Directed Search for Generalized Plans Using Classical Planners*

Siddharth Srivastava and **Neil Immerman** and **Shlomo Zilberstein**
Department of Computer Science
University of Massachusetts Amherst

Tianjiao Zhang
Department of Computer Science
Mount Holyoke College

Abstract

We consider the problem of finding provably correct generalized plans for situations where the number of objects may be unknown and unbounded during planning. The input is a domain specification, a goal condition, and a class of concrete problem instances or initial states to be solved, expressed in an abstract first-order representation. Starting with an empty generalized plan, our overall approach is to incrementally increase the applicability of the plan by identifying a problem instance that it cannot solve, invoking a classical planner to solve that problem, generalizing the obtained solution and merging it back into the generalized plan. The main contributions of this paper are methods for (a) generating and solving small problem instances not yet covered by an existing generalized plan, (b) translating between concrete classical plans and abstract plan representations, and (c) extending partial generalized plans and increasing their applicability. We analyze the theoretical properties of these methods, prove their correctness, and illustrate experimentally their scalability. The resulting hybrid approach shows that solving only a few, small, classical planning problems can be sufficient to produce a generalized plan that applies to infinitely many problems with unknown numbers of objects.

1 Introduction

In real-life domains, an agent may need to tackle situations without precise information about the number of objects of each type. For instance, a transport planner may not know the number of objects to be transported or the number of locations to be visited; a recycling agent that must sort objects from a bin into boxes for different types may not know the number of objects present in the bin. Two significant points motivate the study of such problems: robustness in handling real-world problems, and the need for scalable planning. Indeed, if a general solution can be found, it can obviate the need for planning explicitly in state spaces whose sizes grow exponentially with increasing numbers of objects.

For true applicability across problem instances with different quantities of objects, generalized plans need to include cyclic or recursive structures. Plans with such representations are difficult to verify for correctness; very few approaches have been developed for finding such generalized plans (Shavlik 1990; Hu and Levesque 2010; Winner and Veloso 2007; Srivastava, Immerman, and Zilberstein 2011)—and even fewer attempt to provide guarantees of correctness, or even termination, of their computed plans (Hu and Levesque 2010; Srivastava, Immerman, and Zilberstein 2011).

Any control structure, acyclic or cyclic, is likely to be applicable on a broad class of problem instances (cyclic structures hold greater potential but at a greater risk of unexpected outcomes or non-terminating computation). One approach to this problem, therefore, is to compute cyclic plans for solving a given problem instance and then, to study the class of instances which they can solve. In this paper, we address a significantly different problem: given a class of problem instances of interest, our objective is to produce a generalized plan that can solve as many of *these instances* as possible. In doing so, we *guarantee* that the possible outcomes of executing a computed generalized plan will always be well-defined. Previously, only Hu and Levesque addressed this particular problem, but only for problem instances that varied on a *single* numeric parameter.

We build upon techniques originally developed for static analysis of programs to compute provably correct generalized plans for solving a given class of problems. Our method can be viewed as an incremental process for interleaving the construction of a generalized plan with a validation phase for computing an over-approximation of the set of problem instances that are not solved by the existing plan. Starting with an empty generalized plan, we repeatedly construct a concrete example, or an instance, of the given general problem that is not yet handled by the generalized plan, solve it using a classical planner, generalize it, and assimilate this solution with the current generalized plan. The process terminates when the generalized plan handles the entire desired class of concrete problem instances or when a predefined resource limit is reached.

We focus on three key questions: how to efficiently determine the class of open problems for an existing partial generalized plan? how to efficiently generate small instances of this class? and how to use the solution in order to increase the scope of the generalized plan. We employ a standard classical planner to solve concrete problem instances, thus leveraging existing powerful heuristic search capabilities in this process. We start with a description of the formal framework, followed by the details of the proposed approach and

*A version of this paper will be presented at at ICAPS-2011.

analysis of its properties. The concluding sections discuss experimental results and directions for future work.

2 Formal Framework

We use a state and action representation developed in prior work (Srivastava, Immerman, and Zilberstein 2011). In this formalization of planning, states and actions are represented using first-order logic with transitive closure (FO(TC)). We provide a brief overview of this representation. States are represented as logical structures in a domain's vocabulary as shown below.

Example 1. *A typical blocks world vocabulary would consist of the relations* $\{\text{on}^2, \text{topmost}^1, \text{onTable}^1\}$. *Although* topmost *and* onTable *can be defined in terms of* on*, for clarity in presentation, we will treat each of these relations as distinct. An example structure, S, in this vocabulary can be described as:* $|S| = \{b_1, b_2, b_3\}$, $onTable^S = \{b_3\}$, $topmost^S = \{b_1\}$, $on^S = \{(b_1, b_2), (b_2, b_3)\}$.

Each action a consists of a precondition $pre(a)$ and update formulas defining the new value of each predicate p after a has been applied.

Example 2. *In the blocks world, the action* move *has two arguments:* obj_1*, the block to be moved, and* obj_2*, the block it will be placed on. Update formulas for* on *and* topmost *are:*

$$\begin{aligned} on'(x,y) &= \neg on(x,y) \wedge (x = \text{obj}_1 \wedge y = \text{obj}_2) \\ &\vee \quad on(x,y) \wedge (x \neq \text{obj}_1 \vee y = \text{obj}_2) \\ topmost'(x) &= \neg topmost(x) \wedge (on(\text{obj}_1, x) \wedge x \neq \text{obj}_2) \\ &\vee \quad topmost(x) \wedge (x \neq \text{obj}_2) \end{aligned}$$

Since the ensuing abstraction retains precision on unary predicates, we assume that the goal formula is expressed entirely using unary predicates. Most planning benchmarks can be easily reformulated into this representation. For example, in the blocks world we can use a new predicate to express the goal positions of blocks, and a unary predicate, *inplace*, to express a goal condition. *inplace* would be set for a block by the move action whenever that block is placed upon the correct block, as required in the goal.

2.1 Abstract States and Actions

We need to represent compactly sets of states with unbounded quantities of objects. For this purpose, we use *canonical abstraction*, an abstract representation originally developed for the TVLA system of static analysis of programs (Sagiv, Reps, and Wilhelm 2002). This representation also allows a sound application of actions in an abstract state space. In prior work, we developed a formulation of this approach for planning (Srivastava, Immerman, and Zilberstein 2011). We provide a summary of the most relevant aspects of this formulation below.

In canonical abstraction, a subset of the unary predicates in the domain, A, is identified as the set of *abstraction predicates*. In all examples used in this paper, the set of abstraction predicates is the set of all unary predicates. We define the role of an element in a state as the set of abstraction predicates that it satisfies. The abstraction of a structure is

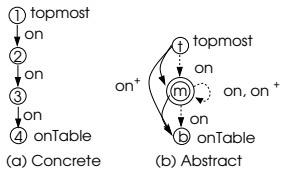

Figure 1: Canonical abstraction of a tower of blocks

computed by merging together all elements with a role into one element with that role. The collapsed element is a *summary* element if there were multiple elements with that role in the original structure. This results in a structure with at most one element per role. Essentially, this abstraction uses symmetry in a domain structure to reduce the effective size of its universe. The notion of symmetry however is restricted to that revealed by the unary predicates. Exploiting symmetry in structures is a well established methodology in model checking (Emerson and Sistla 1996).

A third truth value, $\frac{1}{2}$, is used to express the non-definite information about relations involving summary elements. Truth values of tuples involving summary elements are set to be the most specific generalizations of the truth values of tuples they represented. For example, Fig. 1(b) represents the canonical abstraction of the four block tower shown in Fig. 1(a). In this figure, dotted edges represent predicates with the truth value $\frac{1}{2}$, solid edges, the truth value 1, and false predicates are not shown. Double circles represent summary elements; for clarity, the transitive closure of on, on^+ is shown only on the right. Transitive closure is necessary for expressing relationships between unknown numbers of objects, such as the fact that the topmost block in Fig. 1(b) is above all the other blocks.

Integrity Constraints A set of integrity constraints can be used to clarify the set of legal structures represented by an abstract state. Integrity constraints in the TVLA system are expressed as first-order formulas of the form $\psi(x_1, \ldots, x_n) \rhd \varphi(x_1, \ldots, x_n)$, where ψ is any first-order formula with free variables $x_1, \ldots, x_n$ and φ is a predicate or its negation. These constraints are enforced only when the truth value of ψ is found to be 1 for an interpretation of its free-variables in a structure. In that case, the truth value of the atom on the right is set so that the literal φ evaluates to 1. For example, the following integrity constraint in blocks world asserts that a block cannot be *onTable* if it is on another block: $\exists v_2 \;\; on(v, v_2) \rhd \neg onTable(v)$. TVLA's *Coerce* algorithm enforces each integrity constraint on a structure immediately before and after performing action updates.

In generalized problems where only a subset of the instances represented by an initial structure is solvable, we also allow integrity constraints to specify this subset. In particular, we allow integrity constraints that express inequalities between *role-counts*, or the number of objects of each role in a structure. For example, in the blocks world we can assert that the number of blocks with the role $\{blue\}$ exceeds the number of $\{red\}$ blocks by 3: $\#\{blue\} = \#\{red\} + 3$. To summarize, we use integrity constraints to constrain the set of concrete states represented by an abstract state to just the states that are truly valid and of interest.

Semantics of Abstract Structures To define what it

means for a structure to represent another structure, we first define a precision ordering, "$x \prec y$", to mean that y is more precise than x, i.e., $x = \frac{1}{2}$ and $y \in \{0, 1\}$. Let $x \preceq y$ mean that $x \prec y$ or $x = y$. Structure S_2 *represents* structure S_1 (denoted as $S_1 \in \gamma(S_2)$) iff S_1 is *embeddable* in S_2 and S_1 satisfies the integrity constraints. An embedding is a map from $|S_1|$ onto $|S_2|$ that does not change truth, but may lose precision:

Definition 1. (Embeddings) *The function* $f : |S_1| \underset{onto}{\rightarrow} |S_2|$ embeds S_1 *in* S_2 ($S_1 \sqsubseteq^f S_2$) *iff for all relation symbols* p^a *and elements,* $u_1, \ldots, u_a \in |S_1|$, $[\![p(f(u_1), \ldots, f(u_a))]\!]_{S_2} \preceq [\![p(u_1, \ldots, u_a)]\!]_{S_1}$.

Abstract structures can thus represent states with unknown quantities of objects: Fig. 1(b) represents the set of all towers of height at least three.

Action Application on Abstract States Action arguments need to be separated from their summary elements prior to the action update. Otherwise, indefinite truth values on relations involving summary elements tend to propagate to all predicate tuples through successive action updates.

We accomplish this using "choice" actions, which select a representative element of a given role prior to the real, state-transforming actions. For instance, the *mv*($\mathrm{obj}_1, \mathrm{obj}_2$) action described in Eg. 2 would be preceded by actions *choose* obj_1*:* $role_1$ and *choose* obj_2*:* $role_2$, where $role_i$ are the roles of obj_1 and obj_2 respectively in the structure on which this update is applied. A choice operation can result in two outcomes if the role of the object being chosen was represented by a summary element. These two outcomes correspond to cases where the number of elements represented by this summary element was exactly one, or more than one. After executing choice operations, action updates can be applied on each resulting abstract structure. For action, a, and abstract or concrete structure, T, let $\tau_a(T)$ denote the result of applying action a to T. The following result captures the soundness of action application in this framework:

Fact 1 *If* S *represents* $S^\#$ *then* $\tau_a(S)$ *represents* $\tau_a(S^\#)$. *(Sagiv, Reps, and Wilhelm 2002).*

This implies that if the truth value for any formula, (e.g. the goal formula, or a formula representing an action's preconditions) is 1 or 0 in $\tau_a(S)$, then that formula *must* evaluate to 1 or to 0 respectively in $\tau_a(S^\#)$, for every $S^\# \in \gamma(S)$.

This action mechanism, though already sound, is made more precise for practical purposes. Just prior to action application on an abstract structure, the truth values of action preconditions and update formulas are made precise. This is done using the *Focus* operation. Given a set of first-order formulas and an abstract structure, Focus generates multiple abstract structures corresponding to all the possible definite (0 or 1) truth values of the given formula. The Focus formulas for an action consist of all predicates used in an action's update and preconditions. Each Focused result is processed by Coerce, and action updates are applied on the results only if their preconditions evaluate to 1. If certain restrictions on the actions of a domain hold, the outcomes of a focus operation on a structure depend only on the *role-counts* in that structure. Domains that satisfy these restrictions are called extended-LL domains (Srivastava, Immerman, and Zilberstein 2011). In particular, domains that are expressible in STRIPS using only unary predicates are extended-LL domains.

In concluding this section, we note that Fact 1 continues to hold when Focus is used, but with the single structure $\tau_a(S)$ replaced by the set of structures obtained by applying action updates on each of the Focused, Coerced results of S.

2.2 Plan Representation and Execution

Our representation of generalized plans is similar to finite state controllers. We use directed graphs whose nodes are labeled (via a labeling function *Struc()*) with abstract structures and edges are labeled with actions. Structure labels denote an over-approximation of *all* possible states that may occur at that node in the generalized plan. 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 represents the set of initial problem states. 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 state becomes the new active node. A generalized plan **solves** a concrete state $S^\#$ if every allowed execution of the plan on $S^\#$ ends at a state satisfying the goal after a finite number of operations.

The Generalized Planning Problem We define the generalized planning problem as follows:

Definition 2. *Given an abstract initial state* S_0*, a set of actions* $\mathcal{A}$*, a set of integrity constraints* $\mathcal{K}$ *and a goal formula* φ_g*, compute a generalized plan which solves every* $S \in \gamma(S_0)$.

In this paper we only consider problems where the concrete members of S are the fully observable initial states, or problem instances, that we wish to solve.

3 Generalized Plan Synthesis

We focus on finding generalized plans with two critical properties: that they represent valid executions, and that any cyclic flow of control must terminate after a finite number of iterations. We formalize these two properties as follows:

Definition 3. (Well-defined executions) *A generalized plan* Π *satisfies the property of well-defined executions iff:*

- *It is guaranteed to terminate after a finite number of actions on every instance represented by its start node.*
- *During any execution, if the flow of control is at an internal node, at least one of the outgoing edges is applicable.*

Therefore, a generalized plan with well-defined executions must terminate, and when it does, it must do so at a node with no outgoing edges. We call terminal nodes whose structure labels satisfy the goal formula, the *goal nodes* of the plan. All terminal, non-goal nodes are referred to as *open nodes*. If we maintain the property of well-defined executions, then executing a generalized plan can only end in a

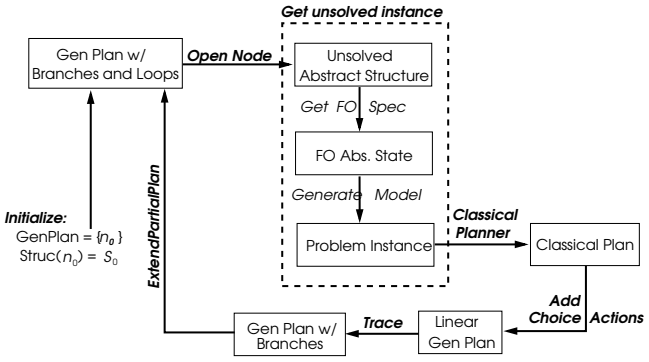

Figure 2: Architecture of the proposed approach: ARANDA-hybrid

concrete state represented by the labels of its terminal nodes, which may either be goal nodes or open nodes. This facilitates our overall approach, which incrementally finds paths from a plan's open nodes to the goal.

Fig. 2 shows our overall approach (ARANDA-hybrid) for generalized plan synthesis; the corresponding algorithm is presented as Alg. 1. The generalized plan is initialized with a single node labeled with a set of initial states represented using an abstract structure S_0. Thus, S_0 labels the first open node in this algorithm. The main loop in Alg. 1 iteratively picks an open node n, obtains a classical plan solving one of the instances represented by *Struc(n)*, generalizes that plan, and finally, merges it with the existing generalized plan. The open node to be solved in each iteration can be prioritized.

Given an open node n, steps 4-6 generate a concrete instance, C_n, of the set of states represented by $S_n = \textit{Struc(n)}$. This process is discussed in Section 3.1. In step 7, a classical planner is invoked with a PDDL version of C_n, an instantiated goal formula and a PDDL version of the domain $\mathcal{D}$. *AddChoiceActions* then generalizes the obtained plan π_n by inserting argument-selecting choice actions before each of π_n's actions. The choice action for selecting an action argument is constructed using the role of that argument, extracted from the concrete state on which the action was applied in π_n. This operation requires knowledge of the sequence of concrete states visited by the obtained classical plan. This can be easily obtained by modifying the classical planner or by simulating the execution of the obtained plan on C_n.

Subroutine *Trace* in step 9 is derived from the "generalize" subroutine introduced earlier (Srivastava, Immerman, and Zilberstein 2011) to compute the portions of S_n that π_n^c may not solve. In this subroutine, abstract action operators are applied to abstract structures starting with S_n. Whenever an action leads to multiple abstract states due to Focus or choice operations, the next action from π_n^c is applied only on the state that embeds the result obtained at that step in an execution of the plan π_n^c upon the original concrete instance C_n. Other structures at each step represent situations that were not encountered in the execution of π_n. These structures are also stored in the trace, using the same representation as generalized plans. The main segment of a trace is a path of abstract states and actions consistent with what occurred in the execution of π_n^c on example C_n; some of these states have secondary edges which terminate at nodes

Algorithm 1: Hybrid Generalized Plan Synthesis

Input: Abstract structure S_0, domain $\mathcal{D}$, goal formula φ_g
Output: Generalized Plan Π

1 $\Pi \leftarrow \langle V = n_0, E = \emptyset, \textit{Struc}(n_0) = S_0, \textit{OpenNodes} = (n_0)\rangle$
2 **repeat**
3 Pick an open node n
4 RoleCounts $\leftarrow$ GetValidRoleCounts($S_n = \textit{Struc}(n)$)
5 $\varphi_n \leftarrow$ GetFOSpec(Struc(n), RoleCounts)
6 $C_n \leftarrow$ ModelGenerator(φ_n)
7 $\pi_n \leftarrow$ ClassicalPlanner(PDDL_Translation(C_n), $\mathcal{D}_{PDDL}$)
8 $\pi_n^c \leftarrow$ AddChoiceActions(π_n, PDDL_Translation(C_n))
9 $t_n \leftarrow$ Trace(π_n^c, *Struc*(n))
10 $\Pi \leftarrow$ ExtendPartialPlan(t_n, Π, n)
until *OpenNodes* = $\emptyset$ *or ResourceLimitReached*

representing structures that were not consistent with the example execution. In this way, members of *Struc*(n) that are not solved by the generalized example π_n^c get collected in the trace's open, or non-goal terminal nodes. Step 10 assimilates this trace into the existing generalized plan. This is described in detail in Section 3.2.

3.1 Generating Concrete Instances

Given an abstract structure S_n, we need to generate a concrete member, preferably as small as possible, of $\gamma(S_n)$. As shown below, any abstract structure can be represented as an FO(TC) formula. The problem therefore is that of model generation, which could have been solved by any existing first-order model generator—if transitive closure had not been used. As discussed above, however FO(TC) is necessary to represent relationships between unknown, unbounded numbers of objects. Below, we utilize a theoretical result to compile transitive closure into first-order logic.

As the first-step in model generation, *GetValidRoleCounts* generates an instance of the role-count inequalities included in integrity constraints using a mathematical package (we used *Mathematica*). Next, the subroutine *GetFOSpec* constructs a first-order representation φ_n, of the abstract structure S_n. φ_n consists of three sets of axioms, Ax_u, Ax_e, and Ax_i, capturing facts about the elements of the universe, its relations, and the integrity constraints respectively. Recall that every element in an abstract structure corresponds to a unique role, and that its summary elements may correspond to multiple elements in a concrete structure that it represents. Let $r(S)$ be the set of roles with non-empty instantiations in S and $u(S_n)$ be the set of roles that correspond to singleton elements in S_n. We use the abbreviation $r_i(x)$ to denote first-order formulas for roles. That is,

$$r_i(x) \iff (\wedge_{p_j \in \textit{role}_i} p_j(x) \wedge_{p_k \notin \textit{role}_i} \neg p_k(x))$$

Ax_u ensures that every element has one of the roles $r(S_n)$ and that roles corresponding to singleton elements hold for unique elements:

$$Ax_u(S_n) \equiv \forall x\, (\vee_{r_i \in r(S_n)} r_i(x)) \wedge \wedge_{r_j \in u(S_n)} \forall x, y\, (r_j(x) \wedge r_j(y) \implies x = y) \quad (1)$$

Ax_u is extended in a straightforward manner to assert that every role holds for the number of elements returned by *GetValidRoleCounts*.

For expressing the constraints on relations, the key observation is that every element in S will be characterized by a unique role, due to canonical abstraction. Ax_e asserts truth values between elements for every pair of roles that have a definite relationship in S:

$$Ax_e(S_n) \equiv \bigwedge_{v \in \{\top,\bot\}} \bigwedge_{\substack{p, r_i, r_j: \\ [\![p(r_i, r_j) = v]\!]_{S_n} = 1}} \forall x, y \, (r_i(x) \wedge r_j(x) \implies p(x, y) = v) \quad (2)$$

Ax_i consists of all the integrity constraints with $\rhd$ replaced by $\implies$. We now need to write an FO formula defining the transitive closure predicates used in Ax_e and Ax_i. We will use the new predicate p^{tc} as the transitive closure of a predicate p in the vocabulary of a domain-schema. Consider the following FO formula:

$$T_1[p] \equiv \forall x, y \, (p^{tc}(x, y) \iff p(x, y) \vee (\exists z \, p(x, z) \wedge p^{tc}(z, y)))$$

The result below ensures that if the relation p is constrained to be acyclic, then in every finite structure, $T_1[p]$ defines p^{tc} as exactly the non-reflexive, transitive closure of p.

Theorem 1. *(Lev-Ami et al. 2009) In any finite and acyclic model of $T_1[p]$, p^{tc} is equivalent to p^+, the non-reflexive transitive closure of p.*

We found that transitive closure in this restricted setting suffices to express necessary state properties such as "every block in the tower is above the bottommost block", "every grid square in a row is to the west of the eastern border" etc. Such expressions are used in all but the delivery and transport problems presented in Section 4.

Let PTC be the set of predicates in the domain for which transitive closure is used. The first-order expression for Ax_i therefore includes, in addition to any integrity constraints used in the domain, the axioms $T1[p]$ and the statements $\forall x \neg p^{tc}(x, x)$, for all $p \in PTC$.

For a three-valued structure S, let $FO(S) \equiv Ax_u(S) \wedge Ax_e(S) \wedge Ax_i$. The following result shows that step 6 in Alg. 1 generates a concrete instance $C_n \in \gamma(S_n)$ as long as the model generator employed is sound.

Theorem 2. *Suppose that all $p \in PTC$ are acyclic and S is an abstract structure corresponding to an open node in a generalized plan. Then a concrete structure C belongs to $\gamma(S)$ iff $C \models FO(S)$.*

Proof. Suppose $C \in \gamma(S)$. Then there must be an embedding of C into S. Because this embedding can only make truth values imprecise, it will have to map an element in $|C|$ to an element in $|S|$ of the *same* role. Thus, C must satisfy the first conjunct in Ax_u, stating that every element must have one of the roles in S. Under this embedding, multiple elements of $|C|$ (say $c_1, \ldots, c_m$) can be mapped to a single element (say s_1) of $|S|$, only if $[\![s_1 = s_1]\!]_S$ is $\frac{1}{2}$. Otherwise, the truth value of $s_1 = s_1$ in S will become inconsistent with at least one of $[\![c_1 = c_2]\!]_C$ (= 0) and $[\![c_1 = c_1]\!]_C$ (= 1). In other words, there must be exactly one element in C for every singleton roles $u(S)$ in S. Thus, we have $C \models Ax_u$. By a similar reasoning, C must satisfy Ax_e: otherwise we get a tuple whose truth value on a predicate in C conflicts with the truth value of the corresponding tuple in S.

Algorithm 2: ExtendPartialPlan

Input: Trace $t = (s_0, a_0, s_1, O_1), \ldots, (s_m, a_m, s_{m+1}, O_{m+1})$, GP Π, open node *prev*
Output: Extended version of Π

```
1  for (s_i, a_i, s_{i+1}, O_{i+1}) in t do
2      for (prev, n_k) in outgoing edges from prev do
3          if (prev, n_k) subsumes (s_i, a_i, s_{i+1}) then
4              prev = n_k; e = (prev, n_k)
5              matched = True
6              break
           end
       end
7      if not matched then
8          n = getSafeNodeToJoin(s_{i+1}, prev)
9          e = addEdge(prev, n, a_i, s_{i+1}, Π)
       end
10     addUnsubsumedOpenNodes(O_{i+1}, e, Π)
   end
11 return Π
```

In order to show that $C \models Ax_i$, we first note that $T_1[p]$ is a sound axiom scheme: it is always satisfied by a model which interprets p^{tc} as the correct transitive closure of p. Since we have $C \in \gamma(S)$, C interprets p^{tc} correctly, and therefore must satisfy $T_1[p]$ for every $p \in PTC$. Finally, by definition of $\gamma(S)$, we know that C must satisfy all the domain-specific integrity constraints. Thus, we have $C \models Ax_i$.

Conversely, suppose $C \models FO(S)$. We need to construct an embedding from $|C|$ into $|S|$ and show that C satisfies all the integrity constraints, and interprets p^{tc} correctly. Because $C \models Ax_u$, we know every element's role in C corresponds to an element's role in S. Further, S must have at most one element of each role because the action update mechanism merges multiple elements with the same role into a summary element of that role. The required embedding therefore maps every element in $|C|$ to the element with the same role in $|S|$. Axiom Ax_e ensures that this mapping is an embedding. Finally, $C \models Ax_i$, so C satisfies all the integrity constraints, and all $p \in PTC$ must be acyclic. This implies that C must interpret p^{tc} correctly, due to Thm. 1. □

3.2 Extending the Generalized Plan

Alg. 2 closes an open node by assimilating a trace representing a path to a goal state. In doing so, it needs to maintain the property of well-defined executions by ensuring that at least one of the action edges added to a node will be applicable when that node is reached during execution. Given a trace t, a generalized plan Π, and an open node *prev*, Alg. 2 extends Π to include the flow of control represented by t, starting at *prev*. For brevity, we represent the trace as a sequence of tuples in this description. Each element of the trace consists of a structure s_i, the action a_i which was applied on s_i, the result s_{i+1} of this application that was consistent with the result observed in the example execution and the set O_{i+1}, possibly empty, consisting of the other abstract result structures. As noted before, these structures represent the trace's open nodes, or cases not handled by the traced plan.

Main Loop Alg. 2 iterates over each tuple in the trace. The main loop operates under the invariant that the initial

structure s_i of the current trace tuple is embeddable in the structure labeling *prev*. In step 2, it searches for an outgoing edge from *prev* that leads to a node whose structure label embeds s_{i+1} and whose action matches a_i. If such a node is found, this edge is recorded as e. If no such outgoing edge is found (step 7), then this means that the control flow in the input trace differed from the existing plan, giving us a new path to the goal. This case always holds in the first iteration of the main loop, because *prev* was an open node for Π and had no outgoing edges. In this case, the subroutine getSafeNodeToJoin is called.

Finding candidate nodes to join In step 8, subroutine getSafeNodeToJoin returns a node n, such that (a) *Struc(n)* embeds s_{i+1}, (b) only simple loops with shortcuts[1] (Srivastava, Immerman, and Zilberstein 2010a) may be created by adding an edge from *prev* to *n*, and (c) any simple loop with shortcuts created in this way *will* terminate after a finite number of iterations. If no such node is found, getSafeNodeToJoin creates a new node in the plan and returns that node. getSafeNodeToJoin searches for nodes first in the set of ancestors and then in the set of non-ancestors of *prev*. The condition that only simple loops with shortcuts are created is enforced by only considering ancestors from which no path to *prev* passes through a cycle.

Termination of simple loops with shortcuts is established by identifying a role r whose role-count, (the number of elements satisfying r) always shows a net decrement in *every* possible single iteration of the simple loop with shortcuts. This is an efficient test that ensures termination because a role-count can never fall below zero. Changes in role-counts due to actions in a generalized plan can be determined efficiently in extended-LL domains. In general, functions that can be shown to decrease in every iteration of a loop are referred to as *ranking functions* in model checking. Any approach for synthesis of ranking functions can be used in getSafeNodeToJoin to ensure that all created loops terminate.

Edge Addition In step 9, an edge is added between *prev* and *n* with appropriate labels. As discussed above, this edge may create terminating loops. Note however that these loops may not be evident from a syntactic analysis of repeating patterns in the concrete plans themselves. In fact, concrete plans may even be too small to exhibit clear repetitive behavior. ExtendPartialPlan still, consistently, found useful loops in experiments where the generated instances were two small to exhibit repetitive patterns. For instance, none of the plans generated for delivery (Section 4) executed two complete iterations of the identified loop.

Maintaining Open Nodes Finally, step 10 conducts bookkeeping for storing open nodes together with the edge e. If e already existed in the plan, it only adds open nodes that are not embeddable in the other nodes attached to e's start node. In extended-LL domains, this routine also computes and stores with each edge, the conditions on role-counts under which each of the action branches will be taken.

Well-defined Executions We already ensured that all loops must terminate. Alg. 2 may still have led to ill-defined executions in steps 4 and 9. Step 4 assimilates a trace edge with an existing edge in the plan and step 9 can merge s_{i+1} into an existing node in the plan. In either case, the next action a_{i+1} *from the new trace*, will be applied to a structure represented by *an existing node in the plan*. The embeddability tests in steps 3 and 8 ensure that a_{i+1} will be applicable to this structure. Step 10 ensures that all possible outcomes of each action edge are stored in the plan. Thus:

Theorem 3. *If the plan input to Alg. 2 has well-defined executions, then its output also has well-defined executions.*

Although other algorithms have been previously developed to merge plans together, to our knowledge Alg. 2 is the first that maintains the property of well-defined executions in the presence of loops. In addition, in comparison to our prior work (Srivastava, Immerman, and Zilberstein 2010b), this algorithm is much more efficient: it also considers the addition of loop edges within the new trace, rather than only between the trace and the previous version of the generalized plan. This reduces the number of examples required for complete solutions by up to 50% in our experiments.

3.3 Optimizations

The subroutines used in Alg. 1 and Alg. 2 can be further optimized. We implemented two optimizations to minimize planner and model-generator invocations. First, we maintain a data structure with all the plans generated, indexed by the concrete instances that they solve. Before generating an instance for an open node, we check if one of the previously generated instances can be embedded in it. If so, we jump to step 9 (tracing) in Alg. 1. This is useful because instance generation, in particular, can be expensive when transitive closure is involved (see Section 4). Second, after every call to ExtendPartialPlan, an attempt is made to merge each open node with a node within the plan that embeds it, while guaranteeing termination using getSafeNodeToJoin. A few other promising optimizations are discussed in Section 5.

3.4 Properties of Generalized Plan Synthesis

We summarize the key properties of Alg. 1 in the form of the following consequences that follow from Theorem 3 and the property of well-defined executions.

Corollary 1. *Generalized plans produced by Alg. 1 have well-defined executions.*

Corollary 2. *If a generalized plan produced by Alg. 1 cannot solve a problem instance represented by its start node's label, its execution will terminate at an open node.*

Theorem 4. *If a generalized plan produced by Alg. 1 has no open nodes, it will solve every instance represented by the label of its start node.*

Reachability of Generated Instances Action update on an abstract structure may result in a structure that embeds a superset of the actually possible concrete results (Sagiv, Reps, and Wilhelm 2002). This over-approximation is neither entirely undesirable nor unintended—an over-approximated state representation can capture future states and is therefore fundamental to the mechanism of recognizing and creating loops. However, open nodes that only

[1] A *simple loops with shortcuts* is a strongly-connected component such that removing one of its nodes makes it acyclic.

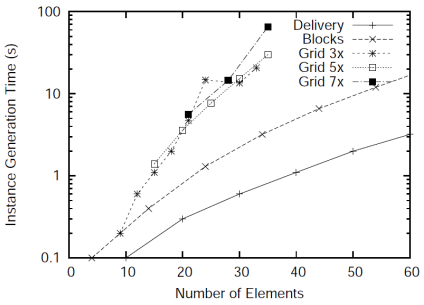

Figure 3: Performance of instance generation

represent concrete states that are unreachable along any existing path in the plan can lead to wasteful model generator and planner invocations. Such open nodes can be pruned using precondition evaluation techniques proposed in prior work (Srivastava, Immerman, and Zilberstein 2010a): if the precondition for reaching a node is not consistent with the initial abstract state, then that node can be removed from the plan. If the preconditions are consistent with the initial abstract state, then they can be used to construct a special concrete instance which will reach the open node and can be solved. Therefore, we get the following result:

Theorem 5. *If preconditions for reaching open nodes can be computed, then the set of problem instances covered by the generalized plan strictly increases with every iteration of the main loop in Alg. 1.*

In prior work (Srivastava, Immerman, and Zilberstein 2010a) we showed that the antecedent of Thm. 5 holds in extended-LL domains for a broad class of plans with loops. Reachable, yet unsolvable open nodes may be produced in domains where action effects cannot be reversed. Identifying an open node as unsolvable requires reasoning abilities beyond the scope of this paper; however, if an open node is known to be unsolvable, and if preconditions can be computed for reaching that node, then an instance of the initial state satisfying those preconditions can be generated. The classical planner solution for this instance can then be added to the generalized plan's start node as the initial open node. In this way, the scope of the generalized plan can be extended to include problem instances that would have led to the unsolvable open node.

4 Implementation and Results

We divide this section into two parts. First, we investigate if our approach for representing abstract structures as first-order formulas can be used in practice for instance generation. Next, we study the overall approach presented in this paper. All experiments were carried out on a 1.6GHz-Intel Core2Duo laptop with 1.5GB of RAM.

Evaluation of Instance Generation We used the first-order model generator Mace4 (`http://www.cs.unm.edu/~mccune/mace4/`) for implementing step 6 in Alg. 1. Mace4 can take as its input the least domain size that it should consider. For tests in this section, we gave it accurate domain sizes. In the overall planning algorithm however this is not always possible, and in its tests we only give Mace4 a lower bound by assuming all summary elements instantiate to a single element (this is often inaccurate).

Fig. 3 shows the times taken to generate instances of different sizes for initial abstract structures from three of the problems considered in the next section. Of these, the delivery problem does not have any transitive closure (TC) properties and the blocks problem has one, asserting that all blocks are in a single tower. The grid problems have multiple TC relations: one for each row, asserting connectivity along the right edges and the TC of up edges. These problems are the hardest to generate instances for, as all the horizontal and vertical TC relationships need to be consistent. The timing results show that as long as we use a limited number of mutually constrained TC relationships, instance generation scales well. For the last points of grid problems our experiments were bottlenecked by system memory. In all cases, generation of first-order formulas from an abstract structure took less than a second.

Overall Approach Our overall implementation is in Python, which is an interpreted language and is not built for performance comparisons. However, we include system-independent metrics for evaluating the algorithm. Any classical planner could be used for solving instances; we used FF (Hoffmann and Nebel 2001) due to its robustness in handling inputs with negative preconditions and quantified goals. Our implementation uses a resource limit of 3000s, but this was never reached in the experiments. The system automatically computes changes in role-counts and branch conditions in terms of role-counts when possible.

We tested the system on open problems in the planning literature (Delivery, Transport, Striped Tower (Srivastava, Immerman, and Zilberstein 2011), Hall-A, GridYx or "Prize-A" (Bonet, Palacios, and Geffner 2009)). All of these problems involve unknown, unbounded numbers of objects. There are currently no approaches capable of computing generalized plans with well-defined executions for any of these problems. In the grid problem, we consider versions with 3, 5, and 6 rows, with unknown, unbounded numbers of squares in each. The goal in these problems is to compute a path for visiting all the grid squares. In this case, classical planners can generate solutions that lack any particular exploration pattern, because ties are broken arbitrarily every time several successor states have the same heuristic value during search. Using a Java version of FF, JavaFF (`http://personal.cis.strath.ac.uk/~ac/JavaFF/`), we made a small modification to always resolve ties using a lexicographic order of actions.

Table 1 shows a summary of the results with the number of planner calls, the largest problem instance and plan generated, applicability status of the resulting solution and the total time taken for computing the generalized plan. In applicability, "T" indicates proven termination, "C" indicates a complete solution, and "P" indicates an automatic proof of completeness (no open nodes). We consider two starting role-counts for each summary element, 1 and 5. The results show that a role-count of 5 is almost always better in terms of the number of planner and model-generator invocations, as well as total time. For Grid6x, we could only increase the initial role-count to 3 before running out of memory.

	Initial Role Count = 1					Initial Role Count = 5^*				
Problem	N_{calls}	$\|\|S\|\|_{max}$	$\|\|\pi\|\|_{max}$	Applicability	T(s)	N_{calls}	$\|\|S\|\|_{max}$	$\|\|\pi\|\|_{max}$	Applicability	T(s)
Delivery	7	**9**	**9**	T, C, P	297	7	14	25	T, C, P	**246**
Grid3x	6	**15**	**12**	T, C, P	166	**3**	21	20	T, C, P	**82**
Grid5x	8	**25**	**22**	T, C, P	631	**4**	35	34	T, C, P	**172**
Grid6x*	10	30	**27**	T, C, P	**1791**	**9**	30	29	T, C, P	1902
Hall-A	7	**10**	**6**	T, C, P	180	**3**	24	18	T, C, P	**70**
Reverse	7	8	**6**	T, C	119	**6**	8	15	**T, C, P**	**128**
Sorting	7	**7**	6	T, C, P	**78**	7	9	6	T, C, P	82
Striped Tower	10	**10**	**11**	T, C, P	831	**5**	14	24	T, C, P	**290**
Y-Transport	9	13	**53**	T, C, P	943	**7**	13	63	T, C, P	**550**

Table 1: Summary of results

A benefit of representing and planning with unknown, unbounded quantities of objects is that we can now model and solve non-trivial programming problems as generalized planning problems. We tested the applicability of this approach on two such problems: reversing a singly linked list (LL) and an online version of sorting. In sorting, given an unbounded, sorted LL and a new element, the problem is to insert the element in the appropriate position, thus returning to the initial abstract state. The resulting solution essentially gives an implementation of insertion sort. We studied two models of LL domains: in sorting, the problem remains interesting even without pointer manipulation and we used high level actions such as *mvPtrFwd(x)* and *insert(x,y)* for moving a pointer and inserting x before y respectively. For reverse, we used lower level programing actions such as `ptr1 = ptr1->next, ptr1->next=ptr2,` and `ptr1=ptr2`. Pointer assignment actions in this domain can be irreversible as data elements can get lost. This makes it difficult for model checkers to even just verify existing LL-manipulation programs. For reverse, when the initial role-count was set to 1, our approach computed the correct program with a loop for reversing the list pointers, but also generated two unreachable and unsolvable nodes. The computed program is guaranteed to terminate and forms a complete solution, but a proof of completeness requires the pruning of these unreachable nodes which is beyond the scope of this paper. In all other problems we obtained generalized plans with loops with provably complete coverage and termination.

5 Conclusions and Future Work

We presented an approach for planning with unknown and unbounded quantities of objects. It is well understood that this problem is unsolvable in general. Even for the problem classes that we solved, no other approach can find generalized plans while guaranteeing the property of well-defined executions. We also showed how unbounded quantities of objects and relationships between them can be represented compactly using transitive closure in many interesting planning problems. These properties make our approach uniquely qualified for settings where plans are intended to be applied on many different problem instances.

The direction of work that comes closest to our approach (Hu and Levesque 2010) requires stronger conditions for asserting correctness: only 1 numeric parameter of variance in the initial state, with only decrementing actions on that parameter. Our own prior work only addressed the problems of finding simple loops in a given classical plan (Srivastava, Immerman, and Zilberstein 2011), and developed a less efficient method for merging *user-generated* plans (Srivastava, Immerman, and Zilberstein 2010b). These methods don't maintain the property of well-defined executions. The approach presented here extends these approaches and introduces methods for axiomatization and instantiation from abstract structures as well as for using classical planners to extend generalized plans with well-defined executions. The resulting approach can compute well-defined, safe solutions to algorithm-design problems. This suggests a great potential for solving non-trivial programming problems while utilizing the heuristic search capabilities of modern classical planners.

The proposed approach can be extended functionally, through methods for pruning open-nodes, as well as qualitatively, through improvements in its subroutines. For example, loop detection could search for loops that make the most progress towards a given goal. The approach is also suitable for generalized planning in non-deterministic environments with sensing actions, but may be more sensitive to order of open node resolution in that case. The extension of the general approach to such domains is left for future work.

Acknowledgments

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

References

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. of the 19th International Conference on Automated Planning and Scheduling*, 34–41.

Emerson, E. A., and Sistla, A. P. 1996. Symmetry and model checking. *Formal Methods in System Design* 9:105–131.

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

Hu, Y., and Levesque, H. J. 2010. A correctness result for reasoning about one-dimensional planning problems. In

Proc. of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning.

Lev-Ami, T.; Immerman, N.; Reps, T.; Sagiv, M.; Srivastava, S.; and Yorsh, G. 2009. Simulating reachability using first-order logic with applications to verification of linked data structures. *Logical Methods in Computer Science* 5.

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* 5:39–70.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2010a. Computing applicability conditions for plans with loops. In *Proc. of the 20th International Conference on Automated Planning and Scheduling*, 161–168.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2010b. Merging example plans into generalized plans for non-deterministic environments. In *Proc. of the 9th International Conference on Autonomous Agents and Multiagent Systems*, 1341–1348.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. In *Artificial Intelligence*, volume 175:2, 615–647.

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