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 novel approaches for finding such plans through search and for learning them from examples. We use state representation and abstraction techniques originally developed for static analysis of programs. The *generalized plans* that we compute include loops and work for classes of problems having varying numbers of objects that must be manipulated to reach the goal. Our algorithm for learning generalized plans takes as its input an example plan for a certain problem instance and a finite 3-valued first-order structure representing a set of initial states from different problem instances. It learns a generalized plan along with a classification of the problem instances where it works. The algorithm for finding plans takes as input a similar 3-valued structure and a goal test. Its output is a set of generalized plans and conditions describing the problem instances for which they work.

1 Introduction

In this paper we develop a new unified framework for computing generalized plans using search or learning in an abstracted state space. We use it to construct novel algorithms for learning or directly searching for provably correct plans for sets of problem instances from a domain. These problem instances 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. This is accomplished by including loops over such objects. Our plans are thus close to algorithms: our input represents an abstract planning problem and the generalized plans we compute solve it for a range of problem instances. We also present a precise analytical characterization of the domains (*extended-LL* domains) where our techniques are guaranteed to work.

This paper is organized as follows. We first provide a high level overview of our technique, followed by sections describing the abstraction mechanism, our methodology and the requirements we impose in order to identify a useful category of domains where we can currently find accurate plan-preconditions. This is followed by sections on learning generalized plans from examples and on finding them from scratch. Since this paper is restricted to 6 pages, all proofs and most of the details can be found at [Srivastava, Immerman, & Zilberstein, 2007]. To save space, we provide only a sketch of the TVLA system. We suggest the reader use our discussion of TVLA as an overview for the description in [Sagiv, Reps, & Wilhelm, 2002].

[*]Research of the first two authors supported in part by NSF CCF-0541018

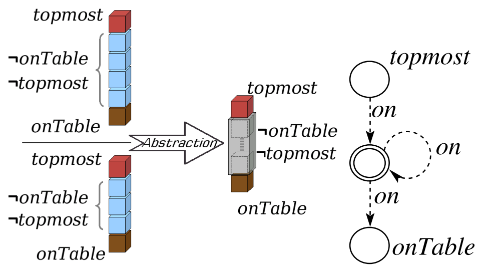

Figure 1: Canonical abstraction in blocks world. Abstracted objects are encapsulated. The abstraction predicates are *topmost* and $onTable$; diagram on the right shows the state in TVLA notation.

2 Overview of the Approach

Our approach uses *abstraction using 3-valued logic*, which has been used effectively in the Three-Valued Logic Analyzer (TVLA) – a tool for static analysis of programs that manipulate pointers [Sagiv, Reps, & Wilhelm, 2002; Loginov, Reps, & Sagiv, 2006]. This 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, 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 operators 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 state into two relevant cases.

Our methodology for finding plans 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

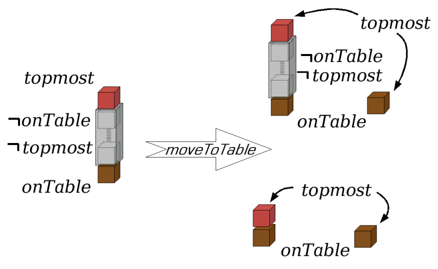

Figure 2: Branching. In blocks world the focus operation models "drawing" an object from a role. In doing so it produces a branch on the number of objects left in the $\neg topmost \wedge \neg onTable$ role.

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 abstract start state to reflect the availability of this partial solution. We then repeat this process until either the entire abstract start state 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. Our algorithm for learning plans uses this framework to trace an example plan in the abstract state space; this makes loops inherent in the example plan obvious, as identical abstract state-action sequences.

3 Framework for Abstraction using 3-Valued Logic

We assume that actions are deterministic and that their results are observable. States of a domain are represented by two-valued logical structures consisting of a universe of constant elements or objects, and definitions for all the predicates in a domain-specific vocabulary. We use $[\![\varphi]\!]^S$ to denote the truth value of a closed formula φ in the structure S. State transitions are carried out using action operators described as a set of first-order formulas defining new values of every predicate in terms of the old values. We represent abstract states using structures in three-valued logic, also called "abstract structures".

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: the universe, $|S| = \{b_1, b_2, b_3\}$, $onTable^S = \{b_3\}$, $topmost^S = \{b_1\}$, $on^S = \{(b_1, b_2), (b_2, b_3)\}$. This domain has two actions: $move(A, B)$ moves block A to the top of block B, and $moveToTable(A)$ moves block A to the table.

We assume actions to be deterministic. The action operator for an action a (written τ_a) consists of a set of preconditions and a set of formulas defining the new value p' of each predicate p. We separate these two parts of an action operator: 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. 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 states, Γ.

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

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

The RHS of this equation consists of two clauses, the first of which holds for arguments on which p_i is changed to true by the action; the second clause holds for arguments on which p_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:

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

The goal condition is 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 operators, and φ_g a first-order formula representing the goal condition.

Given a domain-schema, some special unary predicates are classified as *abstraction predicates*. The special status of these predicates arises from the fact that they are preserved in the abstraction. 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 every 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.

3.1 Canonical Abstraction

Canonical abstraction [Sagiv, Reps, & Wilhelm, 2002] abstracts a structure by merging all the objects of a role into a *summary object* of that role. The resulting abstract structure represents structures with any number (at least 1) of objects corresponding to each summary object. The total number of abstract structures for a domain-schema is thus finite; we can tune the choice of abstraction predicates so that the resulting 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 objects 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, Reps, & Wilhelm, 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 embeddable 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 $f : |S_1| \to |S_2|$ 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, $u_1, \ldots, u_k \in |S_1|$, $[\![p(u_1, \ldots, u_k)]\!]^{S_1} \preceq [\![p(f(u_1), \ldots, f(u_k))]\!]^{S_2}$.

The universe of the canonical abstraction, S', of structure, S, is the set of nonempty roles of S. In order to merge all elements that have the same role, we use the subscript $\{p \in A | [\![p(x)]\!]^{S,u/x} = 1\}, \{p \in A | [\![p(x)]\!]^{S,u/x} = 0\}$ to denote elements in the abstracted domain.

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

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: $\gamma(S) = \{S' | \exists f : S' \sqsubseteq^f S\}$.

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 operators 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 satisfies 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 during TVLA's *coerce* operation 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.

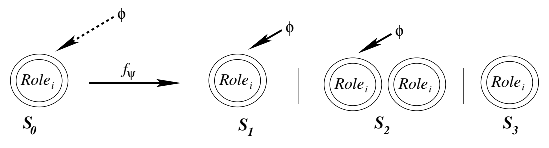

Figure 3: Effect of focus with respect to φ.

3.2 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. Our algorithm chooses 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*.

3.3 Action Application

Recall that the predicate update formulas for an action operator 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 (unary) 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 using focus formulas $\Delta_i^{\pm}$. Once the action operator has been applied, we apply canonical abstraction (this is called "blur" in TVLA) on the resulting structures to get the abstract result structures.

3.4 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 relations $\xrightarrow{a}$, captures the combined effect of these steps:

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 $\{(S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2) | S_1^i \in f_{\psi_a}(S_1) \wedge S_2^i = \tau_a(S_1^i) \wedge S_2 = blur(S_2^i)\}$.

4 Finding Preconditions

For both searching for plans from scratch and learning from examples, we need to be able to find the concrete states that a given path of transitions, possibly with loops, can take to the goal. In order to accomplish this, we need a way of representing regions of abstract states that are guaranteed to take a particular branch of an action's focus operation. We also need to be able to regress these subsets backwards through action edges in the given path all the way up to the initial abstract state – thus identifying its "solved" concrete members. While the terms "structure" and "state" are interchangeable in our setting, we will use the former when dealing with a logic-based mechanism.

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

Definition 6 *(Annotated Structures)* Let $\mathcal{C}$ be a language for expressing constraints on three-valued structures. A $\mathcal{C}-$*annotated structure* $S|_C$ is the 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. Thus we have $\gamma(S|_C) = \gamma(S)|_C$.

The (annotated) pre-image of an annotated structure gives us the preconditions for reaching that annotated structure. If finding this pre-image is possible, we say the domain is *amenable to back propagation*:

Definition 7 *(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. Using the abbreviation $\tau_{k\ldots1}$ to represent the successive application of action transformers a_1 through a_k in this order, we get:

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.

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 of constraints expressed as sets of linear inequalities using $\#_{R_i}(S)$, for annotations.

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.

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 definition and theorem identify a class of domains where back propagation is possible.

Definition 8 *(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.

Handling Paths with Loops In extended-LL domains we can also effectively propagate annotations back through paths consisting of simple (non-nested) loops:

Proposition 2 (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 structures before entering the loop and after exit be S and S_f. 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.*

Theorem 1 *Extended-LL Domains are amenable to back-propagation.*

Methods described in Srivastava, Immerman, & Zilberstein [2007] can be used to find plan pre-conditions in extended-LL domains.

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 change roles 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), problems in the rocket domain, assembly domains where different objects can be constructed from constituent objects of different roles.

5 Algorithm for Generalizing From Examples

In this section we present our approach 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 them by tracing the example plan in the abstract state space and looking for

```
Input: $\pi = (a_1, \ldots, a_n)$: plan for $S_0^\#$; $S_i^\# = a_i(S_{i-1}^\#)$
Output: Generalized plan $\Pi$
1 $S_0 \leftarrow c(S^\#)$; $\Pi \leftarrow \pi$; $C_\Pi \leftarrow \top$
2 Apply operators for $\pi$ 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 $S_i^\# \sqsubseteq S_i$.
3 if $\exists C \in \mathcal{C}_I(\mathcal{R}) : S_n|_C \models \varphi_g$ then
4     $\Pi \leftarrow$ formLoops($S_0 : a_1 \ldots, S_{n-1} : a_n, S_n$)
5     $C_\Pi \leftarrow$ findPrecon($S_0, \Pi, \varphi_g$)
  end
6 return $\Pi, C_\Pi$
```

Algorithm 1: GeneralizeExample

identical abstract state and action sequences. We can then enhance this plan using the identified loops and use the techniques discussed above to find the set of problem instances for which this new generalized plan will work. The procedure is shown in Algorithm 1. It is described with an example in the next section; we describe the main subroutines here. Suppose we are given a concrete example plan π for a state $S_0^\#$. We work with S_0, an abstract state containing $S_0^\#$. S_0 can be any structure which makes the resulting domain extended-LL, and for which we need a generalized plan; the canonical abstraction of $S_0^\#$ forms a natural choice.

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.

5.1 Example

We illustrate this idea using an example in the blocks world domain. Given a stack consisting of red blocks below and blue blocks above, we need to find a plan for constructing a stack with alternating red and blue blocks above an assigned red, $base$ block with a blue block on top. The numbers of red and blue blocks in the given stack are unknown, and unbounded.

Representation Our vocabulary consists of the predicates $\{t[on]^2, on^2, topmost^1, onTable^1, \text{obj}_1^1, \text{obj}_2^1, red^1, blue^1, base^1, misplaced^1\}$, where all the unary predicates are abstraction predicates. The $misplaced$ predicate is used to check that a block is above a stack of alternating colors, *starting with a red block.* $t[on]$ is the transitive closure of on, and is used in integrity constraints for the coerce operation. The $\text{obj}_1, \text{obj}_2$ 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. For simplicity of demonstration, we restrict to only one stack on the table in this example (for discussion on multiple stacks, see Srivastava, Immerman, & Zilberstein [2007]). This representation can also be used to describe linked lists and subsequently for creating or learning algorithms for some

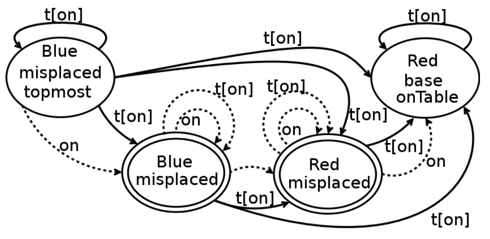

Figure 4: Initial structure for striped blocks world.

```
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 5: A generalized plan with three loops.

programming tasks. The planning goal is to reach a state satisfying $\forall x \neg misplaced(x) \wedge (topmost(x) \implies blue(x))$.

Fig.5 shows Π_G, a generalized plan for achieving the desired configuration. This plan is the goal for our learning algorithm, and we show how to learn it from a concrete plan for a fixed instance of the problem. In fact our techniques can generate Π_G itself through search – we discuss this in the next section.

Suppose we are given a concrete plan $\pi = a_1, \ldots, a_n$ for the initial structure $S_0^\#$ consisting of a stack with 5 blue blocks above 5 red blocks. π consists of the following operations: 5 actions moving blue blocks to the table, followed by 4 actions moving red blocks to the table, and finally by 9 actions moving red and blue blocks alternately back to the stack. This plan can be computed by any classical planner. Fig. 4 shows S_0, the canonical abstraction for $S_0^\#$. $\gamma(S_0)$ contains infinitely many stacks with any number of blue blocks upon any number of red blocks.

Let $S_i^\# = a_i(S_{i-1}^\#), i > 0$. To obtain the general plan, we first convert π into a sequence of action operators by replacing each action with an operator that uses as its argument(s) any block having the role of the real action's argument(s). We then apply the resulting sequence of operators on S_0. 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 $S_i^\# \sqsubseteq S_i$. Let us call the sequence $\{(S_0, a_0), \ldots (S_i, a_i), (S_{n-1}, a_{n-1})\}$ π_{abs}, where S_n is the final structure.

Π_{abs} is actually an unrolled version of Π_G as applied on S_0. Further, every unrolled loop in Π_{abs} is punctuated by a repeated abstract structure. Therefore, when $formLoops$ is executed on Π_{abs}, it finds these repeated structures and recreates the loops, giving us exactly Π_G! As can be observed by the structure of Π_G itself, S_n satisfies the goal and we use the techniques presented in Srivastava, Immerman, &

Zilberstein [2007] to find the annotation on S_0 which Π_G will work.

Our algorithm computes the desired annotation by associating counters $B_t(R_t)$ with roles corresponding to $blue(red$, but not $base)$ blocks that are $onTable$ and $topmost$ and $B_m(R_m)$ with $blue(red)$ blocks that are neither $topmost$ nor $onTable$. The obtained annotation at structure S_0 is $[B_t = l - l_b, R_t = l - l_r - 1, B_m = 3 + l_b, R_m = 4 + l_r]$, where the $l_r(l_b)$ denote number of iterations of the unstack-red (unstack-blue) loop. Together these conditions imply that the total number of blue blocks (counting the topmost blue block and bottommost red block) is $4 + l$, exactly equal to the number of red blocks in the initial structure.

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. We also find the preconditions, or problem instances for which our learned plan achieves the goal.

6 Searching for Generalized Plans

The idea behind our algorithm for searching for generalized plans is to successively search for paths of transitions (possibly with loops) leading to the goal in the abstract state space – and for each such path, to compute the preconditions under which concrete members of the initial abstract state are guaranteed to reach the goal.

Let the annotated-domain-schema $\langle \mathscr{D}, \mathcal{C}_I \rangle$ be an extended-LL domain with a start structure S_0. We can then formulate an algorithm for generalized planning as follows. We proceed in phases of search and annotation. In the search phase, we conduct a search in the abstract state space for states satisfying the goal condition. During search we allow paths to have simple loops. Heuristics applicable to the abstract state space could be used here; we leave this for future work.

When such a path π is found, we enter the annotation phase: we find the annotation on S_0, C_π for π. If C_π adds to the set of concrete states already solved, we include $\langle C_\pi, \pi \rangle$ in the general plan, and continue the search if some part of S_0 does not have plans yet. 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. Since the number of abstract states for any domain-schema is always finite, this algorithm is guaranteed to terminate.

Propositions 2 and 1 thus give us the following theorem:

Theorem 2 (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 with the given abstraction.*

6.1 Example: Striped Blocks World

Our goal is to find a generalized plan for the abstract problem of constructing a stack of blocks of alternating colors as described in the previous section.

Input An abstract structure (Fig.4) representing all problem instances with a single stack of unknown and unbounded numbers of red and blue blocks (at least two of each), with the red blocks below the blue blocks (Fig. 4). The goal condition is $\forall x \neg misplaced(x) \wedge (topmost(x) \implies blue(x))$.

Output Our algorithm finds a set of generalized plans for solving all the *infinitely many* solvable instances of the given abstract problem. It first finds a path each for 2 or 3 red and blue blocks (there have to be at least 2 of each according to S_0). These paths do not contain loops. Finally, it finds Π_G (Fig.5), the most interesting path. Preconditions for each path are calculated as discussed above. A time-based stopping criterion can be used to stop the algorithm before it exhausts the search for all paths to the goal.

We are thus able to use our paradigm for finding generalized plans through search in an abstract state space. For every generalized branch, we provide preconditions using numbers of objects (e.g. #red=#blue). Our ability to model linked lists also allows us to handle program synthesis problems like finding algorithms for reversing linked lists.

7 Related Work

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, Hart, & Nilsson [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 overgeneralization. This initial effort was followed by various approaches to plan reuse. Unlike *case based planning* [Hammond, 1996], our objective is to learn full plans for a well-defined class of similar instances. Also, once the counts of objects satisfy a plan's precondition our entire generalize plan can be executed without further case evaluation. Winner & Veloso [2003] provide an approach for parameterizing, extracting and assimilating example plans into templates. However this technique is limited to plan learning, and does not guarantee correctness. Levesque [2005] presents an interesting method for creating general plans by iteratively generating plans and finding repetitive patterns involving a unique *planning parameter*. Our roles are more generalized forms of such parameters. Also, unlike his approach, we find provably correct plans. However, our approach cannot currently handle numeric fluents, which are accommodated in Levesque's approach.

Our predicate update formulas resemble the successor state axioms in situation calculus: both provide a logical description of the predicates in a successor state. However, we use query evaluation on a compact representation of possible states (a 3-valued structure) rather than theorem proving to derive the effect of an action. Our search also starts with a compact description of initial states.

8 Conclusion and Future Work

We present a new unified framework which uses abstraction across problem instances to learn and search for generalized plans. The main contributions of this paper are the presentation of an abstraction technique particularly conducive for this purpose, the formulation of planning and learning algorithms, and a precise analytical characterization of a setting in which plan correctness is guaranteed. Our planning algorithm is partially implemented; we are working on a full implementation. We use several examples to illustrate how our algorithms work 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. We are exploring natural extensions such as generalizing our techniques to a wider class of domains and to plans with nested loops. Learning from different examples is also a promising area for future research.

References

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. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*.

Levesque, H. J. 2005. Planning with Loops. *In Proc. of IJCAI.*

Loginov, A.; Reps, T.; and Sagiv, M. 2006. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. *Proceedings of Static Analysis Symposium*.

Sagiv, M.; Reps, T.; and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2007. Using Abstraction for Generalized Planning.

Winner, E., and Veloso, M. 2003. Distill: Learning domain-specific planners by example. In *International Conference on Machine Learning*.