Using Abstraction for Generalized Planning

Siddharth Srivastava and **Neil Immerman** and **Shlomo Zilberstein**
Department of Computer Science,
University of Massachusetts,
Amherst, MA 01002

Abstract

Given the complexity of planning, it is often beneficial to create plans that work for a wide class of problems. This facilitates reuse of existing plans for different instances of the same problem or even for other problems that are somehow similar. We present a novel approach for finding such plans using state representation and abstraction techniques originally developed for static analysis of programs. Our algorithm takes as input a finite 3-valued first-order structure representing a set of initial states and a goal test. The output is a set of *generalized plans* and conditions describing the problem instances for which they work. These plans–which could include loops and are thus close to algorithms–work for a large class of problem scenarios having varying numbers of objects that must be manipulated to reach the goal. We show how to use this framework to learn a general plan from examples.

Introduction

Interest in computing plan generalizations in AI is perhaps as old as planning itself. Attempts at producing plans for more than a single problem instance started with Fikes *et al.* [1972]. Their framework parametrized and indexed subsequences of existing plans for use as macro operations or alternatives to failed actions. However, this approach turned out to be quite limited and prone to over-generalization. This initial effort was followed by various approaches to plan reuse: *case based planning* [Hammond, 1996], *analogical reasoning* [Veloso, 1994] and more recently, domain-specific planning templates [Winner & Veloso, 2002, 2003] which is probably the closest in spirit to our approach. Winner & Veloso [2003] provide an approach for extracting generalized domain-specific plans (*dsPlans*) from examples. A dsPlan is composed of programming constructs like branches and loops. Branches are used to merge new example plans with the existing plans. The ability for extracting loops however is limited and the proposed algorithm can only detect loops with a single action. DISTILL has been shown to extract a dsPlan for all blocks world problems consisting of 2 blocks from 6 chosen example plans. In contrast, our work aims to find general plans that would work, for example, on instances of a particular blocks world problem with different (and unbounded) numbers of blocks.

We present a promising approach for finding plans for sets of problem instances from a domain. These problem instances could not only differ in the number of elements which need to be manipulated for reaching the goal, but also have no bounds on these numbers. We accomplish this by including loops over such objects in our plans. Our plans are thus closer to algorithms: our input represents an abstract planning problem and the generalized plans we produce solve it for a range of problem instances. Instead of reusing or generalizing given plans, at the very outset we search for a general plan.

Our approach uses *state aggregation*, which has been extensively studied for efficiently representing universal plans [Cimatti *et al.*, 1998], solving MDPs [Hoey *et al.*, 1999; Feng & Hansen, 2002], for producing heuristics and for hierarchical search [Knoblock, 1991]. Unlike these techniques that only aggregate states within a single problem instance, we use an abstraction that aggregates states from different problem instances with different numbers of objects. This abstraction scheme has been used effectively in static analysis of programs [Sagiv *et al.*, 2002].

The main contribution of the paper is a new framework for generalized planning, a novel technique for accomplishing this using an existing abstraction scheme and a precise analytical charaterization of the domains (*extended-LL* domains) where it currently works. The rest of this paper is organized as follows. The next section presents a high-level overview of our technique and the abstraction mechanism it employs, illustrated with a simple blocks world example. This is followed by a formal description of the abstraction methodology. The next section presents the requirements we impose on this general method thus identifying a useful category of domains that we handle. We then present our planning algorithm and illustrate how it works. We also show an interesting and surprisingly straightforward application of our technique which allows us to learn a general plan from examples.

Overview of the Approach

Our approach is based on *canonical abstraction* that has been used effectively in the Three-Valued Logic Analyzer (TVLA) – a tool for static analysis of programs that manipulate pointers [Sagiv *et al.*, 2002; Loginov *et al.*, 2006]. Canonical abstraction groups together any objects that are the same with respect to certain key properties: unary predicates referred to as *abstraction predicates*. The values of all the abstraction predicates on an object of the domain define the *role* that it plays. Abstract states are then generated by merging all objects in a role into a single summary element. For example, Fig.1 shows the abstraction predicates, roles,

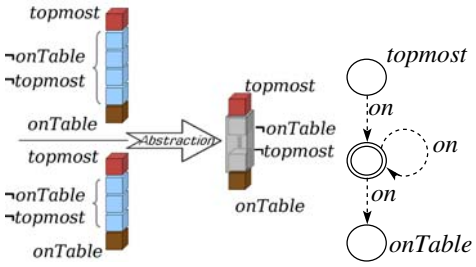

Figure 1: Canonical abstraction in blocks world. Abstracted objects are encapsulated. The abstraction predicates are $topmost$ and $onTable$; diagram on the right shows the state in TVLA notation.

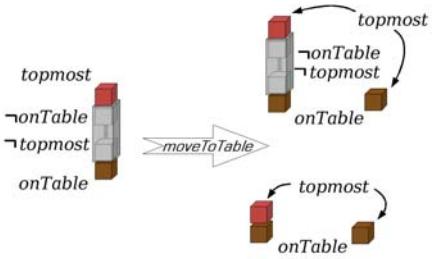

Figure 2: Branching. In blocks world the focus operation models "drawing" an object from a role. In doing so it produces a branch on the number of objects left in the $\neg topmost \wedge \neg onTable$ role.

and an abstract representation for a blocks world domain.

We need a sound methodology for modeling actions with such abstract representations. This is done in TVLA using action transformers specified in first-order logic with transitive closure, discussed in detail in the next section. The most interesting part of this methodology is the automatic modeling of branching when an action depletes the number of objects of a role. Since we abstracted away the true numbers, our model must reflect the possibility that the object removed from a role could be the last one playing that role. Fig.2 shows an example of this situation in the blocks world where the focus operation splits the abstract structure into two relevant cases.

Our overall methodology uses the abstraction mechanism described above to construct an abstract state space. The abstract start state represents a set of concrete states from problem instances with varying numbers of objects. We perform a search in the abstract state space using the action model described. Typically, back edges and loops are encountered. Unlike a search in the concrete state space, not all loops encountered here are stagnant – on the contrary, part of our goal is to identify paths with loops that make progress and lead to the goal. Once such a path is found, we find the preconditions on the concrete states for which it will work, and annotate the start structure to reflect the availability of this partial solution. We then repeat this process until either the entire abstract start structure has been covered, or all the interesting paths have been analyzed. In this paper, we only consider paths with simple, i.e, non-nested loops.

Framework for Canonical Abstraction

We represent states of a domain using logical structures. A structure, S, of vocabulary $\mathcal{V}$, consists of a universe $|S| = \mathcal{U}$ along with a relation R^S over $\mathcal{U}$ for every relation symbol R in $\mathcal{V}$ and an element $c^S \in \mathcal{U}$ for every constant symbol in $\mathcal{V}$. We use $[\![\varphi]\!]^S$ to denote the truth value of a closed formula φ in the structure S.

Example 1 A typical blocks world vocabulary would consist of a binary relation on; this can be used to define other relations like $onTable$ and $topmost$ using first-order formulas. For clarity in presentation however, we will treat all of these relations as separate and equally fundamental. An example structure, S, in this vocabulary can be described as: $|S| = \{b_1, b_2, b_3\}$, $onTable^S = \{b_3\}$, $topmost^S = \{b_1\}$, $on^S = \{(b_1, b_2), (b_2, b_3)\}$.

We assume actions to be deterministic. The *action transformer* for an action a (written τ_a) consists of a set of preconditions and a set of formulas defining the new value r' of each relation r. We separate these two parts of an action transformer: the argument selection and precondition checks are done in a pre-action step. For instance, for the $move$ action, the pre-action steps set up predicates identifying the object to be moved and its destination. For the update, these predicates are used to bind the two variables obj_1 and obj_2 to the block to be moved, and its destination, respectively. Preconditions of actions may enforce *integrity constraints*, for example, the precondition for $move$ could be $topmost(\mathrm{obj}_1) \wedge topmost(\mathrm{obj}_2) \wedge \mathrm{obj}_1 \neq \mathrm{obj}_2$ ensuring that the relation on remains 1:1 and irreflexive.

We write $\tau_a(S)$ to denote the structure obtained by applying action a to structure S. Let, $\tau_a(\Gamma) = \{\tau_a(S) \mid S \in \Gamma\}$ be the application of a to a set of structures, Γ.

Let Δ_i^+ (Δ_i^-) be formulas representing the conditions under which the relation $r_i(\bar{x})$ will be changed to true (false) by a certain action. The formula for r_i', the new value of r_i, is written in terms of the old values of all the relations:

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

The RHS of this equation consists of two clauses, the first of which holds for arguments on which r_i is changed to true by the action; the second clause holds for arguments on which r_i was already true, and remains so after the action.

Example 2 In the blocks world, 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:

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

The goal condition is a represented as a first-order formula; given a start structure, the objective of planning is to reach a structure that satisfies the goal condition. With this notation, we define a domain-schema as follows:

Definition 1 A *domain-schema* is a tuple $\mathscr{D} = (\mathcal{V}, \mathcal{A}, \varphi_g)$ where $\mathcal{V}$ is a vocabulary, $\mathcal{A}$ a set of action transformers, and φ_g a first-order formula representing the goal condition.

Some special unary predicates are classified as *abstraction predicates*. The special status of these predicates arises from the fact that when we perform abstraction, unlike other predicates, these predicates do not become ambiguous. We

define the *role* an element plays as the set of abstraction predicates it satisfies:

Definition 2 A *role* is a conjunction of literals consisting of each abstraction predicate or its negation.

Example 3 In the blocks world, with abstraction predicates $topmost$ and $onTable$, the role $\neg topmost \wedge \neg onTable$ designates blocks that are in the middle of a stack.

Canonical Abstraction

Canonical abstraction [Sagiv *et al.*, 2002] groups states together by only counting the number of elements of a role up to two. If a role contains more than one element, they are combined into a *summary element*. We can tune the choice of abstraction predicates so that abstract structures effectively model some interesting general planning problems and yet the size and number of abstract structures remains manageable.

The imprecision that must result when states are merged is modeled using three-value logic. In a three-valued structure the possible truth values are $0, \frac{1}{2}, 1$, where $\frac{1}{2}$ means "don't know". If we order these values as $0 < \frac{1}{2} < 1$, then conjunction evaluates to minimum, and disjunction evaluates to maximum. See Fig.1 where on holds between the topmost block, e_1, and some but not all of the blocks of the summary element, e_2. Thus the truth value of $on(e_1, e_2)$ is $\frac{1}{2}$, drawn in TVLA as a dotted arc.

We next define *embeddings* [Sagiv *et al.*, 2002]. Define the *information order* on the set of truth values as $0 \prec \frac{1}{2}, 1 \prec \frac{1}{2}$, so lower values are more precise. Intuitively, S_1 is embedabble in S_2 if S_2 is a correct but perhaps less precise representation of S_1. In the embedding, several elements of S_1 may be mapped to a single summary element in S_2.

Definition 3 Let S_1 and S_2 be two structures and $f : |S_1| \rightarrow |S_2|$ be a surjective function. f is an *embedding* from S_1 to S_2 ($S_1 \sqsubseteq^f S_2$) iff for all relation symbols p of arity k and elements, $u_1, \ldots, u_k \in |S_1|$, $[\![p(u_1, \ldots, u_k)]\!]^{S_1} \prec [\![p(f(u_1), \ldots, f(u_k))]\!]^{S_2}$.

The universe of the canonical abstraction, S', of structure, S, is the set of nonempty roles of S:

Definition 4 The embedding of S into its *canonical abstraction* wrt the set A of abstraction predicates is the map:

$$c(u) = e_{\{p \in A | [\![p(x)]\!]^{S,u/x}=1\}, \{p \in A | [\![p(x)]\!]^{S,u/x}=0\}}$$

Further, for any relation r, we have $[\![r(e_1, \ldots, e_k)]\!]^{S'} = l.u.b_{\preceq}\{[\![r(u_1, \ldots, u_k)]\!]^S | c(u_1) = e_1, \ldots, c(u_k) = e_k\}$.

Note that the subscript $\{p \in A | [\![p(x)]\!]^{S,u/x} = 1\}, \{p \in A | [\![p(x)]\!]^{S,u/x} = 0\}$ captures the role of the element u in S. Thus canonical abstraction merges all elements that have the same role, see Fig.1. The truth values in canonical abstractions are as precise as possible: if all embedded elements have the same truth value then this truth value is preserved, otherwise we must use $\frac{1}{2}$. The set of concrete structures that can be embedded in an abstract structure S is called the *concretization* of S: $\gamma(S) = \{S' | \exists f : S' \sqsubseteq^f S\}$.

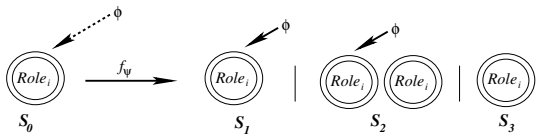

Figure 3: Effect of focus with respect to φ.

Focus With such an abstraction, the update formulas for actions might evaluate to $\frac{1}{2}$. We therefore need an effective method for applying action transformers while not losing too much precision. This is handled in TVLA using the *focus* operation. The focus operation on a three-valued structure S with respect to a formula φ produces a set of structures which have definite truth values for every possible instantiation of variables in φ, while collectively representing the same set of concrete structures, $\gamma(S)$. A focus operation with a formula with one free variable is illustrated in Fig.3: if $\phi()$ evaluates to $\frac{1}{2}$ on a summary element, e, then either all of e statisfies ϕ, or part of it does and part of it doesn't, or none of it does. This process could produce structures that are inherently infeasible. Such structures are either refined or discarded using a set of restricted first-order formulas called *integrity constraints*. In Fig.3 for instance, if integrity constraints restricted ϕ to be unique and satisfiable, then structure S_3 in Fig.3 would be discarded and the summary elements for which $\phi()$ holds in S_1 and S_2 would be replaced by singletons.

The focus operation wrt a set of formulas works by successive focusing wrt each formula in turn. The result of the focus operation on S wrt a set of formulas Φ is written $f_\Phi(S)$. We use ψ_a to denote the set of focus formulas for action a.

Choosing Action Arguments

Usually, action specifications are allowed to have free variables. During a typical TVLA execution, such an action is tried with every binding of the free variables that satisfies the pre-conditions. In static analysis this feature can be used to model non-determinism. We choose the arguments in a series of pre-action focus steps. For example, to choose obj_1, in the $move$ action we would focus on an auxiliary unary predicate $\mathrm{obj}'_1()$ that is constrained to be single-valued and to imply $topmost$.

Action Application

Recall that the predicate update formulas for action transformer take the form shown in equation 1. This equation might evaluate to indefinite truth values in abstract structures. For our purposes, the most important updates are for abstraction predicates since precision in their values determines the accuracy of modeling action dynamics. In this special case, the expressions for Δ_i^+ and Δ_i^- are *monadic* (i.e. have only one free variable apart from the action arguments which are bound by the pre-action steps).

In order to obtain definite truth values for these updates, we focus the given abstract structure S on the expressions $\Delta_i^\pm$. Once the action transformer has been applied, we again apply canonical abstraction (this is called "blur" in TVLA) on the resulting structures to get the abstract result structures.

Input: S_0, $\mathscr{D} = (\mathcal{A}, \varphi_g)$

1 $S \leftarrow S_0$
while $S \not\models \varphi_g$ **do**
2 Nondeterministically execute action $a \in \mathcal{A}$ on S to get S'
3 $S \leftarrow S'$
end

Algorithm 1: NDPlan

Transitions

Once the action arguments have been chosen, there are three steps involved in action application: action specific focus, action transformation, and blur. The transition relation $\xrightarrow{a}$ for each action a, captures the combined effect of these three steps. More precisely,

Definition 5 *(Transition Relation)* $S_1 \xrightarrow{a} S_2$ iff S_1 and S_2 are three-valued structures and there exists a focused structure $S_1^1 \in f_{\psi_a}(S_1)$ s.t. $S_2 = blur(\tau_a(S_1^1))$.

Sometimes however we will need to study the exact path S_1 took in getting to S_2. For this, the transition $S_1 \xrightarrow{a} S_2$ can be decomposed into a set of transition sequences $\{(S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2) | S_1^i \in f_{\psi_a}(S_1) \wedge S_2^i = \tau_a(S_1^i) \wedge S_2 = blur(S_2^i)\}$.

TVLA can effectively generate the full instantiation of these transition relations for a given domain in the form of a *transition graph*, the subject of our next section.

Transition Graphs Transition graphs are graphs of all the transition relations $\xrightarrow{a}$ with nodes representing structures. Formally,

Definition 6 Given a domain $\mathscr{D}$ and an initial state S_0, the *transition graph for $\mathscr{D}$ starting with S_0*, $\mathscr{G}_{\mathscr{D}}(S_0)$ is the edge-labeled multigraph defined by the set of relations $\{\xrightarrow{a}: a \in \mathcal{A}\}$ on the set of structures reachable from S_0. Each edge of $\mathscr{G}_{\mathscr{D}}(S_0)$ is labeled by the appropriate action.

We omit the subscript $\mathscr{D}$ where it is understood from context. The entire transition graph of a domain, though finite, is usually an unwieldy structure. Its generation can be optimized using various heuristics; owing to abstraction, this in itself could be an interesting subject for research. In this paper however, our focus is on using the transition graph rather than its efficient generation. The algorithm we propose for generating plans allows for the dynamic generation of transition graphs; it can also be used in an on-line fashion by interleaving the search and transition graph generation phase with the analysis phase for iteratively obtaining partial solutions. We discuss this issue further in the Algorithm section. For now, we present one very simple algorithm to show that this graph can indeed be effectively generated.

Algorithm 1 shows NDPlan, a non-deterministic algorithm for finding plans. We use NDPlan as a program input to TVLA for generating $\mathscr{G}(S_0)$. TVLA collects *all* the structures that could possibly result from the execution of a program statement at the next node in the program's control flow graph (CFG). Since the non-deterministic action execution could produce the effects of any of the actions in $\mathcal{A}$, TVLA produces all of those structures at the CFG node corresponding to statement #3. Running TVLA on this plan therefore has the same effect as a BFS search for a goal state, except that the search continues until all reachable structures have been generated. In order to speed up this process, it is possible to restrict its behavior so as to not explore actions on structures which are easily seen to be on the wrong track.

Our Methodology

The idea behind our algorithm for generalized planning is to successively find paths in a transition graph and for each such path, to compute the pre-conditions under which it takes concrete structures to structures satisfying the goal.

In order to accomplish this, we need a way of representing subsets of abstract structures that are guaranteed to take a particular branch of an action's focus operation. Next, we need to propagate these subsets backwards through action edges in the given path all the way up to the start structure.

We represent subsets of an abstract structure by annotating a structure with a set of conditions from a chosen constraint language. In static analysis terms, we use annotations to *refine* our abstraction. Formally,

Definition 7 *(Annotated Structures)* Let $\mathcal{C}$ be a language for expressing constraints on three-valued structures. A *$\mathcal{C}-$annotated structure* $S|_C$ is a refinement of S consisting of structures in $\gamma(S)$ that satisfy the condition $C \in \mathcal{C}$. Formally, $\gamma(S|_C) = \{s \in \gamma(S) \mid s \models C\}$.

We extend the notation defined above to sets of structures, so that if Γ is a set of structures then by $\Gamma|_C$ we mean the structures in Γ that satisfy C. Consequently, we have $\gamma(S|_C) = \gamma(S)|_C$.

We now need to be able to identify the (annotated) pre-image of an annotated structure under any action. This requirement on a domain is captured by the following definition:

Definition 8 *(Annotated Domains)* An *annotated domain-schema* is a pair $\langle \mathscr{D}, \mathcal{C} \rangle$ where $\mathscr{D}$ is a domain-schema and $\mathcal{C}$ is a constraint language. An annotated domain-schema is *amenable to back-propagation* if for every transition $S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2$ and $C_2 \in \mathcal{C}$ there exists $C_1^i \in \mathcal{C}$ such that $\tau_a(\gamma(S_1)|_{C_1^i}) = \tau_a(\gamma(S_1^i))|_{C_2}$.

In terms of this definition, since $\tau_a(\gamma(S_1^i))$ is the subset of $\gamma(S_2^i)$ that has pre-images in S_1^i under τ_a, $S_1|_{C_1^i}$ is the pre-image of $S_2|_{C_2}$ under a particular focused branch (the one using S_1^i) of action a. The disjunction of C_1^i over all branches taking S_1 into S_2 therefore gives us a more general annotation which is not restricted to a particular branch of the action update. Therefore, we have:

Lemma 1 *Suppose $\langle \mathscr{D}, \mathcal{C} \rangle$ is an annotated domain-schema that is amenable to back-propagation and $S_1, S_2 \in \mathscr{D}$ such that $S_1 \xrightarrow{a} S_2$. Then for all $C_2 \in \mathcal{C}$ there exists a $C_1 \in \mathcal{C}$ such that $\tau_a(\gamma(S_1)|_{C_1}) = (\tau_a(\gamma(S_1)) \cap \gamma(S_2))|_{C_2}$.*

This lemma can be easily generalized through induction to multi-step back-propagation along a sequence of actions. We use the abbreviation $\tau_{k\ldots1}$ to represent the successive application of action transformers a_1 through a_k, in this order.

Proposition 1 (Linear backup) *Suppose $\langle \mathscr{D}, \mathcal{C} \rangle$ is an annotated domain-schema that is amenable to back-propagation and $S_1, \ldots, S_k \in \mathscr{D}$ are distinct structures such that $S_1 \xrightarrow{\tau_1}$*

$S_2 \cdots \xrightarrow{\tau_{k-1}} S_k$. *Then for all* C_k *there exists* C_1 *such that* $\tau_{k-1\ldots1}(\gamma(S_1)|_{C_1}) = \tau_{k-1\ldots1}(\gamma(S_1)) \cap S_k|_{C_k}$

The restriction of distinctness in this proposition confines its application to action sequences without loops.

Amenability for back-propagation can be seen as being composed of two different requirements. In terms of definition 8, we first need to translate the constraint C_2 into a constraint on S_1^i. Next, we need a constraint selecting the structures in S_1 that are actually in S_1^i. Composition of these two constraints will give us the desired annotation, C_1^i. The second part of this annotation, classification, turns out to be a strong restriction on the focus operation. We formalize it as follows:

Definition 9 *(Focus Classifiability)* A focus operation f_ψ on a structure S satisfies *focus classifiability with respect to a constraint language* $\mathcal{C}$ if for every $S_i \in f_\psi(S)$ there exists an annotation $C_i \in \mathcal{C}$ such that $s \in \gamma(S_i)$ iff $s \in \gamma(S|_{C_i})$.

Since three-valued structures resulting from focus operations with unary formulas are necessarily non-intersecting, we have that any $s \in S$ may satisfy at most one of the conditions C_i. Focus classifiability is hard to achieve in general; we need it only for structures reachable from the start structure.

Inequality-Annotated domain-schemas Let us denote by $\#_R(S)$ the number of elements of role R in structure S. In this paper, we use $\mathcal{C}_I(\mathcal{R})$, the language consisting of constraints expressed as sets of linear inequalities using $\#_{R_i}(S)$ for annotations. In order to make domain-schemas annotated with constraints in $\mathcal{C}_I(\mathcal{R})$ amenable to back propagation, we first restrict the class of permissible actions to those that change role counts by a fixed amount:

Definition 10 *(Uniform Change)* An action shows *uniform change w.r.t a set of roles* $\mathcal{R}$ iff whenever $S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{a} S_2^i \xrightarrow{b} S_2$ then $\forall s_1 \in \gamma(S_1^i), s_2 \in \gamma(S_2^i)$ such that $s_1 \xrightarrow{a} s_2$, we have $\#_{R_i}(s_2) = g(\#_{R_1}(s_1), \ldots \#_{R_l}(s_1))$ where g is a linear function determined by S_1^i and S_2^i, and $R_1 \ldots R_l \in \mathcal{R}$. In the special case where changes in the counts of all roles are constants, we say the action shows *constant change*.

The next theorem shows that uniform change gives us back-propagation if we already have focus-classifiability.

Theorem 1 (Back-propagation in inequality-annotated domains) *An inequality-annotated domain-schema whose actions show uniform change wrt* $\mathcal{R}$ *and focus operations satisfy focus classifiability wrt* $\mathcal{C}_I(\mathcal{R})$ *is amenable to back propagation.*

PROOF Suppose $S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{a} S_2^i \xrightarrow{b} S_2$ and C_2 is a set of constraints on S_2. We need to show that there exists a C_1^i such that $\tau_a(\gamma(S_1)|_{C_1^i}) = \tau_a(\gamma(S_1^i))|_{C_2}$. Since we are given focus-classifiability, we have C_i such that $s \in \gamma(S_1)|_{C_i}$ iff $s \in \gamma(S_1^i)$. We need to compose C_i with an annotation for reaching $S_2^i|_{C_2}$ to obtain C_1^i. This can be done by rewriting C_2's inequalities in terms of counts in S_1 (counts don't change during the focus operation from S_1 to S_1^i).

Suppose $\#_{R_j}(S_2^i) = g_j(\#_{R_1}(S_1), \ldots \#_{R_l}(S_1))$. Then we obtain the corresponding inequalities for S_1 by substituting $g_j(\#_{R_1}(S_1), \ldots \#_{R_l}(S_1))$ for $\#_{R_j}(S_2^i)$ in all inequalities of C_2. Naturally, if the resulting inequalities are satisfied in S_1, C_2 will be satisfied in S_2^i. The resulting set of inequalities which we will call C_1, gives us the rest of the desired inequalities. The conjunction of C_1 and C_i gives us the desired annotation C_1^i. That is, $\tau_a(\gamma(S_1)|_{C_1^i}) = \tau_a(\gamma(S_1^i)|_{C_1}) = \tau_a(\gamma(S_1^i))|_{C_2}$. □

The next theorem provides a simple condition under which constant change holds.

Theorem 2 (Constant change) *Let* a *be an action whose predicate update formulas take the form shown in Eq. 1. Action* a *shows constant change if for every abstraction predicate* p_i*, all the expressions* Δ_i^+, Δ_i^- *are at most uniquely satisfiable.*

PROOF Suppose $S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2$; $s_1 \in \gamma(S_1^i)$ and $s_1 \xrightarrow{\tau_a} s_2 \in \gamma(S_2^i)$.

For constant change we need to show that $\#_{R_i}(s_2) = \#_{R_i}(s_1) + \delta$ where δ is a constant. Recall that a role is a conjunction of abstraction predicates or their negations. Further, because the focus formula f_{ψ_a} consists of pairs of formulas Δ_i^+ and Δ_i^- for every abstraction predicate, and these formulas are at most uniquely satisfiable, each abstraction predicate changes on at most 1 element. The focused structure S_1^i shows exactly which elements undergo change, and the roles that they leave or will enter.

Therefore, since s_1 is embeddable in S_1^i and embeddings are surjective, the number of elements leaving or entering a role in s_1 is the number of those singletons which enter or leave it in S_1^i. Hence, this number is the same for every $s_1 \in \gamma(S_1^i)$, and is a constant determined by S_1^i and the focus formulas. □

Quality of Abstraction In order for us to be able to classify the effects of focus operations, we need to impose some quality-restrictions on the abstraction. Our main requirement is that the changes in abstraction predicates should be characterized by roles: given a structure, an action can change a certain abstraction predicate only for objects with a certain role. We formalize this property as follows: a formula $\varphi(x)$ is said to be *role-specific in* S iff only objects of a certain role can satisfy φ in S. That is, there exists a role R such that for every $s \in \gamma(S)$, if $(s, [c/x]) \models \varphi(x)$ then $(s, c/x) \models R(x)$.

We therefore want our abstraction to be rich enough to make the action change formulas, $\Delta_i^{\pm}$, role-specific in every structure encountered. For example, in the blocks world state shown in Fig.2 the $move$ action can only change the $topmost$ predicate for a block of the role $\neg topmost \wedge \neg onTable$, representing blocks in the middle of the stack. The design of a problem representation and in particular, the choice of abstraction predicates therefore needs to carefully balance the competing needs of tractability in the transition graph and the precision required for back propagation.

The following proposition formalizes this in the form of sufficient conditions for focus-classifiability. Note that focus classifiability holds vacuously if the result of the focus operation is a single structure (e.g., when the focus formula is unsatisfiable in all $s \in \gamma(S)$, thus resulting in just one structure with the formula false for every element).

Proposition 2 *If ψ is unique, satisfiable in all $s \in \gamma(S)$ and locally role-specific for S then the focus operation f_ψ on S satisfies focus classifiability w.r.t $\mathcal{C}_I(\mathcal{R})$.*

PROOF Since the focus formula must hold for exactly one element of a certain role, the only branching possible is that caused due to different numbers of elements (zero vs. one or more) satisfying the role while not satisfying the focus formula (see Fig.3). The branch is thus classifiable on the basis of the count of the number of elements in the role. □

Corollary 1 *If Φ is a set of uniquely satisfiable and role-specific formulas for S such that any pair of formulas in Φ is either exclusive or equivalent, then the focus operation f_Φ on S satisfies focus classifiability.*

Therefore any domain-schema whose actions only require the kind of focus formulas specified by the corollary above is amenable to back-propagation. Because of the similarity of such domain-schemas with linked-lists, we call them extended-LL domains:

Definition 11 *(Extended-LL domains)* An *Extended-LL domain with start structure S_{start}* is a domain-schema such that Δ_i^+ and Δ_i^- are role-specific, exclusive when not equivalent, and uniquely satisfiable in every structure reachable from S_{start}. More formally, if $S_{start} \rightarrow^* S$ then $\forall i, j, \forall e, e' \in \{+, -\}$ we have Δ_i^e role-specific and either $\Delta_i^e \equiv \Delta_j^{e'}$ or $\Delta_i^e \implies \neg\Delta_j^{e'}$ in S.

Corollary 2 *Extended-LL Domains are amenable to back-propagation.*

Intuitively, these domain-schemas are those where:

1. The information captured by roles is sufficient to determine whether or not an object of any role will undergo change due to an action; and
2. The number of objects being acquired or relinquished by any role is fixed (constant) for each action.

Examples of such domains are linked lists, blocks-world scenarios (the appropriate start structures are defined in the section on Examples), assembly domains where different objects can be constructed from constituent objects of different roles etc.

Handling Paths with Loops

So far we dealt exclusively with propagating annotations back through a linear sequence of actions. In this section we show that in extended-LL domains we can effectively propagate annotations back through paths consisting of simple (non-nested) loops.

Let us consider the path from S to S_f including the loop in Fig.4; analysis of other paths including the loop is similar. The following proposition formalizes back-propagation in loops:

Proposition 3 (Back-propagation through loops) *Suppose $S_0 \xrightarrow{\tau_1} S_1 \xrightarrow{\tau_2} \ldots \xrightarrow{\tau_{n-1}} S_{n-1} \xrightarrow{\tau_0} S_0$ is a loop in an extended-LL domain with a start structure S_{start}. Let the loop's entry and exit structures be S and S_f (Fig.4). We can then compute an annotation $C(l)$ on S which selects the structures that will be in $S_f|_{C_f}$ after l iterations of the loop on S, plus the simple path from S to S_f.*

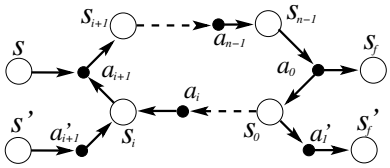

Figure 4: Paths with a simple loop. Outlined nodes represent structures and filled nodes represent actions.

PROOF Our task is to find an annotation for the structure S which allows us to reach $S_f|_{C_f}$, where C_f is a given annotation. Since we are in an extended-LL domain, every action satisfies constant change. Let the individual changes due to action a_i on role count $\#_{R_j}(S)$ be denoted as δ_j^i. In general, this change depends on the structure on which the action is applied and the exact focused branch taken. For clarity in presentation we omit these dependencies in the notation and assume different indices correspond to different actions. We use additional notation when dealing with action a_0: its two branches are characterized by changes δ_j^0 and $\delta_j^{0_f}$. Our convention will be to add δ_j^i to role $\#_{R_j}(S)$ to obtain the count in $\tau_i(S)$, and subtract δ_j^i from a structure to find the role count before action application. As discussed in Theorem 1, the pre-image of an annotation C under action a can be obtained by replacing roles in every inequality in C in terms of their counterparts before the action execution. In other words, we can write $\tau_i^{-1}(C) = [\#_{R_j}(\tau_i(S)) + \delta_j^i / \#_{R_j}(S)]C$.

We need to start going back from S_f and obtain expressions for annotations of every structure in the loop after l unwindings of the loop. Let us denote the annotation at structure S_j during the l^{th} unwinding of the loop to be $C_j(l)$. $C_{n-1}(1)$ for S_{n-1} in Fig.4 is therefore calculated as $\tau_0^{-1}(C_f) \bigwedge C_{n-1\rightarrow f}$, where $C_{n-1\rightarrow f}$ denotes the focus-classification conditions required for taking the $S_{n-1} \rightarrow S_f$ branch of action a_0.

In general these annotations can be defined inductively as follows:

$$\begin{array}{rcll} C_i(l) & = & \tau_{i+1}^{-1}(C_{i+1}(l)) \wedge C_{i\rightarrow i+1} & \text{if } i < n-1 \\ C_{n-1}(l) & = & \tau_0^{-1}(C_0(l-1)) \wedge C_{n-1\rightarrow 0} & \text{if } l > 1 \\ C_{n-1}(1) & = & \tau_{0_f}^{-1}(C_f) \wedge C_{n-1\rightarrow f} & \end{array}$$

Here, we distinguish between τ_0^{-1} and $\tau_{0_f}^{-1}$: the former represents the $S_{n-1} \rightarrow S_0$ branch of a_0; the latter the $S_{n-1} \rightarrow S_f$ branch.

In the rest of this discussion, we use $\tau_{n_1..n_2}^{-1}$ as an abbreviation for $\tau_{n_1}^{-1} \cdot \tau_{n_1+1}^{-1} \cdots \tau_{n_2}^{-1}$. Expanding the inductive rules once, we get:

$$\begin{array}{rcl} C_{n-2}(1) & = & \tau_{n-1}^{-1} \cdot \tau_{0_f}^{-1}(C_f) \wedge \tau_{n-1}^{-1}(C_{n-1\rightarrow f}) \\ & & \wedge C_{(n-2)\rightarrow n-1} \end{array}$$

τ_{n-1}^{-1} distributes over the expression because the inverse operation amounts to a linear substitution.

Continuing in this way, we get the following expression

for $C_m(1)$ where $m < n-1$:

$$\begin{aligned} C_m(1) &= \tau^{-1}_{m+1..n-1} \cdot \tau^{-1}_{0_f}(C_f) \\ &\wedge \bigwedge_{k=m+1}^{n-1} \tau^{-1}_{m+1..k}(C_{k\to k+1}) \wedge C_{m\to m+1} \end{aligned}$$

where the successor of $n-1$ is treated as 0_f. The first term of this expression is the back-propagation of C_f. The second term is a conjunction of back-propagation of all the focus classification conditions necessary for us to stay in the loop until S_f. In subsequent unwindings of the loop, we will need the $S_{n-1} \to S_0$ branch, and consequently we will treat the successor of $n-1$ as 0 (addition modulo n).

Writing the second term as $C^m_{loop}(1)$, denoting the loop conditions for first unwinding of the loop, we get

$$C_m(1) = \tau^{-1}_{m+1..n-1} \cdot \tau^{-1}_{0_f}(C_f) \wedge C^m_{loop}(1)$$

Using the inductive definitions, and writing the transformation over the entire loop, $\tau^{-1}_{n-1..0}$ (or $\tau^{-1}_{n-1..0_f}$) as τ^{-1} (or τ_f^{-1}), we get the general expression for $C_m(l), l > 1$ as:

$$\begin{aligned} C_m(l) &= \tau^{-1}_{m..0} \cdot \tau^{-1(l-2)} \cdot \tau_f^{-1}(C_f) \\ &\wedge \tau^{-1}_{m..0} \cdot \tau^{-1(l-2)}(C^1_{loop}(1)) \\ &\wedge \tau^{-1}_{m..0} \cdot \tau^{-1(l-3)}(C^1_{loop}(2)) \\ &\vdots \\ &\wedge \tau^{-1}_{m..0}\tau^{-1}(C^1_{loop}(l-2)) \\ &\wedge \tau^{-1}_{m..0}(C^1_{loop}(l-1)) \end{aligned}$$

However, note that the loop conditions $C^1_{loop}(i)$ for $i > 1$ are all the same! Even with this simplification, the annotation described above presents a potentially large set of conditions. However, since the conditions are linear inequalities and the changes in roles are monotone with respect to the number of times the inverse operation is performed, there are essentially four conditions at any S_m for l^{th} unwinding of the loop. These conditions ensure all the other conditions (representing intermediate numbers of loop unwindings) hold:

$$\begin{aligned} C_m(l) &= \tau^{-1}_{m..0} \cdot \tau^{-1(l-2)} \cdot \tau_f^{-1}(C_f) \\ &\wedge \tau^{-1}_{m..0} \cdot \tau^{-1(l-2)}(C^1_{loop}(1)) \\ &\wedge \tau^{-1}_{m..0} \cdot \tau^{-1(l-3)}(C^1_{loop}) \\ &\wedge \tau^{-1}_{m..0}(C^1_{loop}) \end{aligned}$$

And finally, we have the conditions $\#_{R_j}(S) \geq 0$ for all j for any S. □

Algorithm for Generalized Planning

Let the annotated-domain-schema$\langle \mathscr{D}, \mathcal{C}_I \rangle$ be an extended-LL domain with a start structure S_0. Algorithm 2 shows our algorithm for generalized planning. It proceeds in phases of search and annotation on the transition graph. During the search phase it finds a path π from the start structure to structures satisfying the goal condition. The transition graph could also be dynamically generated during this phase. During the annotation phase, it finds the annotation on S_0, C_π for π as described in the previous section. The resulting algorithm can be implemented in an any-time fashion, by outputting plans capturing more and more problem instances as new paths are found in the transition graph.

Input: $\mathscr{G}(S_0)$, $\mathcal{S}_g = \{S : S \in \mathscr{G}(S_0)$ and $S \models \varphi_g\}$, $\preceq$: order on paths in $\mathscr{G}(S_0)$
Output: $\Pi = \{\langle C, \pi\rangle : s \in \gamma(S|_C) \Rightarrow \pi(s) \models \varphi_g\}$

```
Π ← ∅
while ∨{C : ⟨C, π⟩ ∈ Π} ≠ ⊤ and ∃π: path not checked
do
    π ← getNextSmallest(S_0, S_g)
    C_π ← findPrecon(S_0, π, S_g)
    if ∨{C : ⟨C, π⟩ ∈ Π} ⇏ C_π then
        Π ← Π ∪ {⟨C_π, π⟩}
    end
end
```

Algorithm 2: GenPlan

Procedure $getNextSmallest(S_0, \mathscr{S}_g)$ returns the next smallest path with simple loops between S_0 and members of $\mathcal{S}_g$. Procedure $findPrecon(S_0, \pi)$ returns the conditions on S_0 under which the path π takes structures to S_g, as discussed in the previous section.

Together with the definition of extended-LL domains and propositions 3 and 1 Algorithm 2 realizes the following theorem:

Theorem 3 (Generalized planning for extended-LL domains) *For an extended-LL domain with a start structure S_0 it is possible to find plans π_i and annotations C_i such that $\forall s \in \gamma(S|_{C_i})$, π_i takes s to the goal; further, for $s \in \gamma(S) \setminus \gamma(S|_{\vee_i C_i})$, the goal is not reachable via plans with only simple loops in the transition graph $\mathscr{G}(S_0)$.*

Examples

In this section we formulate a problem in the blocks world domain in our framework and show how our algorithm finds a generalized plan. The formulation for linked lists is similar and can be used to find generalized plans for problems like reversing a linked list of unknown length i.e., to synthesize simple programs.

Striped Block Tower

Representation Our example is set in the blocks world domain with red and blue blocks. Given a stack consisting of red blocks below and blue blocks above, the goal is to construct a stack with alternating red and blue blocks above an assigned red, $base$ block with a blue block on top.

Our vocabulary consists of the predicates $\{t[on]^2, on^2, topmost^1, onTable^1, \mathrm{obj}_1^1, \mathrm{obj}_2^1, red^1, blue^1, base^1\}$, where all the unary predicates are abstraction predicates. $t[on]$ is the transitive closure of on, and is used in integrity constraints for the coerce operation. The $\mathrm{obj}_1, \mathrm{obj}_2$ predicates are used to select action arguments before an action is applied. There are two actions: $move()$, which places obj_1 on top of obj_2 and $moveToTable()$, which moves obj_1 to the table.

With this set of abstraction predicates, $\Delta_i^{\pm}$ are not role specific in states with two or more stacks (their bases get merged in the abstraction and the block below a selected topmost block could be either a block on table or a middle block, violating role-specificity). One solution is to change the problem definition so as to have a fixed, finite amount

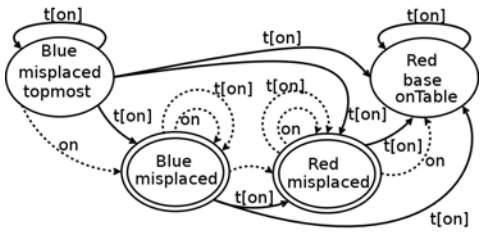

Figure 5: Initial structure for striped blocks world.

$$1 \xrightarrow{mTT} 8 \xrightarrow{mTT} 22 \xrightarrow{select} 42 \xrightarrow{mTT} 21$$

$$21 \xrightarrow{mTT} 19 \xrightarrow{mTT} 46 \xrightarrow{mTT} 70 \xrightarrow{select} 90 \xrightarrow{mTT} 69 \xrightarrow{mTT} 68$$

$$68 \xrightarrow{mv[B]} 93 \xrightarrow{mv[R]} 119 \xrightarrow{mv[B]} 145 \xrightarrow{mv[R]} 170$$

$$170 \xrightarrow{mv[B]} 196 \xrightarrow{select} 216 \xrightarrow{mv[R]} 220 \quad 218 \xrightarrow{mv[R]} 221$$

$$221 \xrightarrow{mv[B]} Goal$$

Figure 6: Generalized plan with three loops found from the transition graph. Numbers represent structure IDs from the transition graph.

of space for stacks on the table. Blocks not on any stack are placed in an unbounded size bag (captured by predicate bag). This will give us an additional set of abstraction predicates, $onStack_i$, holding for every block in the i^{th} stack. This separates stacks in the abstraction and ensures role-specificity for all states reachable from the canonical abstraction of any real state with a bounded number of stacks of unbounded size. For simplicty, we restrict to only one stack in this demonstration.

Input Our initial abstract structure represents all problem instances with a single stack of unknown and unbounded numbers of red and blue blocks (at least two of each). The red blocks are below the blue blocks. Fig.5 shows the initial structure in TVLA notation - relations with truth values $\frac{1}{2}$ are shown using dotted edges and summary elements are shown using double circles. To prune the search space, we add a new abstraction predicate, $misplaced$, and use the following heuristic: a block is misplaced if it is on a block of the same color, or above a misplaced block; a block is only placed on another if it does not get misplaced in the process. This does not effect our main result because using this heuristic reduces the size of the transition graph but not the size of the resulting plan.

Output Fig.6 shows the most interesting branch of the obtained plan. It consists of three loops: the first loop moves all blue blocks to the table; the second, the red blocks, and finally, the last loop moves blue and red blocks alternately back on to the stack. Algorithmically, this plan can be written as shown in Fig.7. The loops in this 20 step plan make it sufficient for infinite problem instances with unbounded numbers of blocks. In the next section, we compute the precise pre-conditions for this plan. Part of these pre-conditions is the interesting fact that for our goal to be reachable, the number of blue blocks has to be equal to the number of red

```
1 Move mispld. blk to table
2 while #(blue blocks in middle) > 1
    Move mispld. blk to table
3 Move mispld. blk to table
4 Move mispld. blk to table
  //All Blue blocks on table
5 Move mispld. blk to table
6 while #(red blocks in middle) > 1
    Move mispld. blk to table
7 Move mispld. blk to table
8 Move mispld. blk to table
  //All blocks on table
9 Move blue blk to stack
10 Move red blk to stack
11 Move blue blk to stack
12 Move red blk to stack
13 Move blue blk to stack
14 while #(blue on table > 1)
     Move red blk to stack
     Move blue blk to stack
15 Move red blk to stack
16 Move blue blk to stack
```

Figure 7: Plan from Fig.6 written out in text.

blocks in the initial structure.

Annotations

In this section we provide an illustration for back-propagation of annotations. We wish to find the pre-conditions at structure 1 for $true$, or no annotation at $Goal$. Let $B_t(R_t)$ be the roles corresponding to $blue(red$, but not $base)$ blocks that are $onTable$ and $topmost$. Similarly, let $B_m(R_m)$ be the roles of $blue(red)$ blocks that are neither $topmost$ nor $onTable$. In these terms, the edge $221 \rightarrow Goal$, is taken if there is exactly one B_t block in 221. Similarly, we exit from the last loop via $196 \rightarrow 218$ if there is only one R_t block in 196. This makes the annotation at 221 as $[R_t = 1, B_t = 1]$. Unrolling this annotation through the loop, we get the the annotations for 196 after l unwindings as: $[B_t(196) = l + 1, R_t(196) = l + 1]$. If $l > 0$, these conditions subsume the conditions required for staying in the loop, which are $B_t > 1$ and $R_t > 1$. Continuing in this way back through the other actions and loops and representing the number of unwindings of "move red to table" ("move blue to table") as $l_r(l_b)$, we get the annotation at structure 1 as $[B_t = l - l_b, R_t = l - l_r - 1, B_m = 3 + l_b, R_m = 4 + l_r]$.

These conditions form the required pre-condition for the given plan to work. Together they imply that the total number of blue blocks, $\#B_t + \#B_m + 1$ ($+1$ for the topmost block in the initial structure) is $4 + l$, exactly equal to the number of red blocks in the initial structure. If we also tally role counts with structures rather than simply propagating them backwards, we obtain the equalities $l - l_b = 0$ and $l - l_r - 1 = 0$ at structure 1. This tells us that number of iterations of the stacking loop is equal to the number of iterations of the unstack-blue-block loop; and further that we needed to unstack one red block less (the $base$).

Cases with fewer number of blocks are captured by other paths (found before this path), and their pre-conditions are obtained similarly. Our algorithm thus finds a generalized plan for solving *infinitely many* instances of the given abstract problem. In this case in fact, we get a solution for all solvable instances. The stacking loop in this example also demonstrates that in comparison to DISTILL, our techniques do not impose any a-priori restrictions on the number of actions in a loop.

Input: $\pi = (a_1, \ldots, a_n)$: plan for $S_0^{\#}$; $S_i^{\#} = a_i(S_{i-1}^{\#})$
Output: Generalized plan Π

$S_0 \leftarrow c(S^{\#})$; $\Pi \leftarrow \pi$; $C_\Pi \leftarrow \top$
Apply transformers for π on S_0 to get S'_{i-1}, S_i s.t. $S'_{i-1} \in f_{a_i}(S_{i-1})$, $\tau_i(S'_{i-1}) = S_i$ and $S_i^{\#} \sqsubseteq S_i$.
if $\exists C \in \mathcal{C}_I(\mathcal{R}) : S_n|_C \models \varphi_g$ **then**
    $\Pi \leftarrow$ formLoops$(S_0 : a_1 \ldots, S_{n-1} : a_n, S_n)$
    $C_\Pi \leftarrow$ findPrecon(S_0, Π, φ_g)
end
return Π, C_Π

Algorithm 3: GeneralizeExample

Generalizing from Examples

In this section we present a simple application of our framework for computing a generalized plan from a plan that works for a single problem instance. The idea behind this technique is that if a given concrete plan contains sufficient unrollings of some simple loops, then we can automatically identify these loops by observing the plan on a suitable abstract structure. We can then enhance this plan using the identified loops and use the techniques discussed in this paper to find the set of problem instances for which this new generalized plan will work. The procedure is shown in Algorithm 3.

Given a concrete example plan π for $S_0^{\#}$, we start with an abstract structure S_0 for the start state. S_0 can be any abstract structure which makes the resulting domain extended-LL, and for which we wish to find the restriction under which a generalization of π will work; the canonical abstraction of $S_0^{\#}$ forms a natural choice. If the final abstract structure S_n or its restriction satisfies the goal, we have a general plan and we proceed to find its pre-conditions.

The $formLoops$ subroutine converts a linear path of structures and actions into a path with simple loops. One way of implementing this routine is by making a single pass over the input path, and adding back edges whenever a structure S_j is found such that $S_i = S_j (i < j)$, and S_i is not part of, or behind a loop. Structures and actions following S_j are merged with those following S_i if they are identical; otherwise, the loop is exited via the last action edge. This method produces one of the possibly many simple-loop paths from π; we could also produce all such paths. $formLoops$ thus gives us a generalization Π of π. We then use the $findPrecons$ subroutine to obtain the restriction on S_0 for which Π works.

Example

We illustrate this idea using the blocks world setting from the previous section. Let us call the generalized plan shown in Fig.6 Π_G. Suppose we are given a concrete version $\pi = a_1, \ldots, a_n$ of Π_G for the initial structure $S_0^{\#}$ corresponding to a stack with 5 blue blocks above 5 red blocks. π would have 9 $moveToTable$ actions followed by 9 actions alternately moving red and blue blocks to the stack. π could have been generated by any classical planner. Let $S_i^{\#} = a_i(S_{i-1}^{\#}), i > 0$.

To obtain the general plan, we first convert π into a sequence of action transformers by replacing each action with a transformer that uses as its argument(s) any block having the role of the real action's argument(s) (focus-based choice operations described earlier could be used to do so). We then apply the resulting sequence of transformers to the canonical abstraction of $S_0^{\#}$ (Fig.5). This gives us a sequence of sets of structures representing the possibilities after each action. From each set in the sequence, we select the structure S_i such that $S_i^{\#} \sqsubseteq S_i$. Note that π_{abs} has the same start state, and action transformers as Π_G. Further, because of abstraction its structure and action sequence is an unrolled version of Π_G's and every unrolled loop in Π_{abs} is punctuated by a repeated abstract structure.

When $formLoops$ is executed on Π_{abs}, it finds the repeated structures and re-creates the loops, giving us exactly Π_G! Finally, since S_n satisfies the goal, we use the techniques presented in this paper to find the problem instances for which Π will work.

We are thus able to extract the generalized plan Π_G from π, a simple concrete plan that could have been found by any classical planner. This approach is particularly effective if the goal condition only uses abstraction predicates (e.g. $\forall x \neg misplaced(x)$).

Conclusion and Future Work

We present a new framework for generalized planning using abstraction across problem instances to conduct the search for generalized plans. The main contributions of the paper are the presentation of an abstraction technique particularly conducive for this purpose, the formulation of the planning algorithms, and a precise analytical characterization of the settings in which they work. We also show how to use this framework to learn a general plan from examples. Our planning algorithm is only partially implemented; a full implementation requires interfacing with TVLA and is left for future work. We use several examples to illustrate how the algorithm works and to demonstrate the overall power of this approach.

Our framework opens up several interesting research avenues for future work. Searching in an abstract space that spans different problem instances presents new challenges for heuristic and pruning techniques. Generalizing our techniques to a wider class of domains and plans with nested loops are natural extensions that are worth exploring. Finally, other annotation languages could be examined as they may provide a way of moving beyond extended-LL domains.

References

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-Based Generation of Universal Plans in Non-Deterministic Domains. *Proc. of AAAI-98*.

Feng, Z., and Hansen, E. A. 2002. Symbolic Heuristic Search for factored Markov Decision Processes. *Proc. of AAAI-02*

Fikes, R.; Hart, P.; and Nilsson, N. 1972. Learning and Executing Generalized Robot Plans. Technical report, AI Center, SRI International.

Hammond, K. J. 1996. Chef: A Model of Case Based Planning. *Proc. of AAAI-96*

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. *Proc. of UAI-99*

Kambhampati, S., and Kedar, S. 1993. A Unified Framework for Explanation-Based Generalization of Partially Ordered and Partially Instantiated Plans. *Artificial Intelligence*.

Knoblock, C. A. 1991. Search Reduction in Hierarchical Problem Solving. *Proc. of AAAI-91*

Loginov, A.; Reps, T.; and Sagiv, M. 2006. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. *Proc. of SAS-06*

Sagiv, M.; Reps, T.; and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *Proc. of TOPLAS-02*.

Veloso, M. 1994. *Planning and Learning by Analogical Reasoning*. Springer-Verlag.

Winner, E., and Veloso, M. 2002. Automatically acquiring planning templates from example plans. *Proc. of AIPS-02*.

Winner, E., and Veloso, M. 2003. DISTILL: Learning domain-specific planners by example. *Proc. of ICML-03*