Challenges in Finding Generalized Plans

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 simple and precise definition of generalized planning together with five natural dimensions of quality for measuring any generalized plan. We argue that no existing approach excels in all these dimensions.

In the remainder of the paper we present a new approach to generalized planning that addresses all five of these dimensions.

The Problem of Generalized Planning

Over the years, many researchers have addressed the problem of coming up with a "general plan" that solves many different planning problems. Methods for doing so have included, among others, indexing and cataloging observed plans, annotating previously observed action sequences for future use, theorem proving, boolean satisfiability, etc. All of these approaches can be understood as developing generalized plans which "efficiently" map problem instances to "good" concrete, deterministic plans for solving them. This leads us to the following natural definition:

Definition 1 A *generalized plan* is an algorithm, G, that maps problem instances, i, to a sequence of actions, s that solves i. We call the concrete plan $s = G(i)$ the *instantiation* of the generalized plan, G, for instance i.

A classical planner is the simplest form of a generalized plan. On the other hand, an algorithm, for instance the "Unstack" algorithm in blocks world for moving all blocks to the table, is a very specific generalized plan which produces output plans for the problem instances that it can solve much more easily than a classical planner:

$$\text{while}(\exists b(\text{clear}(b) \wedge \neg\text{on-table}(b)) : \text{moveToTable(b)}$$

For this particular generalized plan, it is also very easy to test whether it can solve a given problem instance: the goal should be to have all blocks on the table.

In finding generalized plans, any approach needs to address some common challenges:

1. Complexity of checking applicability
2. Complexity of plan instantiation
3. Quality of the instantiated plan
4. Domain coverage
5. Complexity of computing the generalized plan

The most fundamental of these are problems of designing an efficient applicability test and computing an instantiated plan.

Complexity of checking applicability A generalized plan can be designed to proceed in one of two ways when given an input problem instance: (1) conduct a pre-designed applicability test to determine if an instantiation will be possible, and if so, proceed to find it, or, (2) directly attempt an instantiation. The problem with the second approach is that instantiation can be an expensive and wasteful operation if the generalized plan cannot actually solve the given problem instance. While the first approach is desirable, it is often very difficult to construct an applicability test; the ideal situation would be to have a linear time applicability test.

Traditional approaches to finding generalized plans seldom offer applicability tests. KPLANNER(Levesque 2005), as an exception, provides a partial test: within the user-requested bounds on a unique parameter that its input problem instances are allowed to vary over, its generalized plans are guaranteed to produce a correct instantiation. Approaches like case-based planning (Spalzzi 2001) incur large costs of applicability and instantiation while retrieving and adapting potentially applicable previously observed plans; modern approaches like DISTILL (Winner & Veloso 2003) also do not provide applicability tests.

Complexity of plan instantiation Plan instantiation represents the actual process of constructing a concrete, deterministic plan that can solve a given problem instance. For problems that require observations such as those of conditional planning, generation of the sequence s can be interleaved with its execution. In this form of plan instantiation, successive actions in s are generated by the generalized plan after taking the available observations into account. The complexity of plan instantiation distinguishes generalized plans like Unstack above, with plan instantiation time quadratic in the number of blocks (or better if lists of topmost blocks are maintained), from classical planners whose worst-case complexity of plan instantiation is exponential in the number of objects.

Quality of the instantiated plan The quality of instantiated plans produced determines the relative usability of a generalized plan. Ideally, instantiated plans should be optimal according to a measure like the number of actions in the plan or its makespan. Again, determining how well or poorly a generalized plan's instantiations perform is not always easy to determine, and most approaches focus on finding *any* working instantiation.

Domain Coverage Another important factor determining the quality of a generalized plan is the set of distinct problem instances (i.e., requiring distinct operator sequences as solutions) of interest that it solves, or its domain coverage. Concrete plans produced by classical planners actually score very well as generalized plans on all the factors discussed so far, except that they typically work for only one problem instance. This aspect of increasing the domain coverage of

a generalized plan is in fact the single most focused aspect among all the approaches to generalized planning developed so far, and the original motivating factor behind the development of these approaches (Fikes *et al.* 1972). Conditional plans cover larger possible problem instances than classical plans. However, as we discuss below, their coverage is typically limited because conditional plans tend to grow very large.

Complexity of computing a generalized plan Conditional planning produces generalized plans with clear applicability tests (by definition, they solve all instances of the provided initial belief state) and are easy to instantiate. However, the decision tree structured representations that they use for expressing solutions can grow exponentially with every unknown predicate tuple, making plans inherently more difficult to find. Plan representation thus becomes an important factor when considering the complexity of deriving a generalized plan itself. Approaches like DISTILL, KPLANNER, and BAGGER2 (Shavlik 1990) mitigate this cost by constructing plans with loops that can instantiate into much larger concrete plans.

The five factors discussed above determine the quality of a generalized plan. While various approaches have addressed different subsets of these factors, there are none that address all of them. In the sequel, we describe such an approach.

Our Approach

We use three-valued logical structures to represent potentially infinite sets of problems states containing a potentially unbounded number of objects. We compute generalized plans by merging generalized versions of concrete plans. Loops in these generalized plans allow them to handle problems of unbounded size. For a large class of problems, we can automatically determine which instances the current generalized plan can solve. Furthermore, for an incomplete generalized plan, we can then automatically generate an unsolved instance, solve this with a classical planner, generalize the resulting solution, and, finally, extend the generalized plan by incorporating the new solution.

Running Example In the rest of this paper, we will use the recycling problem as a running example: a recycling robot must pick up objects from a set of bins, perform a sensing action to determine recyclability of the drawn object, and store it in an appropriate container.

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.

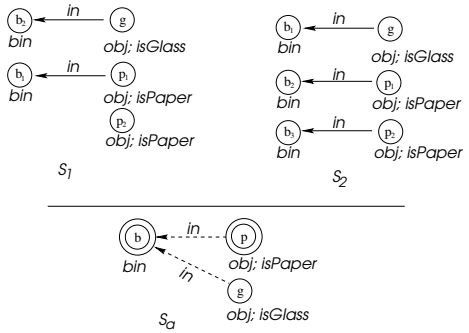


Figure 1: Abstraction for representing belief states

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

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:

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

Definition 2 *(Generalized Planning Problem)* A *generalized planning problem* is a tuple $\langle \mathcal{V}\mathcal{A}, \mathcal{K}, \mathcal{I}, \phi_g \rangle$, where $\mathcal{V}$ is the vocabulary of predicates, $\mathcal{A}$ a set of action operators, each consisting of preconditions and predicate update formulas, $\mathcal{K}$ is a set of integrity constraints expressed in first-order logic, $\mathcal{I}$ is a set of initial states (i.e., the "problem instances"), and ϕ_g is the common goal formula.

State Abstraction Using 3-valued Logic

We represent belief states as in Srivastava *et al.* (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 *et al.* 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, $\gamma(S_a) = \{S \mid S_a \sqsupseteq S; S \text{ concrete}; S \models \mathcal{I}\}$.

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 $\{obj, isPaper\}$).

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 *et al.* 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. While doing so, note that the predicate update formulas (Eq. 1) may lead to some predicates getting truth values of $\frac{1}{2}$.

[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).

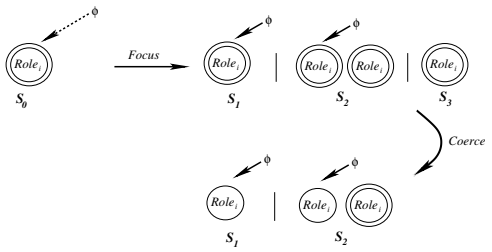


Figure 2: Focus and coerce.

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 $\tau_a(S^{\#})$. (Sagiv* et al. *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 *et al.*(2008a) 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 generalized plans like finite state controllers, as 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 generalized 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 Generalized Plans

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 generalized plan solving the initial belief state S_{init}. Alg. 1 provides an overview of our approach. Its input is a set of concrete (linear) example plans and for each, 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).

Algorithm 1: Generalizing and merging examples

```
Input: EgPlans = {π_1 : S_1, π_2 : S_2, ...}
Output: Plan Π
Π ← ∅
while there is a π_i : S_i ∈ EgPlans do
    trace_i ← generalize(π_i, S_i)
    Merge(Π, trace_i)
    if EgPlans = ∅ and proactiveMode then
        looseEnds = getUnhandledStrucs(Π)
        while looseEnds ≠ ∅ do
            Remove S_0 ∈ looseEnds
            π_0 ←invokeClassicalPlanner(S_0)
            EgPlans ← EgPlans ∪(π_0 : S_0)
return Π
```

Alg. 1 proceeds as follows. An input example plan is first generalized using a technique developed in prior work (Srivastava *et al.* 2008b), resulting in a generalized trace *t*, possibly with loops (Fig. 3(a,b,c)). This process is summarized below. Following this step, the Merge algorithm adds segments of the trace t to relevant points in the existing plan Π (initialized with an empty graph) while minimizing new edges (Fig. 3(d,e)). If the domain knowledge determines all possible effects of actions, then Alg. 1 can provide the abstract structures that are not solved by Π, using the subroutine *getUnhandledStrucs* as described in the following section. Concrete states for each of these structures are then created and solved by a classical planner as described above to create additional example plans.

Generalizing Example Plans

The *generalize* subroutine finds loops in abstract traces of sample plans (Srivastava *et al.* 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.

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 3 *(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.

We conclude this section with a summary of the methods of analysis of plans with simple, non-nested loops presented in prior work (Srivastava *et al.* 2008a; 2008b).

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.

A detailed proof of this fact along with some other results from this paper are available at `www.cs.umass.edu/~siddhart/appendices/genplan09`.

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 Π. 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 returns 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

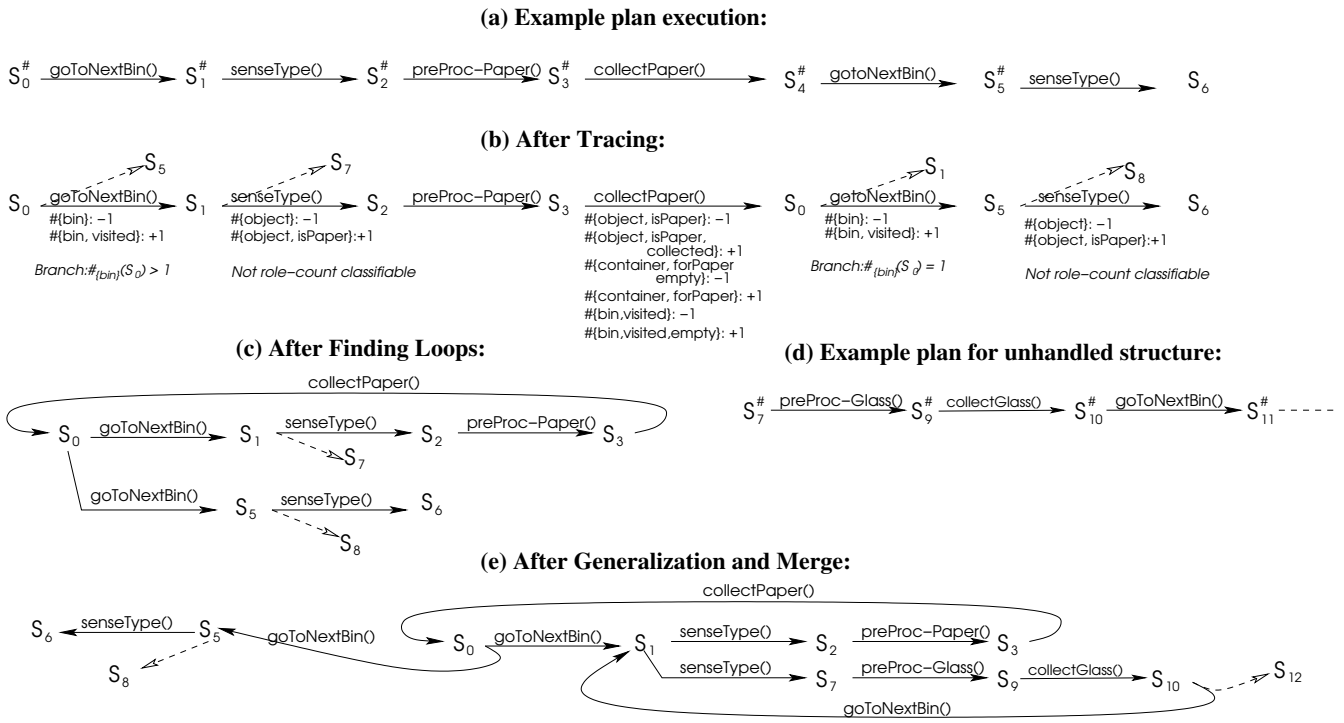


Figure 3: Generalization and merging process in the recycling domain. Dotted edges represent results that did not occur in the example.

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

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 $O(\Pi_E \cdot t_n)$.

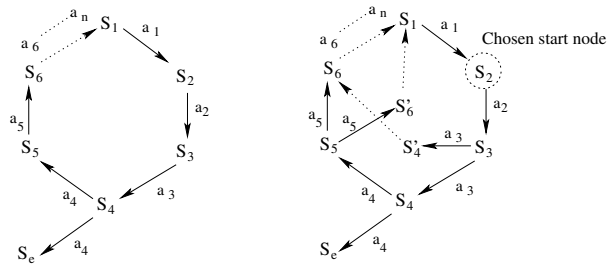
Loop Effects and Preconditions

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 the *isPaper* branch



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

Figure 4: Simple loops and shortcuts

(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. We need to include conditions for ensuring that that 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 (see the branch conditions in Fig. 3(b)). 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 at `www.cs.umass.edu/~siddhart/appendices/genplan09`.

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 *et al.* 2008a), Lemma 1 gives us the following result:

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.*

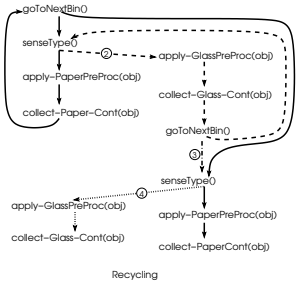


Figure 5: A section of the computed generalized plan for recycling

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 "Quality of Generalization")

Results

We implemented a prototype version of *Merge*. In order to provide accurate expressions of loop effects, structures within loops in the incremental traces were not considered during the search for merge points by the subroutine *findMergePoint*; those within loops in the existing generalized plan were allowed.

We show the result of applying this implementation to the recycling problem in Fig. 5. In this figure, components of the plan added due to different examples are drawn with different edge types and numbered accordingly. 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. Note that the plan learned using the first example solves only n of the $2^{n+1} - 1$ possible problem instances with at most n bins. The second plan covers a single specific problem instance. The generalized, result produced by adding the second example increases the coverage exponentially, solving 2^{n-1} instances (it assumes that the last two bins have paper).

We also used this implementation to solve problems with sensing actions in transport with the chance of packages being lost. The results showed similar comparisons with conditional planners.

Quality of Generalization We measure the quality of plans computed by our algorithm on the basis of their domain coverage. 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

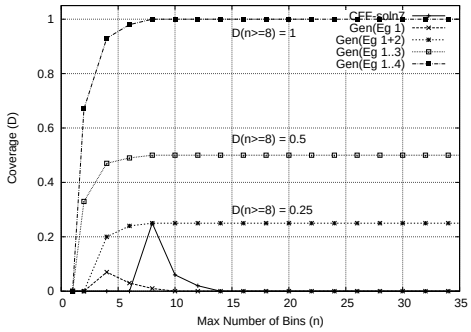


Figure 6: Domain coverage of solutions to the recycling problem.

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.

We illustrate the incremental increases in domain coverage discussed above with plots and computation times for the recycling problem in 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 were 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.

Discussion and Conclusions

In this section we summarize how our approach addresses the challenges described in the introductory section and discuss its current limitations.

Complexity of checking applicability Our preconditions are of the form $r_k^0 + l \cdot \delta \circ c$, where δ and c are constants and $\circ$ is $=$ or $<$. Given an initial role count r_k^0, we can easily determine, for each such inequality if there is a value of l that satisfies these equations. The number of such inequalities is of the order of the number of edges in the plan. In extended-LL domains, our method is guaranteed to be able to compute these preconditions.

Complexity of computing an instantiation Because the choice actions only use roles, instantiating our generalized plans amounts to instantiating an object corresponding to every action argument's role. This can be done at run-time; the cumulative complexity of instantiating the entire plan is linear in the maximum number of objects in any encountered state and the steps in the plan. The cost can be lower in practice if lists of objects of each role are maintained.

Quality of instantiated plan For recycling, the solution generalized plan yields optimal instantiated plans for all the problems that it can solve. In general, the quality of instantiated plans depends on the quality of input example plans.

Domain Coverage Using complex loops our generalized plans handle infinite classes of problems via a small representation; we developed a more detailed metric for domain coverage in the previous section.

Complexity of computing the generalized plan Our approach utilizes the advances in classical planners to compute small classical plans for use in generalization. The use of loops for handling recurrent scenarios allows us to rapidly expand the domain coverage without increasing the size and complexity of deriving the generalized plan itself. Among the other existing approaches to finding generalized plans, ours is unique in providing methods for automatically computing preconditions. Apart from lowering the cost of testing applicability, this is the main idea in our approach for proactively strengthening the generalized plans (Alg 1).

Currently, our approach for determining preconditions is limited to extended-LL domains with sensing actions. This restriction is due to our abstraction mechanism which can handle unary predicates with complete precision. It is much more challenging to keep appropriate precision when modeling higher-arity predicates, for example, the binary order of disk sizes in the general towers of Hanoi problem.

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

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

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

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.

Shavlik, J. W. 1990. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning* 5:39–40.

Spalzzi, L. 2001. A survey on case-based planning. *Artif. Intell. Rev.* 16(1):3–36.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008a. Foundations of generalized planning. *Technical Report UM-CS-2008-039, 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.