let k_1 represent the number of times *isPaper* branch (corresponding to S_2) is taken, and k_2, the number of iterations of the loop corresponding to the *isGlass* branch (with S_7).

In each of these two loops, except for the branch at S_1, the conditions for Fact 2 hold, allowing us to determine the effect of this complex loop on any role r as $k_1\delta_1^r + k_2\delta_2^r$ where δ_1^r and δ_2^r are total changes in r's role-count due to the two respective loops. For instance, the change in role-count for non-empty, unvisited bins $r_1 = \{bin\}$ is $k_1(-1) + k_2(-1)$ because each loop makes one more element with the role $\{bin\}$ *visited*; the change for $r_2 = \{object, isPaper, collected\}$ is k_1 because this role's count is only changed by the *isPaper* loop which increases it by 1. Achievable role-counts r_1^f and r_2^f at the loop's start structure S_1 after l iterations are thus $r_1^0 - k_1 - k_2$ and $r_2^0 + k_1$ respectively, where r_i^0 denote the initial role-counts. However, this is under the assumption that k_1 and k_2 iterations of the two respective loops *can* be executed completely. Sufficient conditions for ensuring this require that the action branches that exit from the loop (leading to S_{12} or S_5) are not taken. These conditions can be found in a manner similar to that for simple loops used in deriving Fact 2; in the recycling problem this amounts to having at least one non-empty unvisited bin at the start of every iteration. Because the count of r_1 drops by 1 in every iteration of these loops and the *isGlass* loop is entered only after visiting one bin in the first iteration of the nested loop, this can be expressed as $r_1^0 - k_1 - k_2 > 2$. We formalize this result below; details of the procedure, proofs of Fact 2 and the results below can be found in the appendix.

Lemma 1 *Suppose a simple loop with shortcuts in an extended-LL domain with sensing actions is entered with the role-count vector $\bar{r_0}$ at loop node S_i. Then sufficient conditions under which the execution of the loop will end via an action branch from a loop node S_t with the role-count vector $\bar{r_t}$ can be computed.*

The time complexity of determining these conditions is $O(s \cdot n_e \cdot m)$, where m is the number of shortcuts, n_e is the number of edges in the simple loop with shortcuts, and s is the maximum number of roles in any structure in the loop.

Together with the fact that it is possible to find preconditions for reaching a given vector of role-counts at a given structure in a linear generalized plan (Srivastava, Immerman, & Zilberstein 2008a), Lemma 1 above allows us to find sufficient conditions for reaching a given node with given

role-counts in plans that are linear except for multiple simple loops with shortcuts.

Theorem 1 *Let Π be a plan whose loops are simple loops with shortcuts in an extended-LL domain with sensing actions. Sufficient conditions determining the achievable role-counts for any structure in Π can be computed in time linear in the number of actions in the plan.*

Quality of Generalization We measure the quality of plans computed by our algorithm on the basis of the fraction of solvable problem instances that they solve. More specifically, we define $D_\pi(n) = |\mathcal{S}_\pi(n)|/|\mathcal{T}(n)|$ where $\mathcal{T}(n)$ is the set of solvable problem instances of size at most n, and $\mathcal{S}_\pi(n)$ is the subset of those that π solves. For example the recycling problem of size n must have $n/2$ each of bins and bin-contents, yielding a total of $2^{n/2}$ instances with different bin contents.

Implementation and Results

In this section we present the results of some of our experiments with an implementation of *Merge*. The test problems were motivated by benchmarks from the international planning competitions and require solutions with different kinds of loops and branches. Incremental results for each problem are shown in Fig. 5, with segments added due to different examples labeled and drawn with different edge types. The actual outputs are more detailed, and include one iteration of the loop learned using the first example prior to the topmost action shown in the figures. To aid readability, edge labels for results of sensing actions were not drawn and some action operands were summarized into action names. We present a summary of these results with their incremental domain coverages, and provide representative detailed results and execution times for the recycling problem.

Transport We have a Y-shaped transport map with depots D_1, D_2, D_3 on the end points. Two trucks, T_1 and T_2 with capacities one and two are originally at D_1 and D_2, respectively. The problem is to deliver *server* crates (from D_1) and *monitor* crates (from D_2) in pairs with one of each kind to D_3. Location L at the center of the Y can be used to transfer cargo between the two trucks. There are two non-deterministic factors in this problem: *server* crates may be heavy, in which case the simple load action drops them and a forkLift action must be used; crates left at L may get lost if no truck is present.

The first example plan delivered 6 pairs of crates to D_3 without experiencing heavy crates or losses. The second example found a heavy crate, and delivered it using forkLift actions instead of load; in the third plan a crate left at L was found missing when T_2 reached L, and another crate had to be picked up from D_1. The plan computed using these three examples does not handle one case of a *server* crate being heavy (Fig. 5). This was detected using the set of unhandled abstract structures and was handled by example plan 4. The final solution has various branches in the loop that do not qualify as shortcuts described in the section on analysis. However, all such branches include only the *forkLift* action which does not change the roles $r^1 = \{\textit{server}, \textit{atD}_1\}$, $r^2 = \{\textit{monitor}, \textit{atD}_2\}$, $r^3 = \{\textit{server}, \textit{atD}_3\}$ and $r^4 = \{\textit{monitor}, \textit{atD}_3\}$. Because loop exits only depend on the number of crates with these roles, the approach described in the analysis section can determine the counts of these roles at the loop start structure after l iterations of the loop, as $r^1_0 - l - k$, $r^2_0 - l$, $r^3_0 + l$ and $r^4_0 + l$ respectively, where k is the number of times the "crate lost" branch is taken. Loop conditions require all of these counts to be greater than 1, giving us the sufficient conditions for reaching the goal, and implying that we will need extra crates with the role r^1_0 to make up for the losses.

Recycling This problem was used as the running example. The first example plan only encountered paper objects and collected them. The second plan was created to handle an instance of the situation where some bins had glass. The solution example plan handled one bin with a glass object and collected it in the appropriate container. The *Merge* algorithm created a new loop by making the branch point for this example the same as the merge point, illustrating how small examples can be used to identify powerful loops. Example 3 dealt with an unhandled branch caused due to the drawing out of elements from a summary element (last bin was reached), and example 4 handled the case where the last object was of type glass. Analysis of this plan was presented in the previous section.

Fire Fighting A room in a building may be on fire. Smoke can be detected from anywhere on a floor iff one of its rooms is on fire. The agent has smoke and heat sensors; it must use the smoke detector and *goToNextFloor* actions to reach the correct floor, and then use the heat sensors to reach the room with the fire and use the extinguish action to extinguish the fire. In this problem, the first plan covered all the floors but found none to be smoky. The second plan started at a smoky floor and proceeded to search for the room with fire. The *Merge* algorithm found a loop in this example plan, and attached the generalization to a structure in the loop obtained using example 1. The last two plans covered unhandled, boundary conditions where the last floor was smoky or the first room of a floor was on fire. There are no unresolved action branches (considering the *known* possibilities of action results), indicating that the goal structure is always reached.

Key Observations Results of the proposed approach show several novel features. The *Merge* algorithm adds only necessary segments from example plans. For instance, only edges for the two *forkLift* actions from the entire second example in transport were added. In fire fighting, the result of *senseHeat* action in example 4 of the fire fighting problem was directly merged to a structure that had already been handled. Merging plan segments within loops is a powerful technique for increasing the scope of the plan far beyond the individual examples: in recycling, the plan learned using the first example solves only n of the $2^{n+1} - 1$ possible problem instances of size at most n. The second plan covers a single specific problem instance. The generalized, merged result using these two plans solves 2^{n-1} instances (it assumes that the last two bins have paper). All the presented solutions solve problems of unbounded sizes.

Further Details and Comparison We illustrate the incremental increases in domain coverage discussed above with plots and computation times for the recycling problem in

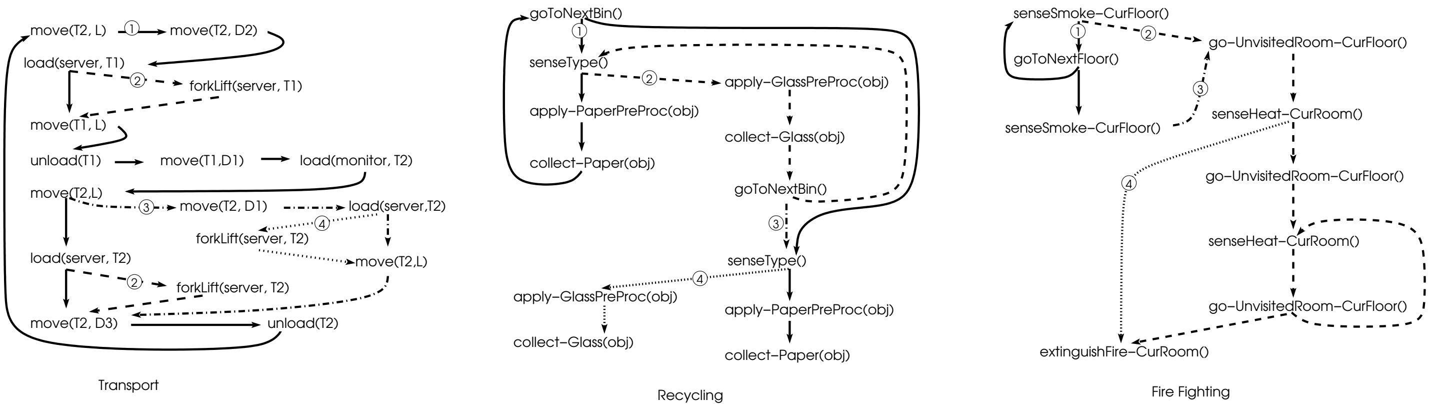


Figure 5: Segments of computed plans. Circled numbers and edge types indicate components added due to different examples.

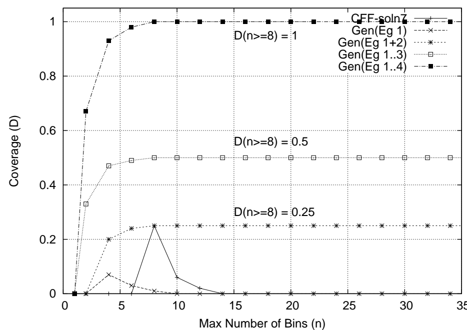


Figure 6: Domain coverage of solutions to the recycling problem.

Plan	Gen(1)	Gen(1..2)	Gen(1..3)	Gen(1..4)	CFF-soln7
Time(s)	110	129	134	144	262

Table 1: Solution Times (see "Further Details and Comparison")

Fig. 6 and Table 1. For our plans, this includes the complete time taken to generalize and merge the input example plans. Since no other approach can solve these problems due to uncertainties in object quantities, comparisons with other approaches are not possible. However, to put this in perspective, we compared these results with the domain coverage and execution time for the largest recycling problem instance (with 7 bins) that we could solve using contingent-FF (Hoffmann & Brafman 2005), a well-established contingent planner. Given the four example plans for recycling described above, the generalization and merging process produces a near complete solution while taking 45% lesser time than the time taken by contingent-FF to find a plan (CFF-soln7) for 7 bins. Solutions to all the other problems discussed above were generated in under 300 seconds and showed similar comparative performance with contingent-FF. These tests were carried out on a machine with a 64-bit AMD Dual-Core 2.5GHz processor and 2GB RAM.

Related Work

Using loops in plans has been previously proposed and analyzed. Winner & Veloso (2003; 2007) present methods for combining example plans into plans with branches and loops. However, this approach does not provide methods for determining if loops make progress, and finds only non-nested loops. Levesque (2005) presents an approach (KPLANNER) for iteratively solving problems of increasing sizes and extracting patterns in the solutions to determine loops that generalize a single given numeric planning parameter. Levesque notes that "even short iterative programs can be quite difficult to reason about". He concludes that "faced with an intractable reasoning problem, we can look for compromises... [and] forego the strong guarantees of correctness". We present a new approach for addressing these challenges in a comprehensive manner for a general class of loops. However, in this paper we focus on problems without numeric variables.

Cimatti *et al.* (2003) consider domains where loops are needed for actions which may have to be repeated for success. They also provide methods for determining reachability of goal states in this context. However, the loops considered by this approach are "hard" loops, in the sense that they return to the exact same problem state. In contrast, our objective is to find loops that make measurable, incremental changes. Hansen & Zilberstein (2001) also present a method for computing policies with hard loops of actions, but in a setting where probabilities of action outcomes and their rewards are used to determine the action which would lead to the best possible value.

Conclusions and Future Work

We present two fundamental techniques to improve the scalability of planning systems. The first technique efficiently combines sample plans produced by classical planners and produces a conditional plan that can solve problems of unbounded sizes. The second technique automatically computes–for a rich class of plans with branches and loops–measures of progress of their loops and the set of problems solved by a given plan. These contributions address problems that are generally known to be undecidable, but we identify an interesting subclass for which we show that they are efficiently solvable. Experimental results show that the approach is scalable, efficient in terms of the strength of generalizations produced and robust in terms of the types of loops allowed.

This work opens up several directions for future research and application, including a greater utilization of existing operator sequences with further analysis of plan segments,

and studying other methods of implementing the merge algorithm which was implemented using a greedy approach here. Our approach is also unique in providing a direction for automatically finding verifiable domain control knowledge, which has been shown to yield significant benefits in classical planner performance (Baier, Fritz, & McIlraith 2007).

Acknowledgments

Support for this work was provided in part by the National Science Foundation under grants IIS-0535061, CCF-0541018, CCF-0830174 and IIS-0915071.

References

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proc. of ICAPS*, 26–33.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proc. of AIPS*, 52–61.

Bryce, D.; Kambhampati, S.; and Smith, D. E. 2006. Planning graph heuristics for belief space search. *J. Artif. Intell. Res. (JAIR)* 26:35–99.

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.

Hansen, E. A., and Zilberstein, S. 2001. Lao*: A heuristic search algorithm that finds solutions with loops. *Artif. Intell.* 129(1-2):35–62.

Hoffmann, J., and Brafman, R. I. 2005. Contingent planning via heuristic forward search witn implicit belief states. In *Proc. of ICAPS*, 71–80.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

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

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. 2008a. Foundations of Generalized Planning. *Technical Report UM-CS-2008-039, Dept. of Computer Science, Univ. of Massachusetts, Amherst*.

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

Winner, E., and Veloso, M. M. 2003. Distill: Learning domain-specific planners by example. In *Proc. of ICML*, 800–807.

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

Proofs of Results

Simple Loops in Extended-LL Domains

For clarity we restate a result first presented by (Srivastava, Immerman, & Zilberstein 2008b), and its proof in the language of the current submission. This result does not work for sensing actions or complex loops.

Proposition 1 *Suppose $S_1 \xrightarrow{a_1} S_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} S_n \xrightarrow{a_n} S_1$ is a loop in an extended-LL domain. For any $1 \le i \le n$ we can compute a set of linear inequalities $C(l)$ which determine the role-counts at S_i after l iterations of the loop starting at S_1, plus the simple path from S_1 to S_i.*

PROOF Since we are in an extended-LL domain, every action changes a structure's role-counts by a constant amount ((Srivastava, Immerman, & Zilberstein 2008a): Theorem 2 and def. of extended-LL domains). We denote the role-counts in a structure using vectors. For example, $\bar{R}^0 = \bar{R} = \langle \#R_1^0, \#R_2^0, \ldots, \#R_m^0 \rangle$ denotes the initial counts of roles $R_1, \ldots R_m$ at structure S_1. Let R_{b_i} be the branch role for action a_i, i.e., the role whose count determines which branch is taken at action a_i. In extended-LL domains, the action branch that is taken when an action is applied to an abstract structure is determined by whether a certain role-count is greater than 1, or equal to 1 ((Srivastava, Immerman, & Zilberstein 2008a)).

We use subscripts on vectors to denote the corresponding role-counts, so the initial count of the branch-role at action a_i is $\bar{R}^0_{b_i}$. If there is no branch at action a_i, we let $b_i = d$, some unused dimension. Let Δ^i denote the role-count change vector for action a_i. Let $\Delta^{1..i} = \Delta^1 + \Delta^2 + \cdots + \Delta^i$.

Before studying the loop conditions, consider the action a_4 in Fig. 4(a). Suppose that the condition that causes us to stay in the loop after action a_4 is that $\#R_{b_4} > 1$. Then the loop branch is taken during the first iteration starting with role-vector $\bar{R}^0$ if $(\bar{R}^0 + \Delta^{1..4})_{b_4} > 1$. This branch will be taken in l subsequent loop iterations iff $\bar{R}^0 + k \cdot \Delta^{1..n} + \Delta^{1..4}_{b_4} > 1$, and similar inequalities hold for every branching action, for all $k \in \{1, \ldots, l-1\}$.

More precisely, the conditions for a full execution of the loop starting with role-count vector $\bar{R}^0$ are:

$$\begin{array}{rcl} (\bar{R}^0 + \Delta^{1..1})_{b_1} & \circ & 1 \\ (\bar{R}^0 + \Delta^{1..2})_{b_2} & \circ & 1 \\ & \vdots & \\ (\bar{R}^0 + \Delta^{1..n})_{b_n} & \circ & 1 \end{array}$$

$\circ$ is one of $\{>, =\}$ depending on the branch that lies in the loop; the entire set of inequalities can be simplified by removing constraints that are subsumed by others. The only variable term in this set of inequalities is $\bar{R}^0$. Let us call these inequalities $\mathsf{LoopIneq}(\bar{R}^0)$. For executing the loop l times, the condition becomes

$$\mathsf{LoopIneq}(\bar{R}^0) \wedge \mathsf{LoopIneq}(\bar{R}^{l-1})$$

where $\bar{R}^{l-1} = \bar{R}^0 + (l-1) \times \Delta^{1..n}$. These two sets of conditions ensure that the conditions for execution of intermediate loop iterations hold, because the changes in role-counts due to actions are constant, and the expression for $\bar{R}^{l-1}$ is linear in them.

If $\bar{F}$ denotes the final role-counts at S_i after l iterations,

we have

$$\begin{array}{rcl} & \mathsf{LoopIneq}(\bar{R}^0) & \\ & \mathsf{LoopIneq}(\bar{R}^{l-2}) & \\ (\bar{R}^{l-1}+\Delta^{1..1})_{b_1} & \circ & 1 \\ (\bar{R}^{l-1}+\Delta^{1..2})_{b_2} & \circ & 1 \\ & \vdots & \\ (\bar{R}^{l-1}+\Delta^{1..i-1})_{b_{i-1}} & \circ & 1 \\ \bar{F} & = & \bar{R}^{l-1}+\Delta^{1..i} \end{array}$$

These conditions on the role vector $\bar{R}^0$ at S constitute $C(l)$. Note that in order to compute this set of conditions we only need to compute at most n different $\Delta^{1..i}$ vectors. $C(l)$ can be computed in $O(s \cdot n_l)$ time, where s is the maximum number of roles in a structure in this loop, and n_l is the number of actions in the loop.

□

Note that final set of inequalities in the proof given above include the exact role counts for all roles after l iterations of the loop. Together with the ability to compute changes in role counts across linear sequences of actions (see (Srivastava, Immerman, & Zilberstein 2008a)), this allows computation of not only whether a path with simple loops can take a certain concrete structure to a desired goal structure, but also the *exact* number of times we need to go around each loop in the path, in order to reach the desired structure with desired role counts.

Preconditions for Simple Loops with Shortcuts

We now state and prove precise versions of Lemma 1 and Theorem 1.

Lemma 1 *Let Π be any plan in the form of a simple loop with m non-composable, monotone shortcuts. Suppose $k_1, \ldots, k_m$ represent the number of times shortcuts $1, \ldots, m$ are taken during the execution of Π. In extended-LL domains with sensing actions, sufficient conditions for the achievable role counts r^i_f at any structure-node S_x are given by the following system of linear inequalities:*

$$\begin{array}{rcl} f^i_x(r^i_0, k_1, k_2, \ldots, k_m, l) & = & r^i_f; \\ k_1 + \ldots + k_m & \leq & l; \\ \forall j : LL_j < r^0_j, f^j_x(r^j_0, k_1, k_2, \ldots k_m, l) & \leq & UL_j \end{array}$$

where LL_j, UL_j are the lower and upper limits for role r^j for staying in the loop (the last inequality comes from Proposition 1 above), and l is the total number of iterations counted at the start node.

PROOF Suppose we are given a plan with a simple loop with m shortcuts and a chosen start node S_{start}. Because the shortcuts are constrained to be non-composable and monotone, the idea is to consider the simple loops formed by taking each of the m shortcuts independently.

In Fig. 4(b), this would give us 3 simple loops:

$$S_1, S_2, S_3, S_4, S_5, S_6, ..., S_1;$$
$$S_1, S_2, S_3, S'_4, ..., S_6, ..., S_1;$$
$$S_1, S_2, S_3, S_4, S_5, S'_6, ..., S_1.$$

We denote the loop created by taking the ith shortcut as $loop_i$, and the original simple loop taken when none of the shortcuts are taken as $loop_0$.

Within each of these loops, the assumptions used in computing the inequalities $C(l)$ in Proposition 1 hold, because these loops do not have any branches due to sensing actions. In other words, the only action branches that have to be constrained for completing an execution in any of these loops come from non-sensing actions in extended-LL domains and are determined by inequalities between role-counts and constants.

Let k_i denote the number of times $loop_i$ is executed in full, with $k_0 = l - \sum_{i=1}^{m} k$. Then the final role-counts can be computed as $\bar{F} = \bar{R}^0 + \sum_{i=0}^{m} k_i \Delta^{loop_i}$ obtained by adding the changes due to each loop using proposition 1, where Δ^{loop_i} is the change vector due to $loop_i$. Finally, in order to ensure that the loop conditions hold for every intermediate iteration, we include the constraints $\mathsf{LoopIneq}(\bar{R}^0)$ and $\mathsf{LoopIneq}(\bar{F})$, for every loop. For the partial loop iteration between S_{start} and S_x, we add to $\bar{F}$ the change due to the linear sequence of actions leading from S_{start} to the structure node S_x to obtain $\bar{F}_x$, and include any conditions due to the non-sensing actions. For details about computing constraints for linear sequences of actions, see Theorem 1 of (Srivastava, Immerman, & Zilberstein 2008a). If S_x is on a shortcut, then we get an additional constraint that the sensing action result leading to that shortcut should occur in the last iteration of the loop.

Finally, the desired form of the linear constraints is obtained by setting f^j_x as the jth component of $\bar{F}_x$. □

Using Lemma 1, we can compute linear constraints for achievable role counts at any structure node in a plan consisting of a linear path of actions with simple loops with shortcuts at multiple positions in the path. Conditions for exiting from a loop through a non-sensing action are enforced by including the appropriate role-count inequality (similar to (Srivastava, Immerman, & Zilberstein 2008a)).

While this gives us sufficient conditions to achieve a certain role-count at a given node if the loop iteration counters k_i's and initial role-counts are left as variables, this does not deal with effects caused due to the merging of different paths of actions.

Theorem 1 *Let Π be a plan whose loops are simple loops with shortcuts in an extended-LL domain with sensing actions. A disjunction of linear inequalities determining the achievable role-counts for any structure in Π can be computed in time linear in the number of actions in the plan.*

PROOF Consider each linear path having simple-loops-with-shortcuts at multiple positions in the path. Linear constraints for each such path can be determined using the linear constraints developed explicitly under Lemma 1 and the methods for computing constraints for linear sequences of actions ((Srivastava, Immerman, & Zilberstein 2008a)). Given a reachable node S_n and the set of such paths leading to it, the disjunction of linear constraints corresponding to each path gives us linear constraints for achievable role-counts at S_n due to the union of those paths. □