Artificial Intelligence (AIJ), Vol. 175, Issue 2, pp. 615-647, 2011

A New Representation and Associated Algorithms for Generalized Planning

Siddharth Srivastava, Neil Immerman, Shlomo Zilberstein

Department of Computer Science
University of Massachusetts,
Amherst, MA 01003

Abstract

Constructing plans that can handle multiple problem instances is a longstanding open problem in AI. We present a framework for *generalized planning* that captures the notion of algorithm-like plans and unifies various approaches developed for addressing this problem. Using this framework, and building on the TVLA system for static analysis of programs, we develop a novel approach for computing generalizations of classical plans by identifying sequences of actions that will make measurable progress when placed in a loop. In a wide class of problems that we characterize formally in the paper, these methods allow us to find generalized plans with loops for solving problem instances of unbounded sizes and also to determine the correctness and applicability of the computed generalized plans. We demonstrate the scope and scalability of the proposed approach on a wide range of planning problems.

Keywords: Automated Planning, Plans with loops, Plan Verification

1. Introduction

Over the years, many researchers have addressed the problem of constructing a *generalized plan* that solves many different planning problems. The fundamental motivation for finding generalized plans stems from classical planning itself. Consider the simple planning problem of unstacking a tower of blocks. Given a problem instance with 3 blocks, with block b_3 on block b_2, and b_2 on b_1, the solution plan would be: $moveToTable(b_3), moveToTable(b_2)$. The problem of classical planning is to find such solution plans for specific problem instances like the three block tower described above. Classical planners tend to suffer significant slowdowns as the number of blocks in such problems is increased. However, many such problems can be addressed by identifying common patterns in solutions, which can be executed repeatedly with minor modifications

Email addresses: siddharth@cs.umass.edu (Siddharth Srivastava), immerman@cs.umass.edu (Neil Immerman), shlomo@cs.umass.edu (Shlomo Zilberstein)

to solve larger problems. Approaches for finding generalized plans aim to identify such common solution and problem structures for efficiently solving new problem instances.

For instance, a generalized formulation of the unstacking problem would be to unstack a tower with an *unknown* number of blocks, or even a set of towers with unknown numbers of blocks in each. Intuitively, such problems can be "solved" by algorithmic plans such as the following "Unstack" plan:

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

1.1. An Execution Model For Generalized Plans

The Unstack plan described above contains the basic idea of how to solve any unstacking problem. However, it cannot be directly executed on a particular instance of the problem. For example, let I be an instance of the unstacking problem. To apply Unstack to I we would first check whether there exists a block that matches the condition of the while loop (a block that is clear and not already on the table). If so, we must choose such a block, b_1, and apply the action $a_1 = \mathit{moveToTable}(b_1)$. These operations need to be repeated as long as possible, thus generating a complete plan, $P = a_1 a_2 \cdots a_k$. (Note that Unstack happens to be a nondeterministic generalized plan: given an instance consisting of several towers of height greater than one, at each step Unstack may choose the top of any such tower to move to the table.)

Fig. 1 extends this approach to a generic model for executing a generalized plan. In this figure, the "world" represents the system on which the plan will be executed, and a *problem instance* is a completely specified state of this system. At any step during plan execution, the current state of the world can be taken into account while computing the next action to be executed; execution starts with the initial state S_0 and terminates with a special termination action (a_f)[1].

The generalized plan therefore executes a *policy with termination actions* which maps sequences of states to actions. Formally, let $\mathcal{S}$ be the set of states in a domain, and $\mathcal{A}$ the set of domain actions. A policy P with termination actions is a function $P : \mathcal{S}^* \rightarrow \mathcal{A} \cup \{a_f\}$ with the restriction that for any $\bar{S_1} \in \mathcal{S}^*$, if we have $P(\bar{S_1}) = a_f$, then $P(\bar{S_1}\bar{S_2}) = a_f$ for all $\bar{S_2} \in \mathcal{S}^*$. This definition, and the subsequent formalization of generalized plans can be extended to partially observable settings by replacing the set of states with a set of observations. The focus of this paper, however, is on completely observable settings.

In deterministic situations, effects of actions on the world can be simulated. Consequently, in such settings generalized plans can be instantiated completely for any initial state by simulating plan execution. In the following development our focus will be on deterministic environments; however, during the process of

[1] Fig. 1 suggests a formal model of a generalized-plan automaton (GPA) interacting in phases with a world-model automaton (WMA): at each round, WMA sends its world state to GPA which transfers to a new program state and sends an action that the WMA then executes. The details of this automaton model are straightforward and we do not go into them here.

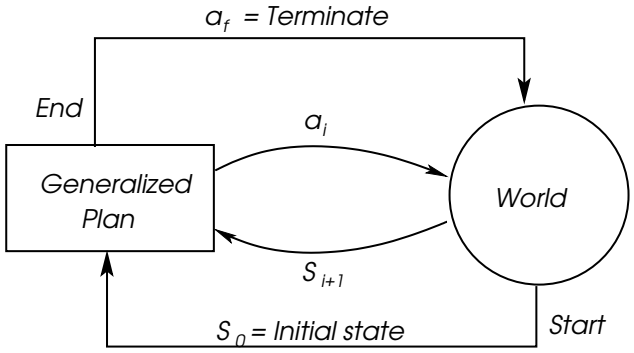

Figure 1: Execution model for generalized plans in deterministic domains

planning we will work with abstract representations of *sets of states* similar to *belief states* as used in planning with partial observability. We discuss how non-determinism and partial observability can be captured in our general approach in section 4.3.

1.2. Architecture of Generalized Plans

Any generalized plan can thus be understood as consisting of two components: (1) a control-structure for representing control knowledge and, (2), a *method for instantiation* which uses this control-structure to compute a policy with termination actions. We will present a formally well-defined class of generalized plans with this architecture, called *graph-based generalized plans* (Definition 3) in the next section.

In general, the *control-structure* component of a generalized plan can be used to store specific algorithms for the class of problem instances of interest (such as a formal representation of the algorithmic plan string of the Unstack plan shown above), or more general domain-control-knowledge (Baier et al., 2007). A generalized plan need not provide the guarantee that all its instantiations will be finite. Plan execution or even a complete offline instantiation of the plan may therefore never terminate. On the other hand, the fact that a plan's instantiation method terminates need not imply that it will always achieve the goal. Proving that a generalized plan is "correct" in the sense of reaching a goal state starting from a given problem instance therefore subsumes proofs of termination as well as goal-reachability.

This architecture of generalized plans unifies various approaches for "efficiently" producing "good" plans for classes of problems. Approaches for macro tabulation such as Triangle Tables (Fikes et al., 1972), or plan compilation such as case-based planning (CBP (Spalazzi, 2001)) can also be understood as developing control-structures in order to utilize instantiation methods more efficient than classical planners. Recent approaches like KPLANNER (Levesque, 2005) and loopDISTILL (Winner and Veloso, 2007) aim to extend the applicability of

generalized plans to unbounded classes of problems by including loops of actions in the generalized plan's control-structure. Planning with hierarchical task networks (HTNs (Erol et al., 1994)) can also be considered as generalized planning with the input task network as a non-deterministic control-structure and an HTN planner as the associated method for instantiation.

1.3. Evaluation Criteria for Generalized Plans

Trivially, *classical planners* can also be used as generalized plans with empty control-structures and instantiation methods based on heuristic search. Classical planners therefore fit naturally into the broad notion of generalized plans by being able to generate a plan for every solvable problem instance, but suffer from expensive methods for instantiation. On the other hand the Unstack algorithm discussed above, is a very specific generalized plan which produces output plans much more efficiently for the problem instances that it can solve. In general, a generalized plan may not solve all the possible problem instances of interest, but it may be computationally much more efficient than a classical planner on the problem instances that it does solve. The benefit of such generalized plans rests on the availability of efficient tests for determining if a given problem instance falls under a given generalized plan's capability. For the Unstack plan, this can be tested efficiently: the goal of the problem should be to have all blocks on the table.

As the discussion above reveals, unlike *classical* plans, the utility of generalized plans depends on several conflicting factors. We list these factors below and discuss each in turn:

Complexity of checking applicability The computational cost of determining if a generalized plan can solve a given problem instance.

Complexity of plan instantiation The total computational cost incurred by the method for instantiation for a given problem instance.

Quality of instantiation A measure of the cost of executing the sequence of actions produced by a generalized plan for a given problem instance.

Domain coverage A measure of the size of the set of solvable problem instances that a generalized plan can solve.

Complexity of computing the generalized plan The computational cost of computing the generalized plan itself.

Complexity of checking applicability An applicability test for a generalized plan is a procedure which takes as its input a problem instance and returns True or False as its output, reflecting whether or not the generalized plan can solve the given problem instance. The complexity of checking applicability is the computational complexity of this procedure. A generalized plan can be designed to proceed in one of two ways when given an input problem instance: (1) conduct a pre-designed applicability test to determine if an instantiation will

be possible, and if so, proceed to find it, or, (2) directly attempt an instantiation. The problem with the second approach is that instantiation can be an expensive and wasteful operation if the generalized plan cannot actually solve the given problem instance. While the first approach is desirable, it is often very difficult to construct an applicability test; the ideal situation would be to have a linear-time or better applicability test.

Approaches for finding generalized plans seldom offer applicability tests. KPLANNER (Levesque, 2005), as an exception, provides a partial test: within the user-requested bounds on a unique parameter that its input problem instances are allowed to vary over, its generalized plans are guaranteed to produce a correct instantiation. Approaches like case-based planning (Spalazzi, 2001) incur large costs of applicability and instantiation while retrieving and adapting previously observed, potentially applicable plans.

Complexity of plan instantiation The complexity of plan instantiation is the total computational cost of executing the method for instantiation for a given problem instance. This factor distinguishes more desirable generalized plans like Unstack above, with an instantiation-complexity linear in the number of blocks (using a list of topmost blocks), from classical planners whose worst-case complexity of instantiation is exponential in the number of objects.

Quality of the instantiation The quality of instantiation of a generalized plan determines its usability on a problem relative to any available alternative solutions. Ideally, the sequence of actions produced by a generalized plan for a given problem should be optimal according to a measure such as the number of actions or their cost. However, in settings where no alternative solutions are available, any instantiation which solves a given problem instance may be desirable.

Domain coverage A concrete plan produced by a classical planner can also be used as a generalized plan by treating the plan itself as the control-structure, and a method that incrementally outputs successive actions from the plan as the method for instantiation. In fact, such generalized plans score very well along all the factors discussed so far, even though they typically work for only one problem instance. The *domain coverage* of a generalized plan evaluates it along one of the most fundamental motivations behind generalized planning: the extent to which the plan is "generalized".

Formally, we first categorize two solvable problem instances as *distinct* if the set of shortest action-sequences for solving each of them have an empty intersection. In other words, a problem instance is distinct from another if the two *require* distinct shortest length solutions. Using this definition, we can define the *size-n domain coverage* ($D_n(\Pi)$) of a generalized plan Π as the ratio of the number of problem instances with n elements that the generalized plan can solve ($S_n(\Pi)$), with the total number of solvable problem instances with n elements ($T_n(\Pi)$). The *asymptotic domain coverage* ($D(\Pi)$) of a generalized plan is defined as the limit of this ratio:

$$D(\Pi) = \lim_{n \to \infty} \frac{S_n(\Pi)}{T_n(\Pi)}$$

The goal of increasing the domain coverage of a generalized plan has received significant attention, starting with initial work by Fikes et al. (1972). Conditional plans typically have a greater domain coverage than classical plans. However, as we discuss below, their coverage is ultimately limited due to their limited expressiveness.

Complexity of computing a generalized plan The complexity of constructing a generalized plan depends on the computational complexity of representing its control-structure. A contingent plan (Peot and Smith, 1992; Bonet and Geffner, 2000) can be used as the control-structure of a generalized plan. Such a generalized plan would have a clear applicability test (by definition, it would solve all instances of the initial belief state used while computing the contingent plan) and a low cost of instantiation. However tree-structured representations used for expressing contingent plans can grow exponentially with every unknown predicate tuple, making such plans inherently more difficult to find. Plan representation thus becomes an important factor when considering the complexity of deriving a generalized plan itself. Approaches like DISTILL, KPLANNER, and BAGGER2 (Shavlik, 1990) mitigate this cost by constructing plans with loops that can instantiate into larger concrete plans. While adding loops can significantly reduce the size of the control-structure used in a generalized plan and increase its domain coverage, it can in general have adverse effects on plan applicability tests and make such plans unreliable. This is because plans with loops and branches approach the expressive power of programs—determining when they will work, or even *terminate* is thus undecidable in general for such plans. HTN's learned using algorithms such as HTN-MAKER (Hogg et al., 2008) can also encode cyclic or recursive task decompositions. However, these approaches do not address the problem of computing applicability tests and incur further costs of instantiation when HTN planners compute solutions from the learned structures.

These five factors together determine the quality and usability of a generalized plan. In the rest of this paper, we describe an approach which addresses the problems associated with all of these factors except for the quality of instantiations, which will be addressed in future work. Our approach draws upon plans for concrete problem instances while creating generalized plans; as such, the quality of instantiations of the resulting plans will depend on these input plans.

1.4. Overview of Our Approach

Fig. 2 shows an overview of our approach. Its input is a classical plan which works for a particular concrete state. As the first step in generalization, we compute and add *choice actions* for selecting the arguments of every action in the input plan. This gives us a linear generalization of the input plan. Next, we apply this linear generalized plan on an abstract state which represents a collection of states including the initial state for which the original plan worked. Action application on an abstract state works along the lines of action application on *belief states* in contingent planning and may produce multiple possible

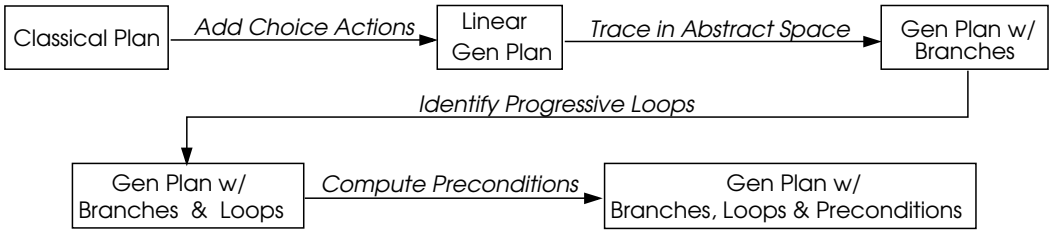

Figure 2: Schematic representation of the overall approach

resulting abstract states. At each step, we keep the abstract state that is consistent with the result of application of the concrete plan on the concrete initial state at that step. Other possible action outcomes are recorded as branches leading to outcomes not handled by this example plan. This process (which we call *tracing*) reveals the effect of the given plan on a class of states. At the same time, because of an abstracted representation, recurring properties become evident as easily identifiable, recurring abstract states. A recurring abstract state is our fundamental cue for identifying a potential loop: it indicates that the sequence of actions lying between the two occurrences can be re-applied. At this point, we need to determine if a loop consisting of this sequence of actions will (a) terminate, and if so, determine the termination conditions, and, (b) make progress towards the goal state.

As the core of our approach, we present methods for efficiently determining answers to both of these problems for a class of problem domains. The first problem is addressed by using changes in the number of objects satisfying certain properties as a measure of progress leading to proofs of termination, akin to related work in model checking such as Terminator (Cook et al., 2006). For addressing the second problem, we propose a novel approach for finding plan preconditions, expressed as combinations of abstract states and linear constraints between constants and counts of objects of certain types. The final guarantee on our computed plans is that they will achieve the goal when applied to any concrete state that is represented by the abstract initial state and satisfies the computed conditions on object counts. We call the resulting approach for generalizing example plans ARANDA-Learn (based on the name of an Australian tribe whose number system captures a similar abstraction).

The rest of this paper is organized as follows. The next section presents our formal framework for representing concrete states, actions and generalized plans. This is followed by a description of a state abstraction technique from software model checking (TVLA (Sagiv et al., 2002)) that allows us to represent unbounded numbers of objects and to identify recurring state properties, or loop invariants (Section 3). Section 4 describes a system for making action application on abstract states more precise. Our approach for finding preconditions of plans with simple loops is described in Section 5, followed by a description of the algorithm for plan generalization in Section 6. Section 7 presents experimental results obtained using an implementation of this approach. This is followed by

a discussion of related work (Section 8) and conclusions (Section 9).

2. Formal Framework

We begin this section by describing the standard, logic-based framework that we use to describe planning. This framework uses two-valued logical structures to represent concrete states and predicate update formulas to represent action updates. We describe our representation of generalized plans in Section 2.1. In subsequent development (Section 3), we will use three-valued logical structures (or "abstract" structures) to represent sets of structures compactly. Throughout this paper, we will use the terms "state" and "structure" interchangeably.

Running Example Consider a unit delivery problem where some crates are at a dock and need to be delivered to their respective destinations via trucks that can only hold one crate at a time.

The state of such a delivery problem is a logical structure of vocabulary $\mathcal{V}_d = \{crate^1, truck^1, loc^1, done^1, destination^2, in^2, at^2; dock\}$, consisting of a constant, *dock*, and predicates whose intuitive meanings are as follows:

- $crate(x), loc(x), truck(x)$: x is a crate, location, or truck, respectively.
- $done(x)$: object x has been delivered.
- $destination(x, y)$: y is the target destination of crate x.
- $in(x, y)$: object x is in truck y.
- $at(x, y)$: object x is at location y.

The delivery domain has the following actions: $\mathcal{A}_d = \langle Move^2, Load^2, Unload^1 \rangle$ with the following intuitive meanings:

- $Move(x, y)$: drive truck x to location y.
- $Load(x, y)$: load crate x into truck y.
- $Unload(x)$: unload the contents of truck x.

Each action a consists of a precondition $pre(a)$ and update formulas, $up(p, a)$, defining the new value of each predicate p after a has been applied. For example, the following is the definition of the action *Move*:

$$\begin{aligned} pre(Move(x,y)) &\equiv truck(x) \wedge loc(y) \wedge \neg at(x,y) \\ up(at(u,v), Move(x,y)) &\equiv [\neg at(u,v) \ \wedge \ (v = y \wedge (u = x \vee in(u,x)))] \\ &\quad \vee \ \ [at(u,v) \ \wedge \ \neg(u = x \vee in(u,x))] \end{aligned}$$

This update states that an object u is at location v after a $Move(x, y)$ operation iff: either, (a) it was not at v before and v is in fact y, and u is either the truck or an object in the truck x, or, (b) it was at v before *Move*, and it is neither the

truck x nor an object in the truck. The *in* predicate is updated similarly by the *Load* and *Unload* actions. The *Unload* action also includes an update for *done*, which is set for the crate being unloaded if the truck is at its destination.

We use the notation up_a to denote the set of all the update formulas for an action, and $up_a(s)$ to denote the result of applying those formulas on a structure s. Throughout this paper, we will represent the update formula for the predicate p—such as the above update formula for the predicate *at*—in the following form, where p' denotes the predicate after action application:

$$p' \equiv [\neg p \wedge \Delta^+_{p,a}] \quad \vee \quad [p \wedge \neg \Delta^-_{p,a}] \tag{1}$$

Here $\Delta^+_{p,a}$ denotes the conditions under which predicate p is changed to true on action a, and $\Delta^-_{p,a}$ denotes the conditions under which it is changed to false. Intuitively, Eq. 1 states that p becomes true for a tuple iff either (a) it was false and action a changes it to true, or, (b) it was already true, and is not removed by action a. In our implementation, constants are represented as unary predicates that are constrained to be unique. They can thus be updated in a manner similar to predicates, using Eq. 1.

In addition to defining the vocabulary and actions of a planning problem, we typically include an *integrity constraint* that specifies the set of valid states. In the abstraction these constraints will be used to clarify the set of concrete states represented by an abstract state.

For example, the integrity constraint, $\mathcal{K}_d$ for our unit delivery is the universally quantified conjunction of the following formulas:

$$\begin{aligned}
done(x) &\rightarrow crate(x) \\
destination(x,y) \wedge destination(x,y') &\rightarrow crate(x) \wedge loc(y) \wedge y = y' \\
crate(x) &\rightarrow \exists y(destination(x,y)) \\
at(x,y) \wedge at(x,y') &\rightarrow loc(y) \wedge (crate(x) \vee truck(x)) \wedge y = y' \\
crate(x) \vee truck(x) &\rightarrow \exists y(at(x,y)) \\
in(x,y) \wedge in(x',y) &\rightarrow crate(x) \wedge truck(y) \wedge x = x'
\end{aligned}$$

Generalizing the above example, we formally define a domain schema for a planning problem as follows:

Definition 1 *(Domain schema)* A domain schema is a tuple $\mathcal{D} = \langle \mathcal{V}, \mathcal{A}, \mathcal{K} \rangle$ where $\mathcal{V}$ is a vocabulary, $\mathcal{A}$ is a set of actions expressed in first-order logic with transitive closure (FO(TC)), and $\mathcal{K}$ is an integrity constraint expressed in FO(TC).

FO(TC) allows us to use the transitive closure of binary relations in integrity constraints, which would not have been possible using first-order logic alone.

We use transitive closure to express connectivity properties such as the transitive closure of *on* ("above") in the blocks world (see the Striped Block Tower, Green Block and Hall-A problems in Section 7 and Appendix A).

Define STRUC[$\mathcal{D}$], to be the set of concrete structures of the domain schema, $\mathcal{D}$, i.e., the set of finite structures of vocabulary $\mathcal{V}$ that satisfy $\mathcal{K}$.

For example, the domain schema of the unit delivery problem is $\mathcal{D}_d = \langle \mathcal{V}_d, \mathcal{A}_d, \mathcal{K}_d \rangle$. We next define a generalized planning problem as follows:

Definition 2 *(Generalized planning problem)* A generalized planning problem is a tuple $\langle \alpha, \mathcal{D}, \gamma \rangle$ where α is an FO(TC) formula describing the possible initial states, $\mathcal{D}$ is the domain schema, and γ is an FO(TC) formula specifying the goal states.

Following the discussion in the introduction, an instance of the generalized planning problem is a concrete initial state, or in other words, a state satisfying the formula α. The unit delivery problem can now be specified as $\mathcal{P}_d = \langle \alpha_d, \mathcal{D}_d, \gamma_d \rangle$ where

$$\begin{aligned} \alpha_d &\equiv \exists x(truck(x)) \wedge \forall x((crate(x) \vee truck(x)) \rightarrow at(x, dock)) \\ \gamma_d &\equiv \forall x(crate(x) \rightarrow done(x)) \end{aligned}$$

2.1. Generalized Plans

Solutions to generalized planning problems are called *generalized plans*. Intuitively, a generalized plan is an algorithm. We represent the control-structure of a generalized plan using a graph representation. Formally,

Definition 3 *(Graph-Based Generalized plan)* A graph-based generalized plan $\Pi = \langle V, E, \ell, s, T \rangle$ is defined as a tuple where V and E are respectively, the vertices and edges of a finite connected, directed graph; ℓ is a function mapping nodes to actions and edges to conditions; s is the start node and T a set of terminal nodes.

We discuss the method of instantiation of graph-based generalized plans below. In the rest of this paper, all references to generalized plans refer to graph-based generalized plans. This representation of actions and plans is similar to situation calculus (Levesque et al., 1998) and Golog programs (Levesque et al., 1997). However, a significant difference between our framework and Golog programs is that we automatically generate edge labels (in the form of summarized, abstract structures) representing the set of concrete states that can provably be solved by the generalized plan starting with the subsequent node's action. Further, while Golog programs are typically hand-coded, albeit sometimes in a partially specified manner, our objective is to automatically find generalized plans and the class of problem instances where they will work.

Fig. 3 shows a generalized plan for the delivery problem. A generalized plan can include *choice* actions for choosing objects to be used as arguments for future actions. These actions select an object which satisfies a given formula in first-order logic, and assign it to a constant used in action update formulas. Intuitively, if multiple objects satisfy the formula used for selection, we

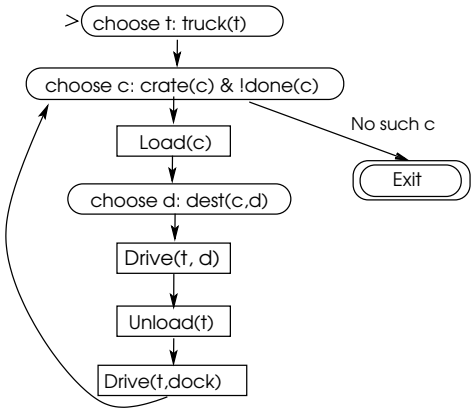

Figure 3: A generalized plan for delivery. The start node is labeled *choose t: truck(t)*.

require that the generalized plan should work with any of those qualifying objects. Choice actions are discussed in detail in Section 4.2; they are constructed automatically in our approach for generalized planning (Section 6.1).

In general, compound node labels consisting of multiple actions and choice actions can be used for ease of expression. For simplicity, we allow only a single action per node.

2.1.1. Instantiation of Graph-Based Generalized Plans

A generalized plan's *control configuration* is given by a tuple $\langle pc, S, i\rangle$ where $pc \in V$ is the current control node, S, the problem state for which an action has to be produced; and i, an instantiation mapping the arguments of $\ell(pc)$ to elements of the state S. As mentioned above, the instantiation i is constructed using choice actions (Section 4.2). A control configuration determines the next action to be executed as the action $\ell(pc)$ with the arguments represented by i. Successive instantiated actions are produced by taking as input, the state resulting from an execution of the previous instantiated action, and following the edge in the generalized plan whose conditions are satisfied by this state, starting with the initial node s. After executing the action at a node $u \in V$, the next possible control nodes are those neighbors v of u for which the condition $\ell(\langle u, v\rangle)$, and the preconditions of action $\ell(v)$ are both satisfied by the current state S with the current instantiation i. We assume the existence of default edges leading to a terminal (trap) state labeled with a termination action, which are taken when suitable next nodes cannot be found in the generalized plan or when an action node is reached without an instantiation for all of its action's arguments.

A generalized plan **solves** a problem instance C (that is, a concrete initial state) if the execution of *every possible instantiation* of the plan on C ends with a structure satisfying the goal. A generalized plan is non-deterministic if it has two edges leaving some node, with overlapping conditions.

In general, it is undecidable to determine the preconditions of a generalized plan because of the undecidability of the halting problem and the fact that a generalized plan can be used to represent an arbitrary program. However, in practice we finesse this problem by only considering finite domains. In particular, we call a generalized planning problem "finitary" if for every problem instance C, the set of reachable states is finite. The simplest way of imposing this constraint is to bound the number of new objects that can be created (or found, in case of partial observability). Finitary domains capture most real-world situations and have a decidable halting problem. In particular, the language consisting of instances that a generalized plan solves in a finitary domain is decidable. This is because in these domains we can maintain a list of visited states (which has to be finite), and identify non-terminating behavior if a state is revisited. We formalize this notion with the following observation:
Observation 1 *(Decidability in finitary domains)* The halting problem and the set of problem instances solved by any generalized plan in a finitary domain is decidable.

3. State Abstraction Using 3-Valued Logic

We now describe a method for state abstraction which can be used to represent unbounded sets of concrete states compactly. This technique was originally developed as a part of the TVLA system (Sagiv et al., 2002) for static analysis. While this approach significantly increases the expressive power of finite logical structures, it also makes the effects of action updates on abstract states imprecise. In the next section (Section 4), we present a method for alleviating this problem.

The TVLA system represents sets of concrete structures using a single, bounded-size three-valued logical structure. In a 3-valued structure, each tuple may be present in a relation with definite logical values 1 (present), 0 (not present), or indefinite value $\frac{1}{2}$ (perhaps present). In the following formalization, we will use the symbol $|S|$ to denote the universe of a structure S, $[\![\varphi]\!]_S$ to denote the truth value of a formula φ in S, and $[\![c_j]\!]_S$ to be the unique element in $|S|$ corresponding to a constant c_j in its vocabulary.

Definition 4 *(3-valued Structure)* A *3-valued structure*, also called an *abstract structure*, S over vocabulary $\mathcal{V} = \langle p_1^{a_1}, \ldots, p_r^{a_r}; c_1, \ldots, c_t \rangle$ with predicates p_1, $\ldots, p_r$ of arities $a_1, \ldots, a_r$ respectively, consists of a non-empty universe $|S|$, and for every predicate symbol $p_i^{a_i}$ and tuple $(u_1, \ldots, u_{a_i}) \in |S|^{a_i}$, a truth value $[\![p(u_1, \ldots, u_k)]\!]_S \in \{0, 1, \frac{1}{2}\}$, and for every constant symbol c_j an element of the universe, $[\![c_j]\!]_S \in |S|$.

The equality relation in a three-valued structure distinguishes *summary elements*, $s \in |S|$, which may represent more than one element of a concrete structure, from *non-summary elements*, $n \in |S|$, which must represent a unique element. Summary elements satisfy $[\![s = s]\!]_S = \frac{1}{2}$, whereas non-summary elements satisfy $[\![n = n]\!]_S = 1$.

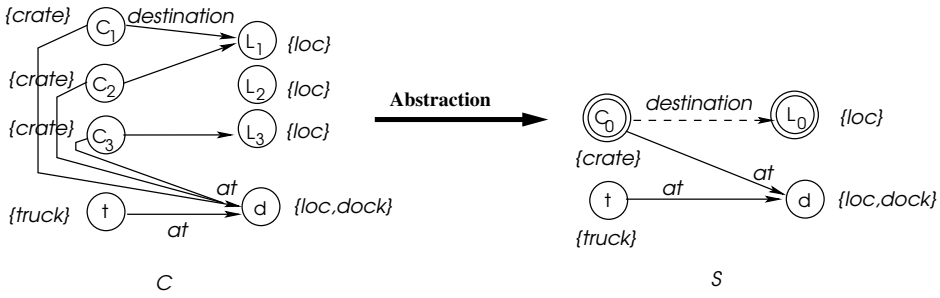

Figure 4: Abstraction in the delivery domain

Example 1 Fig. 4 shows a diagram of a concrete structure, C, representing a state in a unit delivery problem. The universe of C consists of three crates (C_1, C_2, C_3), one truck, one dock, and three locations (L_1, L_2, L_3). A three-valued structure, S, is shown on the right. The double circles represent summary locations. The solid arrows represent truth values of "1" and the dotted arrows represent truth values of "$\frac{1}{2}$". Intuitively, because of the summary elements, the abstract structure S represents the concrete structure, C, as well as all other unit delivery problems that have exactly one truck, with the truck at the dock and empty, and at least one location different from the dock.

To define what it means for one structure to represent another structure, we first define the information ordering: "$x \prec y$" to mean that y is more general than x, i.e, $y = \frac{1}{2}$ and $x \in \{0, 1\}$. Let $x \preceq y$ mean that $x \prec y$ or $x = y$.

Structure S_2 *represents* structure S_1 iff S_1 is *embeddable* in S_2. An embedding is a map from $|S_1|$ onto $|S_2|$ that is monotonic with respect to $\preceq$, i.e. truth does not change, but it may become less precise:

Definition 5 *(Embeddings)* The function $f : |S_1| \underset{\text{onto}}{\rightarrow} |S_2|$ *embeds* S_1 *in* S_2 ($S_1 \sqsubseteq^f S_2$) iff for all relation symbols p^a and elements, $u_1, \ldots, u_a \in |S_1|$, $[\![p(u_1, \ldots, u_a)]\!]_{S_1} \preceq [\![p(f(u_1), \ldots, f(u_a))]\!]_{S_2}$ and for every constant symbol c, $f([\![c]\!]_{S_1}) = [\![c]\!]_{S_2}$.

For dom $\mathcal{D} = \langle \mathcal{V}, \mathcal{A}, \mathcal{K} \rangle$, we use the notation,

$$\gamma_D(S) \quad = \quad \left\{ C \in \text{STRUC}[D] \;\middle|\; \exists f : C \sqsubseteq^f S \right\}$$

to denote the set of (concrete) structures of D that are represented by S. When D is understood, we just write $\gamma(S)$.

In a domain schema, a subset of the unary predicates, A, is identified as the set of *abstraction predicates*. The abstraction process that we describe below may obscure some of a state's properties, but always represents its abstraction predicates accurately. Selecting abstractions to correctly highlight the most signifcant properties of a problem domain while obscuring any irrelevant ones is a longstanding and widely appreciated problem in AI, and is beyond the

scope of the current paper. The function of abstraction predicates suggests that we should have sufficient abstraction predicates to be able to determine if an abstract state satisfies the goal condition. This can help in choosing the set of abstraction predicates for a domain. However, in all the examples used in this paper, the set of abstraction predicates is exactly the set of unary predicates in the domain.

Definition 6 *(Role)* The role of an element $a \in |S|$ is the set of abstraction predicates that it satisfies and the set of constants that it is equal to:

$$role(a) \quad = \quad \{p_i \in A \mid [\![p_i(a)]\!]_S = 1\} \cup \{c_j \mid [\![c_j]\!]_S = a\} .$$

For example, in Fig. 4 elements C_1, C_2, C_3 of the universe have the role $\{crate\}$, t has the role $\{truck\}$, L_1, L_2, L_3 have the role $\{loc\}$, and d has the role $\{loc, dock\}$. In the following development, we will measure the progress made by loops of actions in terms of changes in the number of objects satisfying each role.

Each concrete structure C is represented by its *canonical abstraction*: the most precise abstract structure in which all elements of C with the same role are merged together into a summary element of that role (since exactly one element in a structure can represent a constant, constants will always be interpreted as non-summary elements):

Definition 7 *(Canonical Abstraction)* The canonical abstraction of a concrete structure C is $S = canon(C)$ with $|S| = \{e_r : \exists u \in |C|(r = role(u))\}$, with embedding $C \sqsubseteq^f S$ such that:

1. $f(u) = e_{role(u)}$
2. $[\![p(e_1, \ldots, e_a)]\!]_S \quad = \quad \sup_{\preceq}\{[\![p(u_1, \ldots, u_a)]\!]_C \mid f(u_i) = e_i, i = 1, \ldots, a\}$, for all predicate symbols p^a.

Thus the truth value of $r(e_1, \ldots e_n)$ in S is the definite value 0 or 1, if C agrees on that value of $r(u_1, \ldots, u_n)$ for all elements of C of the appropriate roles. Otherwise, the value in S is $\frac{1}{2}$. For example, in Fig. 4, $S = canon(C)$. In general, suppose that C is a concrete structure and $S = canon(C)$. Then by the above definition, e_r is a summary element of S i.e., $[\![e_r = e_r]\!]_S = \frac{1}{2}$, iff C has more than one element of role r. Furthermore, regardless of how large C is, $|S|$ has no more than 2^a elements where a is the total number of constant symbols and abstraction predicates. Increasing the number of abstraction predicates makes canonical abstractions more precise at the cost of increasing their size.

4. Action Application on Abstract states

We now present the methodology for applying action updates on abstract states. We begin by describing TVLA's focus and coerce operations, which make abstract structures more precise prior to action application; we describe how these operations are used in our system for generalized planning in Section

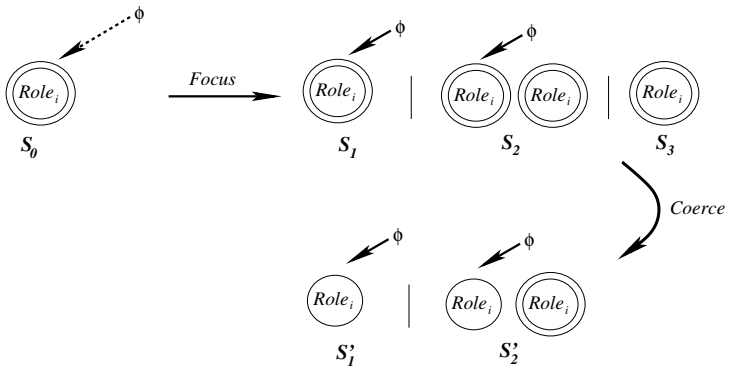

Figure 5: Effect of focus and coerce with respect to ϕ, a formula constrained to hold for a unique element.

4.1.1, followed by a description of choice actions in Section 4.2. Finally, we present a brief discussion of how this framework relates to, and can be used for, modelling belief states and non-deterministic sensing actions of contingent planning.

When applied to an abstract structure with imprecise truth values, update formulas for actions might evaluate to $\frac{1}{2}$. Propagation of the $\frac{1}{2}$ truth value in this way can quickly result in very imprecise structures with no useful information. This is mitigated in TVLA using the *focus* and *coerce* operations.

4.1. Focus and Coerce

Given an abstract structure S and a formula ϕ on which we need precision, a "focus" operation is defined as one that produces a set of possibly abstract structures, $Focus(S, \phi) = \{S_1, S_2, \ldots S_k\}$, which capture exactly $\gamma(S)$ (the set of concrete structures represented by S), and in each of which ϕ evaluates to a definite truth value for any possible instantiation of its free variables. In general, the set $Focus(S, \phi)$ may be infinite. Consequently, there is no general algorithm for focus.

The idea behind TVLA's limited focus algorithm is illustrated on the top row of Fig. 5: if $\phi()$ evaluates to $\frac{1}{2}$ on a summary element, e, then this can be captured by three different abstract structures corresponding to cases where: either all of e satisfies ϕ, or part of it does and part of it doesn't, or none of it does. Additional elements created during this process (as in S_2) inherit the truth values of other predicates from the original summary element. Note that ϕ evaluates to a definite truth value for all elements in all of the structures (S_1, S_2 and S_3) produced by focus. The focus algorithm on a binary predicate, at most one of whose arguments is a summary element, follows the same methodology. In fact, this algorithm works in any situation where at most one of a predicate's free variables is interpreted with a summary element (the focus formulas used in this paper satisfy this requirement). Otherwise, this algorithm does not

terminate. The focus operation w.r.t. a set of formulas works by successive focusing w.r.t. each formula in turn.

This process of splitting summary elements could produce structures that violate the integrity constraints. TVLA's *coerce* operation traverses the list of focused structures. If any structure is inconsistent with the integrity constraints, it is removed; otherwise, coerce attempts to make the truth values of predicates in the structure more precise in order to satisfy the integrity constraints with the truth value 1. Further descriptions of both focus and coerce operations can be found at (Sagiv et al., 2002).

4.1.1. Action Specific Focus Formulas

Using focus prior to action application can improve the precision of action updates. Recall that the predicate update formulas for an action operator take the form shown in Eq. 1. For unary predicate updates, expressions for Δ_i^+ and Δ_i^- are *monadic* (i.e. have only one free variable, corresponding to the free variable on the LHS, apart from action arguments whose values will be constants when an action is applied). When applied on a structure with precise truth values for abstraction predicates, an update of the form of Eq. 1 can result in imprecise truth values for these predicates only if the formulas $\Delta^{\pm}$ evaluate to imprecise truth values. Consequently, in order to keep the abstraction predicates precise, we focus on $\Delta^{\pm}$ expressions prior to action application.

Therefore, in this paper, the set of focus formulas to be used prior to an action update will be exactly the $\Delta^{\pm}$ formulas for the abstraction predicate updates. The fact that these formulas are monadic ensures that the focus algorithm with these formulas terminates. We use F_a to denote this set of focus formulas for an action a. We illustrate this choice of focus formulas using the following example from the blocks world, since non-choice actions in the unit delivery problem do not need focus formulas.

Example 2 Consider a blocks world domain schema with the vocabulary $\mathcal{V} = \{on^2, topmost^1, onTable^1\}$, and abstraction predicates $\{topmost, onTable\}$. Consider the *Move* action which has has two arguments: obj_1, the block to be moved, and obj_2, the block it will be placed on. The update formula for *topmost* is:

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

Following the discussion above, the update formula for *topmost* can evaluate to $\frac{1}{2}$ because $on(\text{obj}_1, x)$ can evaluate to $\frac{1}{2}$ in an abstract structure (see Fig. 15 on page 39 for an example of an abstract structure in the blocks world). Consequently, $on(\text{obj}_1, x) \wedge (x \neq \text{obj}_2)$ is the focus formula for *Move*() (note that this subsumes the Δ^- portion of the second part of the disjunction). In effect, for the focus operation, this formula is $on(\text{obj}_1, x)$ because $x \neq \text{obj}_2$ will evaluate to a definite truth value for every instantiation of x. This is because the constants obj_1 and obj_2 will be assigned to singleton elements by choice actions prior to the *Move* action.

4.2. Isolating Action Arguments

The previous section described methods for making action updates precise *after* suitable action arguments had been selected and labeled by constant symbols. We will now describe how action arguments can be selected in an abstract structure. This requires special techniques because elements of an abstract structure can be summary elements representing sets of similar concrete elements. Actions however, are typically applied upon individual concrete elements. We use focus and coerce to develop an effective mechanism for drawing out representative elements from their summary elements for later use as action arguments.

Consider Fig. 5. If integrity constraints restricted ϕ to be unique and satisfiable, then structure S_3 in Fig. 5 would be discarded by coerce. Further, the summary elements for which $\phi()$ holds in S_1 and S_2 would be replaced by singletons. This would result in two structures, shown in the lower row in Fig. 5: (1) S_1', which has only one element with $Role_i$, and $\phi()$ holds for this element, and (2) S_2', which has multiple elements of $Role_i$, for one of which $\phi()$ holds. In other words, this combination of focus and coerce yields two possible situations depending on whether the summary element of $Role_i$ in S_0 represents exactly one, or more than one elements. This combination of focus and coerce simulates a general "drawing-out" operation from a non-empty set whose cardinality is unknown. A formal analysis of such focus operations and the necessity of classifying its outcomes by comparing certain role-counts with the constant 1 is presented in Section 5.2 on page 27 (in particular, see Proposition 1 and the following discussion).

From the point of view of action application, this operation has the effect of chosing singleton elements from a role represented by a summary element; these singletons can be used as action arguments. Choice actions of the form "choose c: $\xi(c)$" can therefore be implemented by applying the following steps on a given structure. ("*chosen*" is a new predicate, with the integrity constraint of uniqueness)

1. Set the *chosen* predicate: $chosen'(x) \equiv \xi(x) \wedge \frac{1}{2}$
2. Focus w.r.t $chosen(x)$: This triggers drawing out operations if *chosen* holds with the truth value $\frac{1}{2}$ for a summary element, as discussed above.
3. Set the argument: for every resulting structure, set constant c to the element satisfying *chosen*.

Example 3 Consider the sequence of operations in Fig. 6 in a simplified version of the delivery domain (we ignore the trucks and current positions of crates). $chosen(x)$ is initialized to $\frac{1}{2}$ for all objects with the role *crate* in this figure. The first focus operation illustrates the drawing out of an action argument from its summary element, in this case, of role $\{crate\}$. A constant c is set to the drawn out crate, concluding the choice operation. The second focus operation focuses on $destination(c, x)$, effectively creating possible cases for the destination of crate c. Integrity constraints are used to assert that (a) $chosen(x)$ must hold

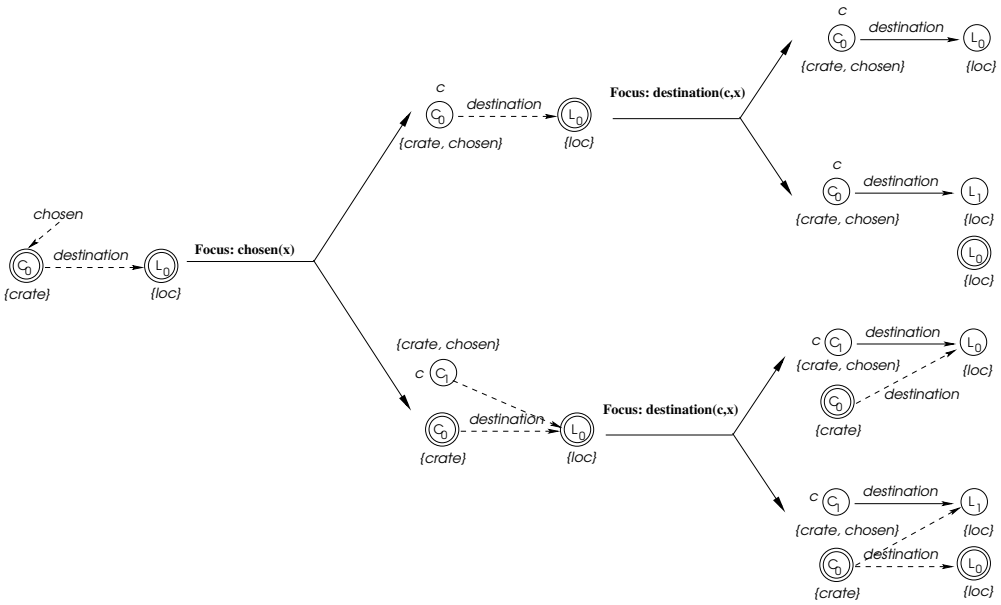

Figure 6: A sequence of focus operations in the delivery domain.

for a unique element, and (b) every crate has a unique destination, so that coerce discards structures where c has none, or non-unique destinations.

Note that in this example, different outcomes of focus operations can be easily differentiated on the basis of the number of elements of a role (the two possible outcomes of the first focus operation are characterized by whether or not there are at least two objects with the role $\{crate\}$). This becomes useful when we need to find the conditions under which an action branch leading to a goal will be taken (Section 5).

Summary of Action Application on Abstract Structures The overall process of applying actions on abstract structures is shown in Fig. 7. The abstract structure is first focused w.r.t. action-specific focus formulas. The resulting focused structures are then tested against the preconditions, and action updates (up_a) are applied to those for which the preconditions evaluate to 1. Any constants representing action arguments are then removed and the resulting structures are canonically abstracted, leading to the final results.

We formalize the different phases of action application as an action transition:

Definition 8 *(Action Transition)* Let a be an action and S_1 a three-valued structure with constants representing each of a's arguments. $S_1 \xrightarrow{a} S_2$ holds iff S_1 and S_2 are three-valued structures and there exists a focused structure $S_1^1 \in f_{F_a}(S_1)$ s.t. $S_2 = canon(up_a(S_1^1))$. The transition $S_1 \xrightarrow{a} S_2$ can be decomposed

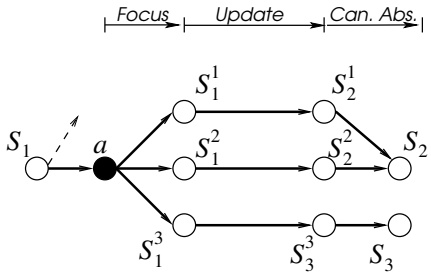

Figure 7: Action update mechanism

into a set of transition sequences for each result of the focus operation: $\{(S_1 \xrightarrow{f_{F_a}} S_1^i \xrightarrow{up_a} S_2^i \xrightarrow{c} S_2) | S_1^i \in f_{F_a}(S_1) \wedge S_2^i = up_a(S_1^i) \wedge S_2 = canon(S_2^i)\}$.

4.3. Canonical Abstraction as a Representation for Belief States

The abstraction methodology described in the previous sections translates the generalized planning problem into a contingent planning problem with partially observable states. More precisely, this abstraction results in a state space with uncertainty about object quantities and properties, such that the only information about object quantities available to the agent during planning is whether there exist there exist zero, one, or more than one elements of each role. These abstract states represent sets of possible concrete states in a manner similar to the modelling approach used in contingent planning, where belief states (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005) represent sets of possible real world states which are indistinguishable due to lack of information. Existing belief state representations, however, cannot capture uncertainty in object quantities. Contingent planners use "sensing" actions to determine properties of belief states. A sensing action results in multiple possible belief states, corresponding to the different values of the property being sensed.

Focus operations associated with actions described in the previous section are thus analogous to sensing actions of contingent planning. More precisely, we can define a sensing action in our framework as an action operator with a given monadic focus formula representing the property to be sensed. The only difference between such actions and a regular action operator in our framework is that the focus formula for a sensing action is specified independently of the updates that the action may perform.

Example 4 A partially observable version of the delivery domain can be constructed by adding uncertainty about the number of crates and locations and the *destination* relation. The canonically abstracted structure on the right in Fig. 4 can be used to represent the belief state of such a formalization. We can define a sensing action, *findDest(c,d)*, for determining a crate's destination using the focus formula $dest(c, l)$ and update formulas setting a new constant d to

the crate's destination. This formulation allows us to solve the sensing version of the delivery problem, as discussed in Section 7.

In the following sections we use the abstraction and action mechanisms presented above to develop algorithms for generalized planning.

5. Computing Preconditions of Plans with Simple Loops of Actions

In this section we present our approach for computing preconditions of plans with simple loops of actions. We define a simple loop in a graph as follows:

Definition 9 *(Simple Loop)* A simple loop in a graph is a maximal strongly connected component consisting of exactly one cycle.

We begin by illustrating the idea behind finding preconditions for success of action sequences on a special class of domains that use only unary predicates. These ideas are then generalized to abstract domains with binary relations that satisfy some key requirements (FC^3 domains, Def. 12). A complete presentation of the method for finding preconditions is provided in Section 5.1. Section 5.2 presents a set of necessary conditions under which canonical abstraction produces FC^3 domains; the complexity of our algorithms is discussed in Section 5.2.1. Finally, Section 5.3 discusses a special class of the domains where our approach for finding preconditions is applicable; the transport example discussed below will turn out to be a member of this class.

Consider a simplified transport domain where objects need to be moved from one location to another by a single truck of capacity one. The vocabulary for this domain consists of unary predicates {*atL1, atL2, inT, object, truck*}. The actions are

- $moveTL_i()$: move the truck to location i,
- $loadT(x)$: load object x into the truck,
- $unloadT()$: unload object from the truck.

Fig. 8 shows a sequence of actions on an abstract initial structure S_1. For the purpose of this example, assume that the goal is to have exactly one object at $L1$, as in structure S_6. Note that this sequence of actions creates a loop with the only occurring branch caused by the choice action. Unlike a loop over a sequence of concrete states, this loop makes progress towards the goal.

In this case, it is possible to compute the changes in role-counts due to each action. It can also be proved that every concrete structure represented by the abstract structures in Fig. 8 will undergo the same changes, as annotated near the top of the figure (this is not true in general for action application on abstract states). Further, the condition determining whether or not the branch exiting the loop is taken can be determined, and depends on a role-count.

Let n denote the initial role-count of $\{object, atL1\}$ for a concrete structure embeddable in S_1. The role-count change annotations near the top of Fig. 8

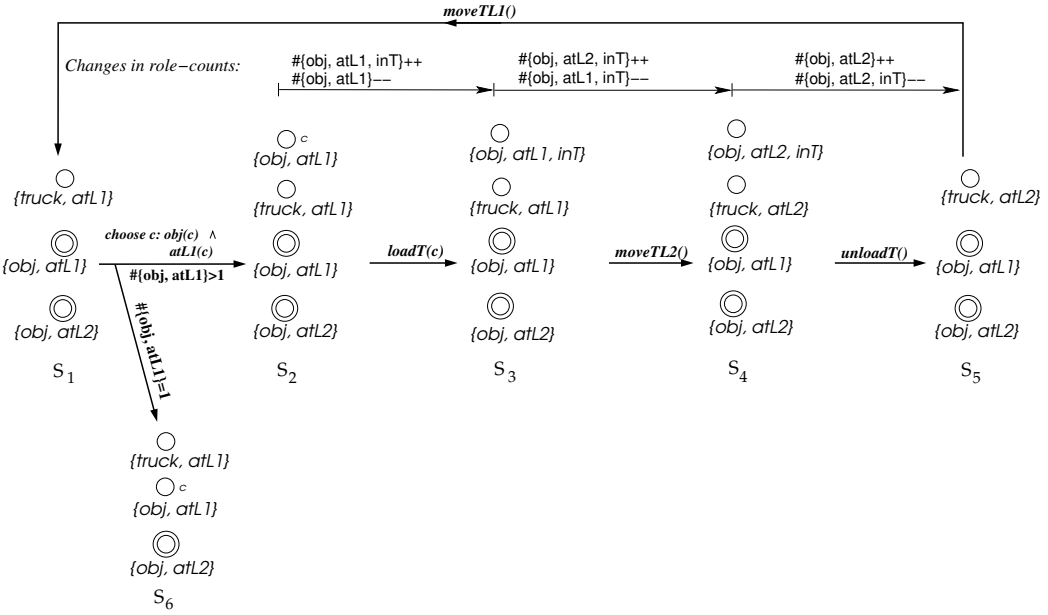

Figure 8: A sequence of actions in a unary representation of transport domain. Role-count changes are shown only for roles involving *object*, abbreviated as *obj*.

indicate that n will drop by one in every iteration of the loop. Therefore, we can determine that the branch exiting the loop will be taken after exactly $n-1$ iterations. This means that

1. The goal is provably reachable from *any* of the infinitely many structures represented by S_1.
2. Given a structure $s \in S_1$ the number of steps required to reach the goal following the given loop can be easily determined.

In any domain representation constructed using just unary predicates if action arguments are drawn out prior to action application (Section 4.2), it is possible to carry out this method of analysis to determine facts like (1) and (2) above for a generalized plan with any number of simple loops (this is discussed formally in Section 5.3). In the remainder of this section we provide the details for a generalization of this technique to a broader class of domains.

The most important properties of the simplified transport domain that made it possible for us to determine preconditions for the loop of actions in Fig. 8 are:

1. When an action has multiple abstract structures as outcomes, role-counts in the initial structure determine which branch will be taken.
2. Given an action transition $S_1 \xrightarrow{f_{Fa}} S_1^i \xrightarrow{up_a} S_2^i \xrightarrow{c} S_2$, the changes in role counts of every concrete structure represented by S_1 due to a are the same. This enables us to precisely represent the changes in role-counts caused by an action on an abstract structure.

Note that combining 2 with 1 above, we can easily find preconditions on a linear sequence of actions leading to a desirable branch by first computing the branch condition, and then inverting the effect of every action on the role counts involved in that condition.

In order to extend this idea to domains with binary relations, we will need some restrictions on these relations in order to make the results of focus operations categorizable in terms of role counts. Formally, we want certain relations to be *focus-classifiable* with respect to a chosen language, i.e, properties expressed using this language should be sufficient to determine what the result of a given focus operation will be, on a given abstract structure. In this paper, we use the language $\mathcal{E}_{\mathcal{R}}$ consisting of conjunctions of inequalities between constants and the counts of elements of roles coming from a set of roles $\mathcal{R}$. A generalization to more expressive languages is left for future work.

Focus classifiability will allow us to categorize branches caused due to the focus operation in terms of simple inequalities, as in the case of the first action in Fig. 8.

Definition 10 *(Focus Classifiability w.r.t. $\mathcal{R}$)* A focus operation f_F on a structure S satisfies *focus classifiability w.r.t.* $\mathcal{R}$ if for every $S_i \in f_F(S)$ it is possible to compute a constraint $l_j \in \mathcal{E}_{\mathcal{R}}$ such that for every $C \in \gamma(S)$, $C \in \gamma(S_i)$ iff $C \models l_j$.

Given focus classifiability, we need the ability to back-propagate constraints $l \in \mathcal{E}_{\mathcal{R}}$ through actions in order to express the conditions on an abstract structure under which an action branch occuring after multiple intermediate actions will be taken. We achieve this by formalizing property (2) of the simplified transport domain: we want actions to show *constant change* w.r.t. the set of roles $\mathcal{R}$ required for focus-classifiability.

Definition 11 *(Constant Change)* An action transition $S_1 \xrightarrow{f_{F_a}} S_1^i \xrightarrow{up_a} S_2^i \xrightarrow{c} S_2$ shows *constant change w.r.t. a set of roles* $\mathcal{R}$ iff there exists a constant δ_j for each $R_j \in \mathcal{R}$ such that whenever $C_1 \in \gamma(S_1^i), C_2 \in \gamma(S_2^i)$ and $C_1 \xrightarrow{a} C_2$, we have $\#_{R_j}(C_2) = \#_{R_j}(C_1) + \delta_j$.

With constant change and focus classifiability, we can compute preconditions for linear sequences of actions.

Definition 12 *(FC^3 Domains)* Let $\mathcal{S}$ be a set of abstract states closed under transitions for actions from a set $\mathcal{A}$ (i.e., if $S_i \in \mathcal{S}$ and $S_i \xrightarrow{a_1} \cdots \xrightarrow{a_k} S_f$ with $a_1, \ldots, a_k \in \mathcal{A}$, then $S_f \in \mathcal{S}$). $\mathcal{S}$ is an FC^3 domain[2] w.r.t. $\mathcal{E}_{\mathcal{R}}$ and $\mathcal{A}$ iff for every $S_1 \in \mathcal{S}$ and $a \in \mathcal{A}$, the transition $S_1 \xrightarrow{f_{F_a}} S_1^i \xrightarrow{up_a} S_2^i \xrightarrow{c} S_2$ shows constant change and its included focus operation f_{F_a} satisfies focus classifiability w.r.t. $\mathcal{R}$.

[2] FC^3 stands for "focus-classifiability and constant change"

We omit writing the set of actions $\mathcal{A}$ for an FC^3 domain when it is understood. We now prove that preconditions for reaching a particular abstract structure through a linear sequence of actions can be found in FC^3 domains. For convenience, we use the notation $S\|_l$ to denote the *refinement* of S such that $\gamma(S\|_l) = \{C : C \in \gamma(S) \wedge C \models l\}$.

Lemma 1 (Precondition for a single action) *Suppose* $S_1 \xrightarrow{f_{F_a}} S_1^i \xrightarrow{\text{up}_a} S_2^i \xrightarrow{c} S_2$ *is a transition in an* FC^3 *domain w.r.t.* $\mathcal{E}_\mathcal{R}$. *Then for every* $l_2 \in \mathcal{E}_\mathcal{R}$ *there is an* $l_1 \in \mathcal{E}_\mathcal{R}$ *such that for all* $C_1 \in \gamma(S_1)$, $C_1 \in \gamma(S_1\|_{l_1})$ *iff* $up_a(C_1) \in \gamma(S_2\|_{l_2})$.

Proof Since action f_{F_a} satisfies focus classifiability, there is a constraint l_i such that $C \in \gamma(S_1\|_{l_i})$ iff $C \in \gamma(S_1^i)$. We therefore need to compose l_i with a constraint for reaching $S_2^i\|_{l_2}$ to obtain l_1. This can be done by rewriting l_2's inequalities in terms of counts in S_1 since counts don't change during the focus operation from S_1 to S_1^i.

More precisely, suppose $\#_{R_j}(S_2^i) = \#_{R_j}(S_1^i) + \delta_j$ (we can write this expression because a shows constant change). Then we obtain the corresponding inequalities for S_1 by substituting $\#_{R_j}(S_1) + \delta_j$ for $\#_{R_j}(S_2^i)$ in all inequalities of l_2. Let us call the resulting set of inequalities l_1^i. Now l_1^i is satisfied by a $C_1 \in \gamma(S_1^i)$ iff $up_a(C_1)$ satisfies l_2. The conjunction of l_1^i and l_i thus gives us the desired constraint l_1. □

This method can be inductively extended to linear sequences of transitions:

Theorem 1 (Preconditions for a linear sequence of structures and actions) *Suppose we have a sequence of actions* $a_1, a_2, \ldots, a_n$ *such that* $S_1 \xrightarrow{f_{F_{a_1}}} S_1^i \xrightarrow{up_{a_1}} S_2^i \xrightarrow{c} S_2 \cdots \xrightarrow{c} S_n \xrightarrow{f_{F_{a_n}}} S_n^i \xrightarrow{up_{a_n}} S_{n+1}^i \xrightarrow{c} S_{n+1}$, *in an* FC^3 *domain. Then we can find a constraint* $l_{initial}$ *on* S_1 *such that a member* $C \in \gamma(S_1)$ *reaches* $S_{n+1}\|_{l_{\text{final}}}$ *along this path of transitions iff* $C \in \gamma(S_1\|_{l_{\text{initial}}})$.

5.1. Preconditions of Paths with Simple Loops

So far we dealt exclusively with finding preconditions over a linear sequence of actions. In this section we show that in FC^3 domains we can effectively propagate constraints back through paths consisting of simple (non-nested) loops (see Def. 9 on page 20), thus finding preconditions over simple loops of actions.

Let us consider the path of transitions from S to S_f including the loop in Fig. 9; analyses of other paths including the loop are similar. Each edge in the loop represents a transition with its specific focus branch and an action update. This is explicitly illustrated for action a_1 in Fig. 9. The restriction to simple loops therefore rules out cases where multiple branches resulting from an action's focus operation merge back into the loop. Analysis of such loops with internal branches is matter for future research.

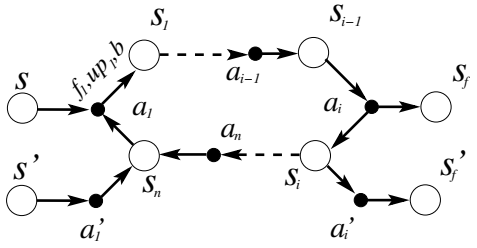

Figure 9: Paths with a simple loop. Outlined nodes represent structures and filled nodes represent actions.

Overview. Returning to Fig. 9, in order to find a constraint on the structure S which allows us to reach $S_f\|_{l_f}$, where l_f is a given constraint in $\mathcal{E}_{\mathcal{R}}$, we need to compute expressions for (1) the effect on role-counts after k iterations of the loop and (2) the conditions on S under which k iterations of the loop can be executed. The expression for (1) can be computed easily by adding the net change in role-counts due to each iteration of the loop. For (2), we need to ensure that in all the k different iterations, whenever an action has multiple possible branches, the branch that lies *in* the loop is taken.

Notation. Let the vector $\bar{R} = \langle \#R_1, \#R_2, \ldots, \#R_m \rangle$ consist of role-counts. Conceptually, in this vector we can include counts for all the roles; in practice, we can omit the irrelevant ones. Recall that in FC^3 domains every action satisfies constant change and every action branch can be classified in terms of inequalities between role-counts and constants. Let R_{b_i} be the branch role for action a_i, i.e., the role whose count determines the branch at action a_i (for simplicity, we assume that each branch is determined by a comparison of only one role with a constant; our method can be easily extended to situations where a conjunction of such conditions determines the branch to be followed). We use subscripts on vectors to denote the corresponding projections, so that the count of the branch-role at action a_i would be $\bar{R}_{b_i}$. If there is no branch at action a_i, we let $b_i = d$, an integer larger than m. Let Δ^i denote the role-count change vector for action a_i. Let $\Delta^{1..i} = \Delta^1 + \Delta^2 + \cdots + \Delta^i$. Let the initial role-count vector be $\bar{R}^0$, and the role-count vector after x complete iterations of the loop be $\bar{R}^x$.

Methodology. If we can assume that k iterations of the loop are completed starting with $\bar{R}^0$, the final role-count vector can be computed by adding the effect due to each iteration of the loop. In other words, we have $\bar{R}^k = \bar{R}^0 + k \times \Delta^{1..n}$. We now need to compute the conditions under which k complete iterations of the simple loop will be executed.

In the first iteration of the simple loop, in order to take the branch of action a_i that lies in the loop, we require the role-count R_{b_i} just before the application of a_i to satisfy an inequality with a constant. More precisely, we require ($\bar{R}^0 +$

$\Delta^{1..(i-1)})_{b_i} \circ c_i$, where $\circ$ is one of $\{>, =, <\}$ depending on the branch that lies in the loop and c_i is a constant.

Because the loop has n actions, the condition for a full execution of the loop starting with role-count vector $\bar{R}^0$ therefore is:

$$\begin{aligned} \bar{R}^0_{b_1} &\circ c_1 \\ (\bar{R}^0 + \Delta^1)_{b_2} &\circ c_2 \\ &\vdots \\ (\bar{R}^0 + \Delta^{1..(n-1)})_{b_n} &\circ c_n \end{aligned}$$

Let us call these inequalities $\mathsf{LoopIneq}(\bar{R}^0)$, so that $\mathsf{LoopIneq}(\bar{X})$ represents the condition for executing one complete iteration of the loop, starting with any m-dimensional role-count vector $\bar{X}$. Thus, for executing k complete iterations of the loop, we require:

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

These two conditions ensure all the intermediate loop conditions hold, because the changes are linear. For an exit during the $(k+1)^{th}$ iteration, we need the conditions for k complete iterations, and the conditions for the exit during the $(k+1)^{th}$ iteration:

$$\mathsf{LoopIneq}(\bar{R}^0) \wedge \mathsf{LoopIneq}(\bar{R}^{k-1}) \tag{2}$$
$$(\bar{R}^k)_{b_1} \circ c_1 \tag{3}$$
$$(\bar{R}^k + \Delta^1)_{b_2} \circ c_2 \tag{4}$$
$$\vdots \tag{5}$$
$$(\bar{R}^k + \Delta^{1..(i-1)})_{b_i} \bullet c_i \tag{6}$$

where in the last inequality, the "$\bullet$" corresponds to the condition for the branch that leaves the loop. These conditions capture exactly the conditions required for executing k complete and one partial iteration of the loop. This set of conditions assumes at least one full iteration; conditions for executing only a partial iteration of the loop can be computed by treating the partial loop segment as a linear segment of actions. Finally, we can express the role-count vector at the end of k complete and one partial iteration as:

$$\bar{R}^f = \bar{R}^k + \Delta^{1..i} \tag{7}$$

Algorithm 1 summarizes this process. Methods *ConstrucLoopIneq* and *ConstructPartialIneq* construct symbolic expressions for $\mathsf{LoopIneq}$ and the inequalities for the final, partial iteration respectively. *ComputeCumulativeChange* relies upon the ability to automatically compute the change in role-counts caused due to an action. A precise method for doing so is discussed in the next section, in Alg. 2.

The following theorem formalizes the result of the process for finding loop preconditions described above.

Algorithm 1: findLoopPreconditions

Input: Loop with actions $a_1, \ldots, a_n$, desired exit action a_f, desired final role-counts $\bar{F}$

Output: Preconditions $l_0(k)$ for reaching $\bar{F}$ immediately after exiting the loop during the $(k+1)^{th}$ iteration

```
for i = 1 to n do
    Δ^{1...i} ← ComputeCumulativeChange(i)
    Ineq_i ← ComputeRequiredBranchCondition(i)
LoopIneq ← ConstructLoopIneq(Δ^1, ..., Δ^{1..n}, Ineq_1, ..., Ineq_n)
LoopIneq_partial ← ConstructPartialIneq(Δ^1, ..., Δ^{1..f}, Ineq_1, ..., Ineq_f)
finalRCEq ← "F̄ = R̄^0 + k × Δ^{1..n} + Δ^{1..f}"
return LoopIneq[R̄^0], LoopIneq[R̄^{k-1}], LoopIneq_partial, finalRCEq
```

Theorem 2 (Preconditions of simple loops) *Suppose* $S_1 \xrightarrow{f_{F_{a_2}}} S_1^i \xrightarrow{up_{a_2}} S_2^i \xrightarrow{c} S_2 \cdots \xrightarrow{c} S_n \xrightarrow{f_{F_{a_1}}} S_1^i \xrightarrow{up_{a_1}} S_1^i \xrightarrow{c} S_1$, *is a simple loop in an* FC^3 *domain. Let the loop's entry and exit structures be* S *and* S_f, *such that* $S \xrightarrow{a_1} S_1$ *and* $S_i \xrightarrow{a_i} S_f$.

Alg. 1 returns a set of constraints $l_0(k)$ *such that for any* $C \in S$, *after* k *iterations of the loop and the simple path from* S *to* S_f, *the resulting structure* C_f *will be in* $\gamma(S_f \| _{l_f})$ *iff* $C \in \gamma(S \| _{l_0(k)})$.

Note that the final set of inequalities in the process described above includes the final values of role counts of all roles (R^f), parameterized by the number of iterations of the loop. Together with the ability to compute changes in role counts across linear sequences of actions (Thm. 1), Thm. 2 implies that in FC^3 domains we can compute whether a path of action transitions which is linear except for simple loops will take a concrete member of the initial abstract structure to a desired refinement $l \in \mathcal{E}_{\mathcal{R}}$ of the final structure. Further, these results allow us to compute the *exact* number of times we need to go around each loop in order to reach the desired structure with desired role counts.

These results can be extended to the more general setting of a graph of transitions, all of whose strongly connected components are simple loops. The precondition for reaching any desired structure from any initial structure can be computed as a disjunction of the preconditions for every path with simple loops from the initial structure to the desired structure. In this paper however, we focus on computing and analyzing plans in the simpler setting discussed above.

Example 5 Returning to the example in Fig. 8, let r denote the role-count of the role $\{obj, atL1\}$. We demonstrate the construction of the final set of equations (2-7) using the initial value r^0 alone, since this is the only role that classifies a branch in Fig. 8. Considering the left-most choice action as the first action in the loop, the general expression for r^k (the value of r after k complete iterations) is $r^0 - k$ since the net change in r due to 1 iteration of the loop is -1 (see the role-count changes listed on top of the figure). Eq.

(2) therefore amounts to $r^0 > 1$ and $r^0 + (k-1)(-1) > 1 \equiv r^0 > k$. For an exit during the $(k+1)^{th}$ iteration, corresponding to Eqn. (7), we require $r^0 + (k)(-1) = 1 \equiv r^0 = k+1$ since the loop exit condition requires the role-count of $\{obj, atL1\}$ to be 1.

To summarize, for k complete iterations and an exit during the $(k+1)^{th}$ iteration (along the only edge leaving the loop), we get the following conditions:

$$r^0 > 1; r^0 = k+1; r^f = r^0 - k = 1$$

In this example, r^f, the value of r after exit gets constrained to be exactly 1. The final values of other roles can be calculated simply by adding $k = r^0 - 1$ times the change caused due to a single loop iteration.

In order to compute the set of conditions we only need to compute at most n different $\Delta^{1..i}$ vectors. In our discussion so far, we assumed that these vectors, together with the constraints determining focus branches *can* be computed. The availability and efficiency of these operations ultimately determines the value of Theorem 2. The next section presents a class of domains where these operations can be conducted efficiently.

5.2. Sufficient Conditions for Obtaining FC^3 Domains

We now provide a set of sufficient conditions on abstract states and the syntax of action operations under which the FC^3 conditions are satisified. In domains satisfying these sufficient conditions, constraints determining focus branches, and role-count change vectors due to actions can be executed in time linear in the number of elements in the initial abstract structure.

We call a formula φ with a single free variable *role-specific* if it can only hold for objects of a certain specific role in a given structure. More formally,

Definition 13 *(Role-Specific Formulas)* A formula φ with a single free variable is *role-specific* in S iff there exists a role R such that for all $C \in \gamma(S)$ we have $C \models \forall x(\varphi(x) \rightarrow R(x))$, where we use $R(x)$ as an abbreviation for the conjunction of predicates in R together with literals denoting negations of abstraction predicates not in R.

The following proposition gives sufficient conditions for focus-classifiability. We call a formula "uniquely satisfiable" if it can hold for at most one element.

Proposition 1 (Sufficient conditions for focus-classifiability) *If φ is uniquely satisfiable in all $C \in \gamma(S)$ and role-specific in S then the focus operation f_φ on S satisfies focus classifiability w.r.t $\mathcal{E}_\mathcal{R}$.*

Proof Since the focus formula must hold for exactly one element of a certain role, the only branching possible is due to different numbers of elements satisfying the role while not satisfying the focus formula: either there is only one element of the role, and it satisfies the focus formula, or there is more than one element of that role and one of them satisfies the focus formula (see Fig. 5 on page 15). The branch is thus classifiable on the basis of the number of elements in the role ($= 1$ or > 1). □

This proof shows that when the premise of Prop. 1 is satisfied, the focus operation amounts to a comparison between the role-count of a role and the constant 1. This is the smallest number with which the comparison of a role-count can be useful; if a role is present in a structure, then we know that it must represent at least a single element.[3]

We can immediately extend Proposition 1 to a set of role-specific and uniquely satisfiable formulas as long as any pair of these formulas either always, or never coincide:

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.*

The condition of unique satisfiability on the focus formulas for actions (the $\Delta^{\pm}$ expressions) used in proposition 1 and corollary 1 also gives us actions with constant change:

Proposition 2 (Sufficient conditions for 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 uniquely satisfiable.*

PROOF Suppose $S_1 \xrightarrow{f_{F_a}} S_1^i \xrightarrow{up_a} S_2^i \xrightarrow{c} S_2$; $C_1 \in \gamma(S_1^i)$ and $C_1 \xrightarrow{up_a} C_2 \in \gamma(S_2^i)$.

For constant change we need to show that $\#_{R_i}(C_2) = \#_{R_i}(C_1) + \delta$ where δ is a constant. Recall that a role is a set of abstraction predicates. Furthermore, because the set of focus formulas f_{F_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 2 elements. The focused structure S_1^i shows exactly which elements undergo change, and the roles that they leave or will enter.

Therefore, since C_1 is embeddable in S_1^i and embeddings are surjective, the number of elements leaving or entering a role in C_1 is the number of those singletons which enter or leave it in S_1^i. Hence, this number is the same for every $C_1 \in \gamma(S_1^i)$, and is a constant determined by S_1^i. □

Since the required conditions in Proposition 2 are subsumed by those in Corollary 1, Corollary 1 provides sufficient conditions under which a focus operation on an abstract structure satisfies the FC^3 conditions of focus classifiability and constant change.

Therefore, if every abstract structure reachable from a given abstract structure S_{init} satisfies the conditions of Corollary 1 for every action possible on it, the space of reachable structures from S_{init} will constitute an FC^3 domain.

We call domains that satisfy Corollary 1 extended-LL domains because of their close relationship with linked lists in the abstraction.

[3] If we relax this notion and allow summary elements to potentially represent 0 elements, abstract structures become uninformative, stating, for every summary element, the tautology that there may or may not be an element with that summary element's role.

Definition 14 *(Extended-LL domains)* An *extended-LL domain* is a domain schema $\mathcal{D}$ with a start structure S_{start} such that all its actions' focus formulas F_{a_i} are role-specific, exclusive when not equivalent, and uniquely satisfiable in every structure reachable from a start structure S_{start}.

More formally, if $S_{start} \rightarrow^* S$ and $\Delta_i^{\pm}$ are the focus formulas, 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.

Note that if actions can be decomposed so that each action operator has only one focus formula, the restriction of the set of formulas being "exclusive when not equivalent" in Def. 14 becomes true trivially.

Intuitively, extended-LL domain schemas are those where the information captured by roles is sufficient to determine whether or not an object of any role will undergo change due to an action. Examples of such domains are linked lists, blocks-world scenarios, assembly domains where different objects can be constructed from constituent objects of different roles, and transport domains. In terms of computational expressiveness, extended-LL domains form a powerful class: in (Srivastava et al., 2010a), we show that actions in extended-LL domains are Turing-complete and therefore are sufficient for expressing any computational process, including any plan with PDDL actions. Note that finitary domains and extended-LL domains are distinct characterizations: finitary domains have a decidable halting problem and therefore constitute a practical class of problems. Our results for FC^3 domains however compute closed-form preconditions for a certain class of plans and do not make the finitary assumption.

In general, domains can be proved to be extended-LL domains by inductively proving the properties in Def. 14 for all the structures reachable from a given start structure. In practice, this can be proved more easily. In the delivery domain for instance, the only focus operations correspond to choice operations (which satisfy the extended-LL conditions: the choice formulas are defined to be unique and role-specific) and the focus operation for crate destinations, which are also role-specific to the role *loc* and constrained to be unique.

Theorem 3 (Sufficient conditions for FC^3 domains: extended-LL domains) *The space of all reachable states in an extended-LL domain constitues an FC^3 domain.*

5.2.1. Complexity of Finding Preconditions in Extended-LL Domains

Algorithm 2 shows a simple and efficient algorithm for computing the role changes due to an action on an abstract structure in extended-LL domains. While computing *count*, summary elements are counted as singletons. Changes computed in this way are accurate because in extended-LL domains, only singleton elements can change roles. The algorithm conducts $O(s)$ operations, where s is the number of distinct roles in the two structures.

Conditions classifying branches from a structure S can also be computed efficiently in extended-LL domains: we know all action branches take place as a result of the focus operation. The role(s) responsible for the branch will have

Algorithm 2: ComputeSingleStepChanges

Input: Action transition $S_1 \xrightarrow{f_{Fa}} S_1^i \xrightarrow{up_a} S_2^i \xrightarrow{c} S_2$
Output: Role-change vector Δ

$R \leftarrow$ roles in S_1^i or S_2^i
for $r \in R$ **do**
  $count_{old}(r)$= No. of elements with role r in $|S_1^i|$
  $count_{new}(r)$= No. of elements with role r in $|S_2^i|$
  $\Delta_r = count_{new}(r) - count_{old}(r)$

different numbers of elements in the focused structures prior to action update. Using a straightforward comparison of role counts, the responsible role and its counts (> 1 or $= 1$) for different branches can be found in $O(s)$ operations where s is the number of roles in S.

Using the algorithm for computing one step change vectors Δ^i (Algorithm 2), the constraints $l_0(k)$ representing preconditions of loops of transitions (Theorem 2) can be computed in $O(s \cdot n)$ time, where s is the maximum number of roles in a structure in the loop, and n is the number of actions in the loop.

5.3. Classical Unary Domains

We can now see the motivating example shown in Fig. 8 as a special case of extended-LL domains where all the predicates in the vocabulary are unary. We define *classical unary domains* as domain schemas with only unary predicates whose action updates can be represented using finite, but possibly conditional *add* and *delete* lists of properties.

More precisely, the action updates in classical unary domains are of the form:

$$up(p(x), a) \equiv \neg p(x) \wedge [\vee_{i=1..n}\{x = arg_i\} \wedge \Delta^+(x)] \quad (8)$$

$$\vee \quad p(x) \wedge \neg[\vee_{i=1..n}\{x = arg_i\} \wedge \Delta^-(x)] \quad (9)$$

This form of action updates restricts an action's effects to a finite set of action arguments. Such restrictions are common in classical planning problem descriptions where all the objects whose properties may be changed as a result of an action need to be provided as action arguments (hence the qualifier "classical" in the name for these domains).

Under canonical abstraction, such domains lose almost no information. Since we always draw-out action arguments prior to action application in the abstract state space, the update carried out by eqns (9) and (10) always shows constant change in the abstract state space (the reasons are similar to those in Prop.2). Action updates in classical unary domains require no focus operations - every formula evaluates to definite truth values since all the unary predicates are abstraction predicates. The only branches are caused due to the operations for drawing out action arguments. These operations use focus formulas constrained to be role-specific and uniquely satisfiable, and thus satisfy focus-classifiability (Prop. 1). This leads to the following theorem:

Theorem 4 (Classical unary domains are extended-LL domains)*All classical unary domains are extended-LL domains, and consequently, FC^3 domains.*

Many interesting problems can be translated into classical unary domains by creating an instance for every relation of arity k with all possible values for the other $k-1$ arguments: in the simplified transport domain introduced earlier in this section we constructed unary relations atL_i corresponding to the different possible locations. This process is at most as expensive as propositionalizing the relations, where the resulting vocabulary would have had a constant, (intead of a unary relation) for every relation and tuple of arity k (instead of tuples of arity $k-1$). Note that although this formulation would be more expensive than the original representation, it would still be more efficient than the propositional representation used by classical planners due to the use of canonical abstraction.

The limitation of this approach is that it does not allow generalization in the numbers of arguments which have been converted into relation instances (e.g. the locations in the transport example). Extended-LL domains are thus a strict generalization of classical unary domains, allowing us to represent problem domains like the delivery problem and blocks-world problems as described in the section on results.

6. Algorithms for Generalized Planning

In this section we describe algorithms for computing generalized plans using the representation and methods developed in the preceding sections. The computed plans will consist of linear sequences of actions separated by simple loops, allowing a direct application of Theorems 1 and 2 for computing their preconditions.

6.1. Plan Generalization

We present our approach for computing a generalized plan from a plan that works for a single problem instance in Algorithm 3. A preliminary version of this algorithm was described in (Srivastava et al., 2008). The input to Alg. 3 is a concrete example plan $\pi = (a_1, a_2, \ldots, a_n)$ for a concrete state C_0. Let S_0 be an abstract structure embedding C_0. In order to be able to find preconditions, S_0 should be such that the space of structures reachable from it constitutes an extended-LL domain. In our experience, the canonical abstraction of C_0 suffices; if the space of reachable structures is not extended-LL, loops can still be found, but their preconditions may not be computable using the methods developed in Section 5.

The idea behind Alg. 3 is to apply a given concrete plan in the abstract state space, starting with an abstract start state. This is done through a process called *tracing* (line 1). Because of abstraction, recurring properties become easily identifiable as repeating abstract states. Procedure *formLoops* uses these recurring identical structures to identify potential loops (line 2). *formLoops* returns a data structure representing all the loop positions and lengths; this

Algorithm 3: ARANDA-Learn

Input: $\pi = (a_1, \ldots, a_n)$: plan for C_0
Output: Generalized plan Π

1 SASequence $\leftarrow$ Trace(C_0, π)
2 loopSet$\leftarrow$ formLoops(SASequence)
3 $\Pi \leftarrow$ createGraph(SASequence, loopSet)
4 $S_f \leftarrow$ last structure in SASequence
5 **if** $\exists l \in \mathcal{E}_{\mathcal{R}} : S_f\|_l \models \varphi_{goal}$ **then**
6 $\quad l_\Pi \leftarrow$ findPrecon(S_0, Π, φ_g)
end
7 return Π_r, l_Π

is converted in a straightforward manner to a graph representation with nodes and edges by the subroutine *createGraph* (line 3).

If there is a constraint on the final abstract structure under which the goal formula is satisfied, then this is back propagated into a constraint on the initial structure in Π using methods described in Section 5. This is implemented in the *findPrecon* subroutine (line 6). Since the methodology for *findPrecon* has been discussed extensively in Section 5, we now provide a description of the subroutines *Trace* (listed on pg. 33) and *formLoops* (listed on pg. 34).

Procedure *Trace* takes as input, a concrete plan π and a concrete structure C_0 and returns a trace, or a sequence of abstract structures and actions (SASequence). In order to do so it first generalizes the choice actions in π (line 2). The generalized choice action selecting action a_i's k^{th} argument is specified using a formula capturing exactly the role of the element o_k chosen by the original choice action, in the preceding concrete state, C_{i-1}. The choice action is constructed as discussed in Section 4.2. The resulting sequence of actions is successively applied on concrete and abstract states, starting with C_0 and its canonical abstraction, S_0 (lines 3,4,5). After each action's application, the set of abstract structures obtained is traversed while searching for the one that embeds the corresponding concrete result (line 7). Since action updates on abstract structures capture all possible results, and the results of the focus operation are mutually inconsistent, exactly one such abstract structure will be found. This abstract structure becomes the next abstract structure in the trace, and the one on which next action operator will be applied. Once all actions have been applied and all the abstract structures capturing the observed concrete results at each step have been obtained, a sequence of (abstract state, action) tuples is returned.

The *formLoops* subroutine converts a sequence of structures and actions into a path with simple (i.e, non-nested) loops. The restriction to simple loops is imposed so that we can efficiently find plan-preconditions. More precisely, it returns a set of tuples consisting of the loopStart, loopEnd and loopExit indices in the input SASequence (computed by *Trace*). In each tuple, the segment of SASequence between loopStart and loopEnd denotes the body of a simple loop

Procedure `Trace`(C_0, π)

```
1  S_0 ← canon(C_0)
2  (a_1, ..., a_{n_c}) ← GeneralizeChoiceActions(π)
3  for i in [1, ..., n_c] do
4  |  C_i ← a_i(C_{i-1})
5  |  AbsStrucSet ← a_i(S_{i-1})
6  |  for S in AbsStrucSet do
   |  |  if C_i ⊑ S then
7  |  |  |  S_i ← S
   |  |  end
   |  end
   end
8  return SASequence ← (S_0, a_1), (S_1, a_2), ... (S_{n_c-1}, a_{n_c-1}), S_{n_c}
```

and the segment of SASequence between loopEnd and loopExit can be rolled into an iteration of this loop.

formLoops makes a single pass over the input sequence of abstract-state and instantiated action pairs while maintaining a look-up table, *Last*, for the last index where a particular (state, action) pair occurred. If the k^{th} element of SASequence matches its j^{th} element ($j < k$), then the index j is taken as the beginning of a loop (loopStart) and index k as its end (loopEnd). Such a repeated pair $(S_{j-1}, a_j) = (S_{k-1}, a_k)$ indicates that some properties that held in the concrete state after application of a_{j-1} were true again after application of a_{k-1} as witnessed by the fact that $S_{k-1} = S_{j-1}$, and further, that in the example plan, the same action $a_j = a_k$ was applied at this stage. This is our fundamental cue for identifying a sequence of actions that can be placed in a loop—as long as an identical abstract state can be reached again, the same actions can be applied. The subroutine *safeLoop* returns True iff the loop makes a net non-zero change for any role-count, determined by adding up all the role-count changes due to actions in the loop.

The elements between positions loopStart and loopEnd in SASequence constitute a single loop iteration. Once these positions are identified, further iterations of the loop are identified (lines 9-13). In order to do so, elements following SASequence[loopEnd] are matched with the corresponding elements in the newly identified loop, following SASequence[loopStart]. The *mod* operation (line 10) is used to roll back to the beginning of the loop in case multiple iterations occur after the loop is identified. The index after which elements of SASequence do *not* match the elements of the loop is identified as the loop's exit (line 12). Finally, the newly identified loop, characterized as (loopStart, loopEnd, loopExit) is added to the set of loops provided as input. The entire procedure then recurses on the segment of SASequence after loopExit.

Example 6 Consider the transport problem discussed in section 5. Fig. 10 shows a plan execution in the concrete state space. By adding a choice action

Procedure `formLoops`(*SASequence, loopSet* = {})

```
   /* SASequence=(S_0, a_1), (S_1, a_2), ... (S_{n_c-1}, a_{n_c-1}), S_{n_c}        */
 1 for sa in SASequence do
   |  Last[sa] ← −1
   end
 2 loopFound ← False
 3 for sa in SASequence do
   |  if Last[sa] > −1 and safeLoop((Last[sa], indexInSASequence(sa))) then
 4 |  |  loopStart ← Last[sa]
 5 |  |  loopEnd ← indexInSASequence(sa)
 6 |  |  loopFound ← True
 7 |  |  break /* exit the loop                                                   */
   |  end
   |  else
 8 |  |  Last[sa] ← indexInSASequence(sa)
   |  end
   end
   if loopFound then
   |  /* Extend the loop by capturing any subsequent iterations                  */
 9 |  i ← loopEnd; loopLength ← loopEnd − loopStart
10 |  while SASequence[i] = SASequence[loopStart +
   |  (i-loopEnd)mod(loopLength)] do
11 |  |  i ← i + 1
   |  end
12 |  loopExit ← i − 1
13 |  loopSet ← loopSet ∪ {(loopStart, loopEnd, loopExit)}
14 |  SASequence ← segment of SASequence after LoopExit
15 |  return formLoops(SASequence, loopSet)
   end
16 return loopSet
```

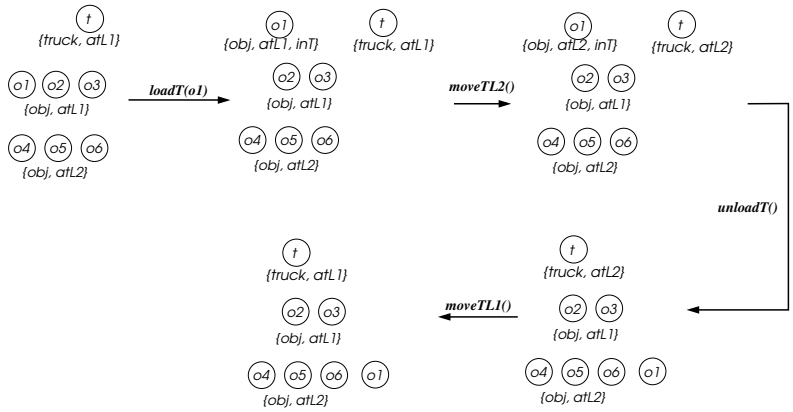

Figure 10: An example plan in the transport domain.

before the first *load* operation, and tracing out the plan on the canonical abstraction of the initial structure we get exactly the path shown in Fig. 8 on page 21. The included loop can be identified using *formLoops*, as described above.

7. Empirical Results

We implemented a prototype for ARANDA-Learn in Python, using TVLA as an engine for computing action results. We ran this implementation on some problems discussed in recent work on finding plans with loops. We discuss three of these problems in detail below; Tables 1, 2 and 3 summarize the results for the remaining problems which were originally proposed by Bonet et al. (2009). Further details of these problems, their representations and our solutions can be found in Appendix A. The class of initial instances for each of these problems was represented using a three-valued structure. All problems except for the green block problem in Appendix A constitute extended-LL domains. The green block problem includes a non-deterministic sensing action for detecting the color of a block, and demonstrates how our methods work on situations where we do not have focus-classifiability.

Table 1 shows a comprehensive summary of the preconditions for the generalized plans found for these problems, and the start structures on which they apply. The l_i variables in this table correspond to the number of iterations of the i^{th} loop in the generalized plan, where the numbering begins from the terminal node. Timing results for different phases of plan generalization, and for precondition evaluation are shown in Table 2. Table 3 shows the quality of the computed generalized plans along the evaluation measures developed in this paper.

Delivery

We implemented the non-deterministic version of the delivery problem with a sensing action *findDest* for one truck. The action and vocabulary for this problem were defined in Sections 2 and the non-deterministic, sensing aspects were described in Section 4.3. Because of the restriction to a single truck, the *Move* and *Load* actions require only one argument representing the destination and the crate to be loaded respectively; the *Unload* action does not require arguments, and the predicate *in* becomes unary and holds for the object currently in the truck. The input example plan delivered five objects to two different locations. The abstract start structure is shown in Fig. 11. ARANDA-Learn found the generalized plan shown in Fig. 12. Since the delivery domain is an extended-LL domain, we can use the methods described in Section 5 to compute the preconditions for this plan as $\#(item) \geq 3$. In fact, here and in all the following examples the preconditions also show how many loop iterations there will be in a plan execution (e.g. $\#(crate) = l + 3$, where $l \geq 0$ is the number of loop iterations).

Problem	Start Structure	Preconditions on Start Structure
Delivery	Fig. 11	$\#\{item\} = 3 + l_1; \#\{loc\} > 1$
Trucks	Fig. 14	$\#\{monitor, atL2\} = 2 + l_1;\ \#\{server, atL1\} = 2 + l_1$
Blocks	Fig. 15	$\#\{Blue, misplaced\} = 2 + l_3$ $\#\{Red, misplaced\} = 3 + l_2$ $\#\{Red, misplaced, onTable, topmost\} = -1 + l_1 - l_2$ $\#\{Blue, misplaced, onTable, topmost\} = l_1 - l_3$
Green Block	Fig. A.17	$\#\{\} = 2 + l_1$
Hall-A	Fig. A.18	$\#\{wborder\} = 4 + l_1;\ \#\{nborder\} = 4 + l_2$ $\#\{eborder\} = 4 + l_3;\ \#\{sborder\} = 4 + l_4$
Prize-A(5 rows)	Fig. A.19	$l_1 = l_2 = l_3 = l_4 = l_5; \#\{dFromE\} = 3 + l_6$ $\#\{dFromW\} = l_5 - l_6; \#\{dFromN\} = 5$
Prize-A(7 rows)	Fig. A.19	$l_1 = l_2 = l_3 = l_4 = l_5 = l_6 = l_7; \#\{dFromE\} = 3 + l_8$ $\#\{dFromW\} = l_7 - l_8; \#\{dFromN\} = 7$
Corner-A	Fig. A.20	$\#\{dFromN\} = 2 + l_1; \#\{dFromE\} = 3 + l_2$

Table 1: Preconditions for example problems. l_i denote the number of iterations of loop i in the corresponding plan; preconditions for the Green Block problem are necessary, but not sufficient.

Problem	Tracing	Loop Finding	Computing Preconditions	Total
Delivery	66.12	3.93	0.01	70.05
Trucks	85.79	2.94	0.01	88.74
Blocks	65.01	2.04	0.02	67.06
Green Block	17.04	0.52	0.00	17.56
Hall-A	32.30	1.89	0.01	34.19
Prize-A(5 rows)	35.73	0.51	0.01	36.24
Prize-A(7 rows)	47.93	0.79	0.02	48.73
Corner-A	6.94	0.04	0.00	6.98

Table 2: Timing results for ARANDA-Learn. All results in seconds; runs were carried out on a Linux machine with an Intel Core2 Duo 1.6GHz processor and 1.5GB RAM.

Problem	Domain Coverage	Applicability Test	Instantiation Cost	Quality
Delivery	≥ 3 items	$O(n)$	$O(n)$	1
Trucks	≥ 2 pairs	$O(n)$	$O(n)$	$\frac{9p}{11p} = 0.81$
Blocks	≥ 4 pairs	$O(n)$	$O(n)$	1
Green Block	≥ 4 blocks	$O(n)$	$O(n)$	1
Hall-A	≥ 6 segments/wing	$O(n)$	$O(n)$	1
Prize-A (5)	$5 \times (3 + l)$ grids	$O(n)$	$O(n)$	1
Prize-A (7)	$7 \times (3 + l)$ grids	$O(n)$	$O(n)$	1
Corner-A	$(2 + l) \times (3 + k)$ grids	$O(n)$	$O(n)$	1

Table 3: Evaluation of some generalized plans. n denotes the size of the problem instance and l, k are variables ≥ 0. See Sec. 7.2 for discussion of quality of the Trucks solution.

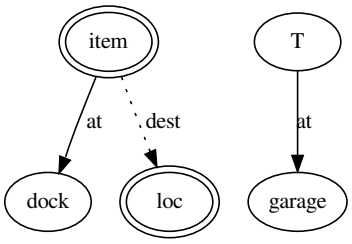

Figure 11: Abstract start structure for the Delivery problem

Trucks
Vocabulary: {*Monitor*, *Server*, *T1*, *T2*, *atL1*, *atL2*, *inT1*, *inT2*}
Actions: {$LoadT_i(x)$, $UnloadT_i()$, $GoToL_j\,T_i()$}

This is a problem from the transport domain. We have two source locations L1 and L2, which have a variable number of monitors and servers respectively (Fig. 14). There are two trucks, $T1$ at L1 and $T2$ at L2 with capacities 1 and 2 respectively. The generalized planning problem is to deliver *all*—regardless of the actual numbers—items to L4, but only in pairs with one item of each kind.

We represented this domain without using any binary relations, as a classical unary domain. Fig. 14 shows initial abstract structure used for tracing. The example plan for six pairs of such items worked as follows: $T1$ moved a monitor from L1 to L3 and returned to L1; $T2$ then took a server from L2 to L4, picking up the monitor left by $T1$ at L3 on the way. Fig. 13 shows the main loop discovered by our algorithm. The computed preconditions for the final generalized plan are shown in Table 1, and constrain the counts of servers and monitors to be equal, and at least 2.

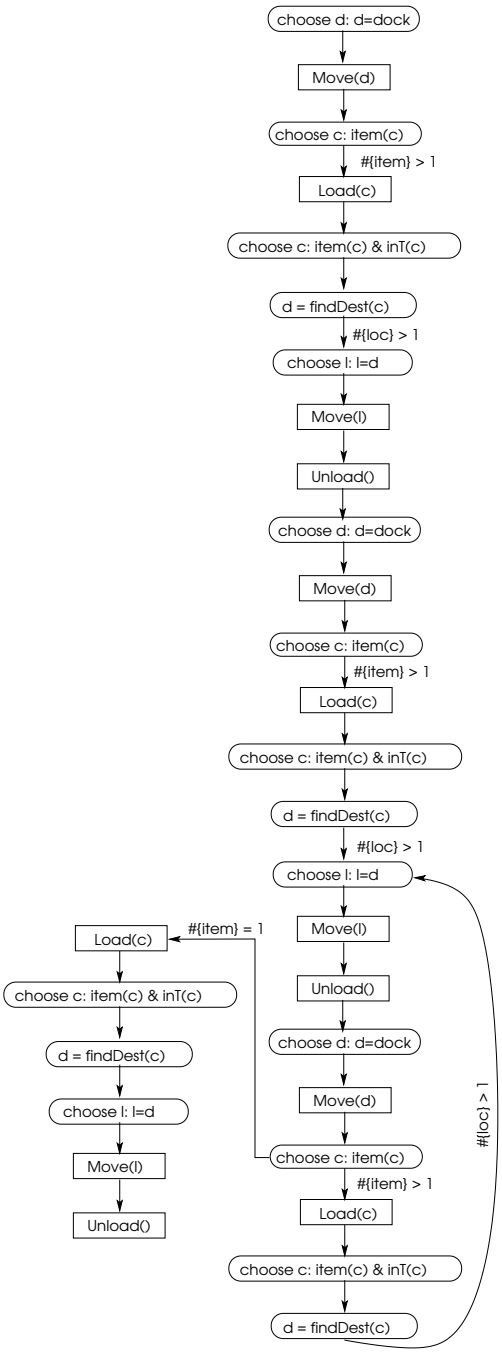

Figure 12: Generalized plan for unit delivery problem instances with at least 2 crates.

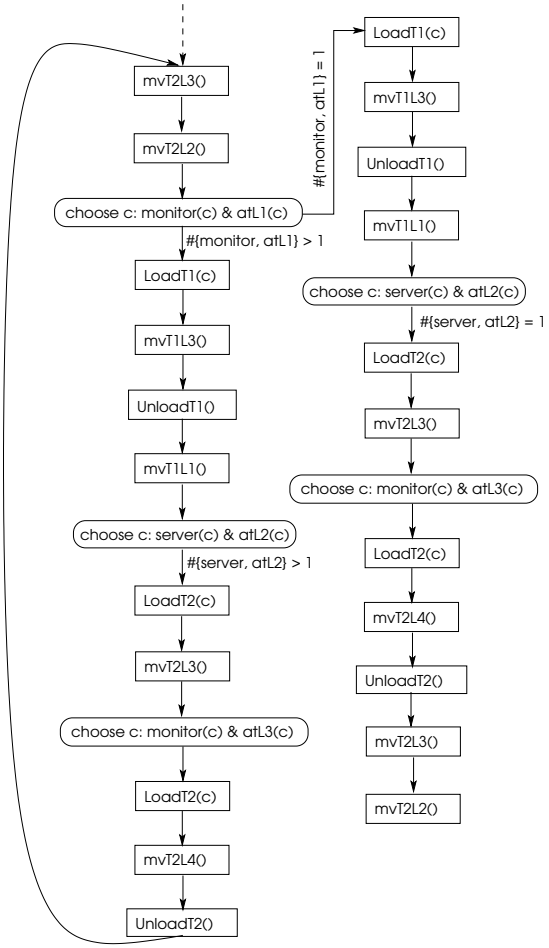

Figure 13: Main loop for Trucks

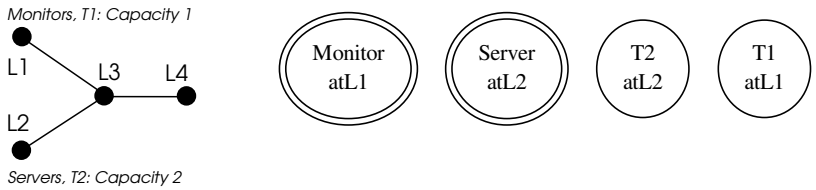

Figure 14: Map and the start structure for Trucks

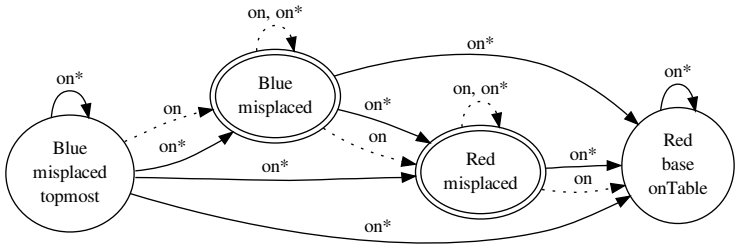

Figure 15: Start structure for striped block tower

Striped Block Tower
Vocabulary: $\{Red^1, Blue^1, base^1, onTable^1, on^2, topmost^1, on^{*2}, misplaced^1\}$
Actions: $\{Move(x, y), moveToTable(x)\}$

Given a tower of red and blue blocks with red blocks at the bottom and blue blocks on top, the goal is to find a plan that can construct a tower of alternating red and blue blocks, with a red "base" block at the bottom and a blue block on top. We used transitive closure of the "on" relation, on^*, to express stacked towers and the goal condition.

Fig. 15 shows the abstract initial structure. The *misplaced* predicate is used to determine if the goal is reached. *misplaced* holds for a block iff either it is on a block of the same color, or above a block which is on a block of its own color.

The input example plan worked for six pairs of blocks, by first unstacking the whole tower, and then placing blocks of alternating colors back above the base block. Our algorithm discovered three loops: unstack red, unstack blue, stack blue and red (Fig. 16). The preconditions shown in Table 1 describe possible role-counts at the start structure. These conditions capture a more general situation where the start structure may have some blocks on the table, corresponding to the roles $\{Red, misplaced, onTable, topmost\}$ and

$\{Blue, misplaced, onTable, topmost\}$. If we set these quantities as zeros, we get $l_1 = l_3$ and $l_1 = l_2 + 1$, which constrain the number of red and blue blocks in the initial stack should be equal. Further, the number of blue blocks should be $3 + l_2 + 1$, counting the blocks with roles $\{Blue, misplaced\}$ and one extra block with the role $\{Blue, misplaced, topmost\}$.

7.1. Summary of Timing Results

Timing results for all the test problems are shown in Table 2. Although the entire process of tracing can be understood as contributing to the information utilized for computing preconditions, once the role-count changes due to action operations have been computed, the time required for computing preconditions for the obtained generalized plan is negligible.

Many optimizations are possible on our protoype implementation of the presented algorithms. Our implementation is written in Python, which is an interpreted language. Faster results can be obtained from an implementation in a compiled language. Profiler outputs show that most of the time is spent in calls to TVLA and in python's module for adding or removing edges from graphs that we use to implement first-order structures. Optimization of these data structures can also improve the run times.

7.2. Evaluation of the Obtained Plans

Table 3 shows an evaluation of the generalized plans found by ARANDA-Learn, and described in the previous section. Testing for applicability requires only counts of elements of different roles in the start structure. Table 3 lists this cost as $O(n)$ but it can be reduced to a constant number of numeric comparison operations if these counts are provided with the initial concrete state.

Plan instantiation cost is always $O(n)$ because we find plans with simple loops, all of which reduce the count of some role(s) and thus can be iterated at most $O(n)$ times. Each iteration has a constant number of choice operations, and each of these can be executed in constant time by maintaining look-up tables containing elements of each role, as a part of the action updates.

We use the ratio of the length of an instantiation for a problem instance of size n with the length of the optimal plan for that size as a measure of the quality of the generalized plan. All the obtained plans except for Trucks execute a minimal number of operations and are optimal. Plan quality for Trucks is less than 1 because our plan uses both vehicles; the fewest actions are used if only the Truck is used for all transportation, in which case a problem instance with p pairs of deliverables is solved in $9p$ actions. On the other hand, the obtained plan has a better makespan.

For all the test problems, the computed generalized plans solve all problem instances above a certain lower limit on the size; the asymptotic domain coverage of all the generalized plans is maximal (one). Typically, the unsolved instances are small and have at most four elements per summary element in the initial abstract structure. While these instances could easily be solved using classical planners, we present more general methods for extending the applicability of partial generalized plans in (Srivastava et al., 2010b).

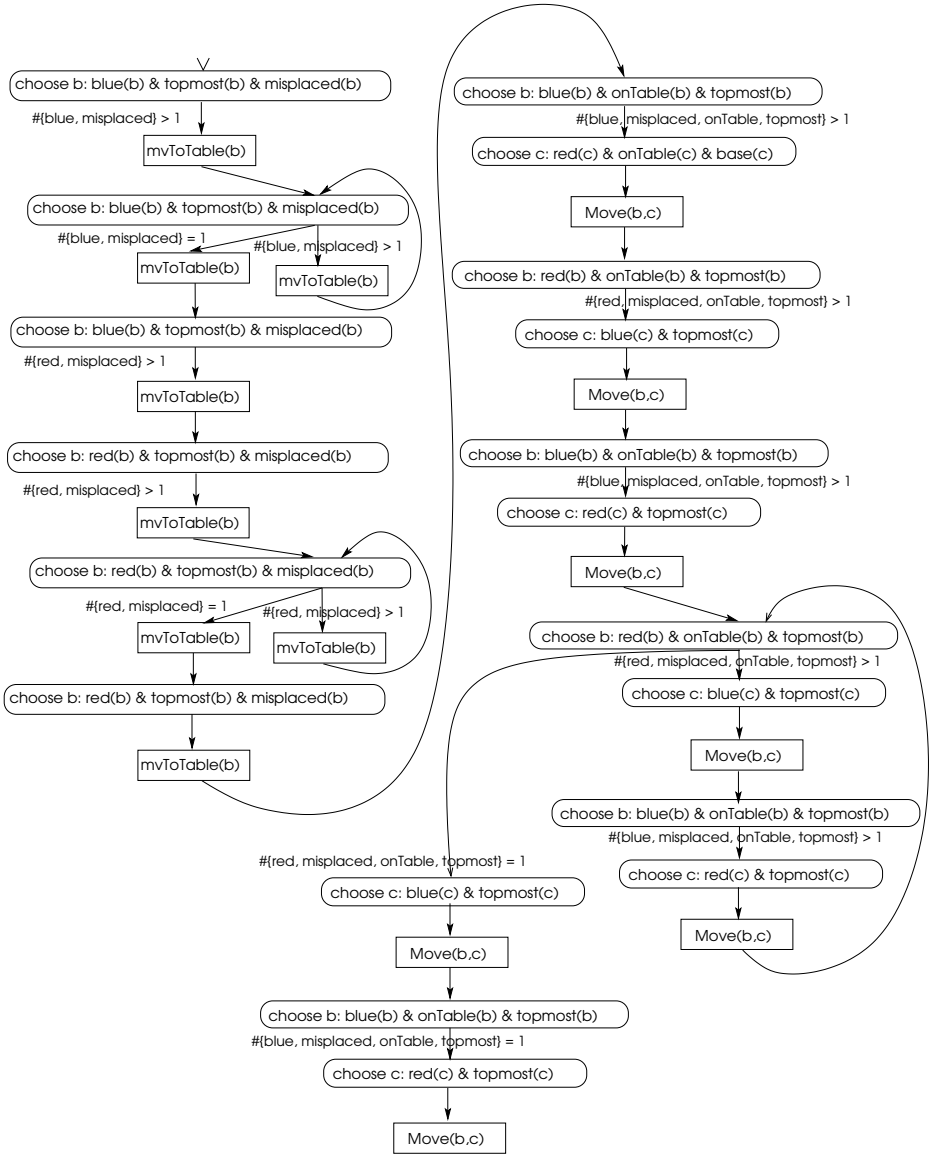

Figure 16: Generalized Plan for Striped Block Tower. In choice actions, only the predicates belonging to the role being chosen are shown.

7.3. Observations and Key Features of the Results

The empirical results presented above and in Appendix A share some key features which can inform the choice of input example plans for ARANDA-Learn and also provide opportunity for subsequent extensions of these plans. We discuss these features below.

Length of Input Plans. Our approach for identifying useful loops in example plans rests on being able to find recurring abstract structures which represent invariants of the loop's execution (e.g., Fig. 8 on page 21).

In order to express a loop's invariant (and consequently, to identify the loop) in terms of an abstract structure, all the roles which gain or lose elements due to an iteration of that loop must be summary elements in the abstract structure. If this is not the case, and we have an initial abstract structure with a singleton element for a role R_1 being affected by an iteration of the loop, an application of the loop will result in a different abstract structure in which role R_1 will either be represented by a summary element or will be entirely absent.

As long as any sequence of actions leads to such an invariant state, our approach can identify a subsequent sequence of actions which returns to this invariant state as a potential loop. In most of our examples, this can be achieved if the concrete instance solved by the example plan has at least 5 elements for every summary element of the initial abstract structure. With this number it is possible to reach, and subsequently revisit an invariant state: the first two transfers of elements lead to the first occurrence of an invariant state with two elements each in the roles which gain or lose elements; the next sequence of operations which cumulatively makes a similar transfer will reach the abstract invariant state again, leading to the identification of a potential loop.

In the initial abstract structure for the delivery problem for instance (see Fig. 11 on page 37), we need at least five different elements with role $\{item\}$; the role-count of $\{loc\}$ is never changed (no action changes the properties of location elements) so that any number of locations greater than 1 (to necessitate a summary element) suffices. A more efficient approach for future work could be to trace the example plan on an abstract structure which already has summary elements for all the affected roles and is thus closer to the invariant state.

Computed Generalized Plans. The output generalized plans shown in Figs. 13 and 16 include some "unrolled" iterations of each loop. These iterations were not merged into the loop because the abstract structures in these segments could not be embedded into the abstract structures within the loop—recall that these structures are computed during tracing and represent the set of concrete states possible at any step in the generalized plan.

This representation has the drawback that the unrolled iterations make outputs longer and harder for users to understand. The output plans could be made more readable while also retaining correctness and the ability to compute preconditions, through post-processing steps: once the role-count changes have been computed, abstract structures can be discarded and every loop's preceeding and succeeding actions can be merged with the loop whenever the role-count

changes of those actions match with the actions within the loop. However, we do not currently perform these operation as they sacrifice the potential for automatically extending the computed generalized plans using their abstract structures (Srivastava et al., 2010b). Development of methods for improving output quality without losing this capability is left for future work.

8. Related Work

There have been very few directed efforts towards developing generalized plans with the capabilities we demonstrate. Repeated effort for solving similar problems was identified as a serious hurdle to a practical application of planning almost as soon as the first modern planners were developed (Fikes et al., 1972). Various planning paradigms have since been developed to handle this problem by extracting useful information from existing plans. However, to our knowledge no approach addresses all the challenges in generalized planning described in the introduction. In this section we discuss other work on finding plans that could be understood as generalized plans. We also discuss related uses of abstraction, both in planning and software model checking.

Abstraction in Planning. Our approach uses abstraction for *state aggregation*, which has been extensively studied for efficiently representing universal plans (Cimatti et al., 1998), solving MDPs (Hoey et al., 1999; Feng and 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.

Hoffmann et al. (2006) study the use of abstraction for STRIPS-style classical planning. They prove that for a wide class of abstractions motivated by those used for evaluating heuristics in planning, searching over the abstract state space cannot perform better than informed plan search (using heuristics or resolution based search). We use abstraction to solve a different problem, that of observing the effect of a plan on a set of distinct problem instances with varying numbers of objects; further, our abstraction is not propositional and does not satisfy some of their planning graph based requirements.

Explanation Based Learning. In explanation based learning (EBL) (Dejong and Mooney, 1986), a proof or an explanation of a solution is generalized to be applicable to different problem instances. A domain theory is used to generate the required proof for a working solution. The BAGGER2 system (Shavlik, 1990) extended this paradigm by generalizing the structure of the proofs themselves. Given a hand-coded domain theory including the appropriate looping constructs, it could identify their iterations in proofs of working plan instances, and subsequently generalize them to produce plans with recursive or looping structures. However, the BAGGER2 system does not address the problem of proving termination for its output plans.

Plans With Loops. Two recently proposed approaches to finding plans with loops share many of the objectives of the approach presented in this paper. KPLANNER (Levesque, 2005) proceeds by iteratively finding plans for small problem instances and identifying recurring patterns which can be placed into loops. However, KPLANNER only identifies loops that generalize a single numeric *planning parameter*. Levesque notes that "even short iterative programs can be quite difficult to reason about". He concludes that "faced with an intractable reasoning problem, we can look for compromises. ... [and] forego the strong guarantees of correctness". In contrast with other existing approaches, KPLANNER addresses the problem of developing applicability tests by guaranteeing that the computed plans work across a user-supplied interval of values for the planning parameter.

DISTILL (Winner and Veloso, 2003, 2007) is a system for learning domain specific planners (dsPlanners) through examples. DISTILL uses partially ordered example plans with annotations reflecting every operator's needs and effects. These annotation are used to compile parametrized versions of example plans into a *dsPlanner*, which consists of a sequence of statements like *if...then...else* and *while()*{..}. At any stage during execution, the conditions of multiple such statements may be true; the execution model of dsPlanners is to always execute the first such statement in the dsPlanner. While DISTILL ensures that if a loop's condition is true then one iteration of the loop *will* be executable, it does not guarantee goal reachability, or address the problem of developing efficient applicability tests: the applicability test for a dsPlanner requires a simulated execution until none of the steps in the plan can be applied.

Approaches for strong cyclic planning (Cimatti et al., 2003) aim to generate plans with loops for achieving temporally extended goals and for handling actions which may fail. Although strong cyclic plans include loops, their objectives and motivations differ significantly different from those of the presented work. The utility of loops in strong cyclic plans lies in being able to repeatedly return to a previous state, from where a sequence of actions with a chance of success can be re-applied; plans with loops are only found if no acyclic plan can solve the given problem.

In contrast, our objective is to construct loops whose iterations make measurable, incremental changes leading to a goal. Execution of these loops never revisits a concrete state. In fact, linear plans exist for all the problem instances we consider, but are less desirable than plans with loops due to their much smaller domain coverage.

Programming By Demonstration. The objective of learning loops by generalizing concrete plans is similar to the objectives of programming by demonstration (PBD). In practice, approaches for PBD follow very different assumptions compared to those in the field of AI Planning. Approaches like (Lau et al., 2003) address the significantly different problem of using a given segment of a user's actions (e.g. keystrokes in a text editing task) to predict the remainder of the program being executed. In this approach, loop iterations in training examples are explicitly annotated by the user. The SHEEPDOG (Lau et al., 2004) system

on the other hand uses an extension of Hidden Markov Models to predict the next most likely action required during a technical support task. Instead of computing the preconditions for their learned structures, both of these systems provide probabilistic quality and usability guarantees. Such guarantees can be useful in many settings, particularly those where a limited domain theory prevents precondition analysis. The PLOW system (Allen et al., 2007) also captures loops via demonstration, but uses a mixed-initiative approach where the user provides cues to the system for beginning a loop recognition process, proactively corrects the system's errors while demonstrating a solution and provides explicit loop termination conditions.

A related area of research is workflow inference, where actions are replaced by functions whose inputs and outputs are data-collections. Approaches for workflow inference like LAPDOG (Eker et al., 2009) and WIT (Yaman and Oates, 2007) learn loops of actions from example traces but fundamentally differ from planning in the notion of actions: in effect, action outcomes of workflow actions, which amount to data or information, are never "deleted". This implies that an observed sequence of actions can always be repeated. This allows WIT to work without any information about action preconditions and effects: action occurrences in an observed trace can be treated like alphabets in a problem of grammar induction. In planning however, actions regularly remove facts on which successive actions, or loop iterations may depend. Our approach represents summarized information about the states possible after an action application using abstract states. This information captures action effects and allows us to determine when (and how many times) a loop of actions may be executed.

Policies and Plans. Fern et al. (2006) present an approach for developing general policies which can be used over a wide variety of problem instances. Their approach however does not aim to produce algorithm-like plans and requires intensive initial training. Their policies also do not come with applicability tests.

Contingent Planning. Contingent planning (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005) can be seen as an instance of generalized planning where the class of initial instances represents the set of possible initial states. Contingent planners already use state abstraction to represent sets of possible states (world states) as belief states. Sensing actions serve to divide a belief state in terms of the truth value of the proposition(s) being sensed.

However, existing contingent planners expect a finite set of initial states and cannot model belief states with unknown quantities of objects. Existing representations of contingent plans are also limited, leading to tree-structured solutions exponential in the number of objects. As discussed in the introduction, this tends to increase the computational complexity of finding the desired contingent plan.

Software Model Checking. Software model checking is the problem of verifying that the behavior of a formal specification or program satisfies desirable proper-

ties. In general, such properties may range over segments of execution. Software model checking literature consists of a wealth of different abstraction techniques for effectively capturing state properties and soundly updating states as a result of program steps. More recent approaches employ automated abstraction refinement to increase the precision of abstraction in the branch of execution where a possible counterexample to correctness is found (Henzinger et al., 2002).

Methods such as Terminator (Cook et al., 2006) use linear inequalities to represent changes caused due to loop iterations. For a class of programs with while loops, Terminator is guaranteed to find a measure of progress, if one exists in the form of a linear ranking function. The measure of progress proves that the loop will eventually terminate; such methods can be used to provide weaker guarantees of correctness in our framework such as the absence of non-terminating loops. Terminator's use of linear inequalities for representing changes caused by program statements are also suitable for representing the changes in role counts.

9. Conclusion

Contributions. In this paper, we presented a formal notion of generalized planning with a set of fundamental measures for evaluating generalized plans along different dimensions of utility. We introduced tools and techniques motivated by software model checking for addressing the problem of finding provably correct generalized plans. We used an abstraction method from the TVLA system to develop a sound approach for the construction of generalized plans with loops using example classical plans. The resulting framework allows us to (a) compactly represent sets of states with unknown, and unbounded quantities of objects, (b) recognize recurring state properties in the search for segments of classical plans which could potentially be useful as loops of actions, and, (c) efficiently compute plan preconditions for use as efficient applicability tests. From a broader perspective, although the general problem of determining when a loop of actions will even *terminate* is undecidable, we presented a novel approach for computing the conditions under which a class of plans with loops will not only terminate, but also lead to a desired goal state.

In conclusion, perhaps the most significant property of the presented approach is that it offers an efficient method for computing plans with simple loops *as well as* characterizing their applicability on unbounded classes of problem instances.

Limitations and Future Work. The presented approach is based on abstraction in terms of unary predicates. In some situations, a domain's unary predicates may not capture all the properties necessary for determining the progress made by loops. This problem involves a trade-off between decreasing the granularity of abstraction for increased precision in inference on the one hand, and maintaining a tractable reasoning process on the other. This is a broad and well-studied problem in the model checking literature, with two categories of approaches: (1) the addition of new defined, or *instrumentation* predicates whose update formulas can be computed automatically (Reps et al., 2003), and (2) dynamically

refining the abstraction by adding new predicates as needed (Henzinger et al., 2002). Both of these approaches could be applied in the context of identifying loops that make measurable progress in our approach.

At present, our approach is sensitive to the initial concrete instance on which the example plan is applied. As discussed in Section 7.3, modifying the initial concrete and abstract structures used for tracing could allow us to find plans that are more compact, and also more general. Another direction for future work could be a more streamlined representation of the computed plans, which could make the plans intuitively more easy to understand. The presented algorithm for identifying loops works in a greedy fashion by always accepting the first terminating loop that it finds. A more exhaustive search could be conducted for maximizing a specific measure of plan quality, such as domain coverage.

The presented approach constructs generalizations of single examples. In related, ongoing work, we have made some initial progress in generalizing and merging useful segments from multiple classical plans (Srivastava et al., 2010b) into a coherent generalized plan. This process uses abstract structures stored in our generalized plans to identify situations where segments of additional classical plans could be applicable. In practice, this approach can also be used to extend the coverage of plans computed in this paper, to include all the small problem instances that are currently not covered.

The focus of algorithms in this paper was on methods for finding generalizations of classical plans. However, the approach for finding preconditions of plans with simple loops of actions can also be used to conduct a direct search for generalized plans. This can be done by searching for paths to goal states like classical planners, with two exceptions: first, the search would be conducted in an *abstract* state space using the abstract action application described in this paper; second, the search would also include paths with cycles that can be determined to make progress. Once a path to the goal is found, its preconditions can be computed. Search for more paths to the goal would continue until a desired coverage is reached. This algorithm could be implemented in an anytime fashion, with solution quality improving over time in terms of domain coverage. The smaller size of abstract state spaces in comparison to concrete state spaces for large numbers of objects makes this approach viable. In practice, the search process can be guided with heuristic functions for efficiency.

Efficient representations of preconditions of nested loops and the extension of our methods to wider classes of domains are also natural directions for future research on the more fundamental questions addressed in this paper. For the latter, the combination of features of unary and extended-LL domains is a promising next step. This direction of research can also benefit from methods used in generation of ranking functions for loops (Podelski and Rybalchenko, 2004).

Acknowledgments

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

Appendix A. Detailed Empirical Results

Note on Problem Domains. The Hall-A, Corner-A, Prize-A and GreenBlock problems were first discussed by Bonet et al. (2009), under a framework for partial observability where observations are automatically triggered when the real states generating them are reached. This formulation thus does not require sensing actions. Although this is a very different formulation from ours, the problems remain meaningful and interesting in our setting as well. In most cases, the abstraction we used to represent the multiple possible initial states corresponded with the belief states used by Bonet et al. to reflect partial observability. In general, their solutions are much more compact than ours - this is expected, as we restrict our implementation to find only simple loops.

We restrict our evaluation to the focus of this paper, which is on plan generalization and precondition computation; a more detailed evaluation of the capabilities of our approach for conditional planning using sensing actions is beyond the scope of this paper.

Green Block
Vocabulary: $\{topmost^1, onTable^1, on^2, on*^2, isGreen^2\}$
Actions: $\{unstack(), senseColor(x), collect(), discard()\}$

The Green Block problem is to find a green block in a stack of blocks. We formulate this problem using a sensing action to determine the color of the topmost block (cf. note on problems above). Fig. A.17 shows the abstract initial structure. We use the *isGreen(arg,x)* predicate in order to implement the sensing action for block x's color using the focus operation.

The unstack action places the topmost block into the gripper; the senseColor action senses the color of the topmost block; the collect action collects the block in the gripper and the discard action discards it. The color of a block is visible only while the object is on the stack, and is obscured when the block is in the gripper.

This problem domain does not belong to the extended-LL class because its goal depends on a sensing action whose result cannot be predicted based on role-counts alone. However, ARANDA-Learn still computes a determininstic generalized plan that can be proved to work for all but the smallest instances (with fewer than 4 blocks, see Table 1 on page 36) of the green block problem.

The smallest example plan in which a loop could be recognized found a green block and collected it after discarding 3 non-green blocks. The computed generalized plan recognizes the loop with an exit when the topmost block's color is sensed to be green.

The learned generalized plan is thus correct and deterministic, but its preconditions are not expressible in terms of the counts of available roles. Because of this, the preconditions we obtain are necessary, but not sufficient.

Hall-A
Vocabulary: $\{e^2, n^2, e^{*2}, n^{*2}, wborder^1, nborder^1, eborder^1, nborder^1, visited^1\}$
Actions: $\{mvE(), mvW(), mvN(), mvS()\}$

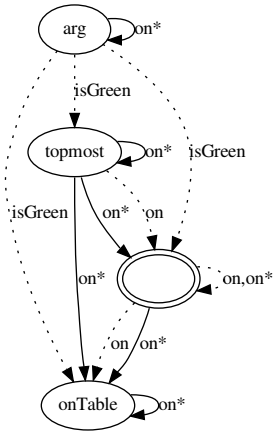

Figure A.17: Initial abstract structure for the green block problem.

The problem domain consists of 4 hallways arranged to form a quadrilateral. Each hallway is segmented into multiple segments denoting room boundaries which have to be traversed to cross the hallway (see Fig. A.18, left); the problem is to find a plan for visiting all four corners and returning to the starting point, for an agent starting at a given corner.

The e and n relations represent the east and north relations between hall segments and the e^* and n^* relations, their corresponding transtive closures. The *visited* relation is used to determine the goal condition and $x - border$ predicates define different hallways. The smallest example plan from which our approach could identify loops solved this problem for a square arrangement with 7 segments in each hall. The canonical abstraction of this initial state, used as the initial abstract structure during tracing, is shown in Fig. A.18.

The computed generalized plan consists of four loops, and its preconditions (Table 1 on page 36) show that it can solve any quadrilateral (as opposed to rectangular) arrangement of halls with at least 6 segments in each hallway.

Bonet et al. (2009) formulate this problem with an initial belief state of a fixed size. The controller learned by their approach can be seen to work for halls of any dimensions with the agent starting at any of the hallway segments, but this fact is not discovered automatically. In our approach, the initial belief state itself generalizes the problem to quadrilaterals with sides of arbitrary lengths. The amount of information revealed by abstract states in our formulation is close to the information revealed by Bonet et al.'s partially observable formulation: the agent knows only which hallway it is in, and whether or not it is at, or next to, a corner.

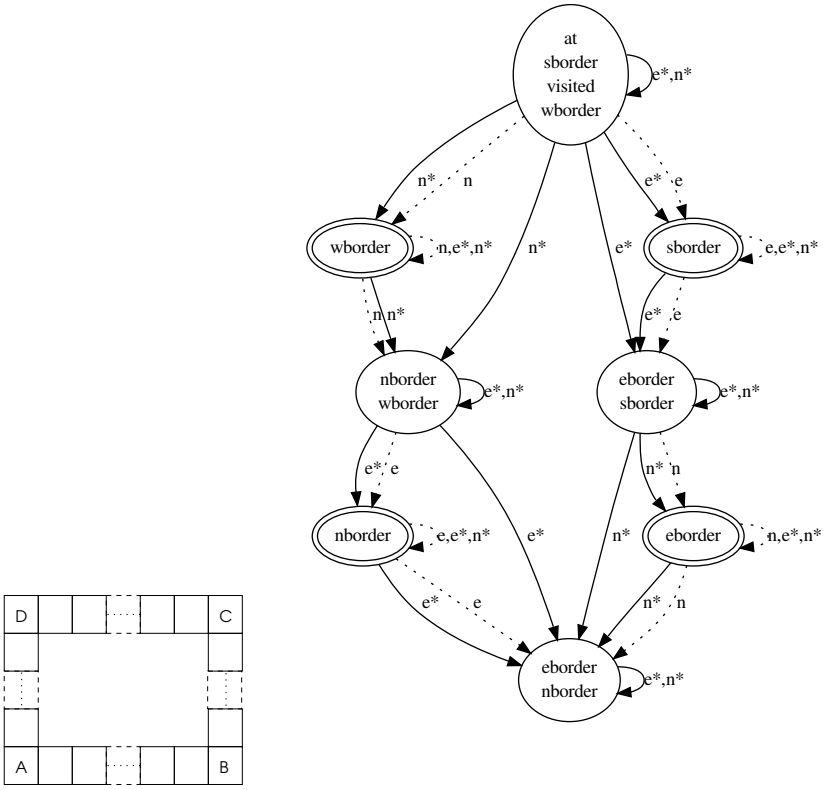

Figure A.18: The quadrilateral hallway and its abstract representation for the Hall-A problem.

Representing Grid-world Problems

In all of the following problems we model grids by representing the agent's location in the grid using distances from all the four borders. These distances are represented as role-counts of the four single-predicate roles created by the abstraction predicates $\{dFromE, dFromW, dFromN, dFromS\}$. Each of the four $mvX()$ actions for moving along the cardinal directions adds and subtracts an element from the corresponding pair of roles. As with the formulation of Hall-A above, the amount of information revealed by the abstract states closely matches that of the belief states used by Bonet et al.

The vocabulary and actions for each of the following problems therefore, are:
Vocabulary: $\{dFromE, dFromW, dFromN, dFromS\}$
Actions: $\{mvE(), mvW(), mvN(), mvS()\}$

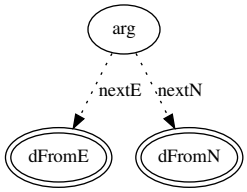

Figure A.19: Initial abstract structure for the Prize-A problem.

Prize-A

In Prize-A, the agent must completely traverse all the squares of a given rectangular grid, starting at a given corner. The abstract start structure is shown in Fig. A.19.

For this problem, Bonet et al. obtain a single-state controller for a 4×4 grid which can actually work for all grids composed of 4 columns of squares. Their implementation could not solve problems with more rows.

Utilizing example plans that traversed the grid row-wise, our approach easily scales to grids with higher numbers of rows. We present timing results with 5 and 7 row grids in Table 2. Note that a complete general solution to a grid with n rows is quadratic in n, and consequently cannot be learned from such example plans because of our restriction to generalized plans with simple loops. The obtained generalized plans have a different simple loop for each row in the grid.

The preconditions constrain the number of iterations of all but the last loop to be equal; as in the blocks problem, they are more general than the initial abstract structure in Fig. A.19 and allow the starting location to be at a distance from the West corner. Consequently, the number of iterations of every loop other than the first eastward traversal are constrained to be equal, restricting the plan to rectangular grids. Further, if $\#\{dFromW\}$ is set to zero, denoting a start at the southwest corner, the number of iterations of the first loop (l_6) also get constrained to be equal to the others.

Corner-A

In the Corner-A problem, the agent must reach the top right corner of the grid. The start structure for this problem is shown in Fig. A.20.

We used an example plan that moved the agent to the right and then up along the right boundary. The learned generalized plan consists of a loop of *mvE()* actions followed by a loop of *mvN()* actions. Preconditions show that the plan works for any grid at least 3 squares wide and 2 squares high.

Allen, J. F., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M. D.,

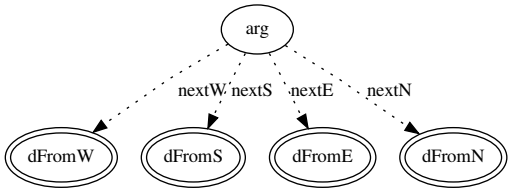

Figure A.20: Initial abstract structure for the corner problem.

Taysom, W., 2007. Plow: A collaborative task learning agent. In: Proc. of the 22nd National Conference on Artificial Intelligence. pp. 22–26.

Baier, J. A., Fritz, C., McIlraith, S. A., 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In: Proc. of the 17th International Conference on Automated Planning and Scheduling. pp. 26–33.

Bonet, B., Geffner, H., 2000. Planning with incomplete information as heuristic search in belief space. In: Proc. of the 6th International Conference on Artificial Intelligence Planning and Scheduling. pp. 52–61.

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

Cimatti, A., Pistore, M., Roveri, M., Traverso, P., 2003. Weak, strong, and strong cyclic planning via symbolic model checking. Artificial Intelligence 147 (1-2), 35–84.

Cimatti, A., Roveri, M., Traverso, P., 1998. Automatic OBDD-based generation of universal plans in non-deterministic domains. In: Proc. of the Fifteenth National Conference on Artificial Intelligence. pp. 875–881.

Cook, B., Podelski, A., Rybalchenko, A., 2006. Termination proofs for systems code. In: Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 415–426.

Dejong, G., Mooney, R., 1986. Explanation-based learning: An alternative view. Machine Learning 1 (2), 145–176.

Eker, S., Lee, T. J., Gervasio, M., 2009. Iteration learning by demonstration. In: Papers from AAAI 2009 Spring Symposium on Agents that Learn from Human Teachers.

Erol, K., Hendler, J., Nau, D. S., 1994. HTN planning: Complexity and expressivity. In: Proc. of the 12th National Conference on Artificial Intelligence. pp. 1123–1128.

Feng, Z., Hansen, E. A., 2002. Symbolic heuristic search for factored Markov decision processes. In: Proc. of The 18th National Conference on Artificial intelligence. pp. 455–480.

Fern, A., Yoon, S., Givan, R., 2006. Approximate policy iteration with a policy language bias: Solving relational markov decision processes. Journal of Artificial Intelligence Research 25, 85–118.

Fikes, R., Hart, P., Nilsson, N., 1972. Learning and executing generalized robot plans. Tech. rep., AI Center, SRI International.

Henzinger, T. A., Jhala, R., Majumdar, R., Sutre, G., 2002. Lazy abstraction. In: Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 58–70.

Hoey, J., St-Aubin, R., Hu, A., Boutilier, C., 1999. SPUDD: Stochastic planning using decision diagrams. In: Proc. of the 15th Conference on Uncertainty in Articial Intelligence. pp. 279–288.

Hoffmann, J., Brafman, R. I., 2005. Contingent planning via heuristic forward search with implicit belief states. In: Proc. of the 15th International Conference on Automated Planning and Scheduling. pp. 71–80.

Hoffmann, J., Sabharwal, A., Domshlak, C., 2006. Friends or foes? An AI planning perspective on abstraction and search. In: Proc. of the 16th International Conference on Automated Planning and Scheduling. pp. 415–469.

Hogg, C., Munoz-avila, H., Kuter, U., 2008. HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In: Proc. of the 23rd National Conference on Artificial Intelligence. pp. 950–956.

Knoblock, C. A., 1991. Search reduction in hierarchical problem solving. In: Proc. of the 9th National Conference on Artificial Intelligence. pp. 686–691.

Lau, T., Wolfman, S. A., Domingos, P., Weld, D. S., 2003. Programming by demonstration using version space algebra. Machine Learning, 111–156.

Lau, T. A., Bergman, L. D., Castelli, V., Oblinger, D., 2004. Sheepdog: Learning procedures for technical support. In: Proc. of the 9th International Conference on Intelligent User Interfaces. pp. 109–116.

Levesque, H. J., 2005. Planning with loops. In: Proc. of the 19th International Joint Conference on Artificial Intelligence. pp. 509–515.

Levesque, H. J., Pirri, F., Reiter, R., 1998. Foundations for the situation calculus. Electronic Transactions on Artificial Intelligence 2, 159–178.

Levesque, H. J., Reiter, R., Lesperance, Y., Lin, F., Scherl, R. B., 1997. GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming 31, 59–83.

Peot, M. A., Smith, D. E., 1992. Conditional nonlinear planning. In: Proceedings of the 1st International Conference on Artificial Intelligence Planning Systems. pp. 189–197.

Podelski, A., Rybalchenko, A., 2004. A complete method for the synthesis of linear ranking functions. In: Proc. of VMCAI 2004: Verification, Model Checking, and Abstract Interpretation. pp. 239–251.

Reps, T., Sagiv, M., Loginov, A., 2003. Finite differencing of logical formulas for static analysis. In: Proc. of European Symposium on Programming. pp. 380–398.

Sagiv, M., Reps, T., Wilhelm, R., 2002. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24 (3), 217–298.

Shavlik, J. W., 1990. Acquiring recursive and iterative concepts with explanation-based learning. Machine Learning 5, 39–70.

Spalazzi, L., 2001. A survey on case-based planning. Artificial Intelligence Review 16 (1), 3–36.

Srivastava, S., Immerman, N., Zilberstein, S., 2008. Learning generalized plans using abstract counting. In: Proc. of the 23rd National Conference on AI. pp. 991–997.

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

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

Winner, E., Veloso, M., 2003. DISTILL: Learning domain-specific planners by example. In: Proc. of the 20th International Conference on Machine Learning. pp. 800–807.

Winner, E., Veloso, M., 2007. LoopDISTILL: Learning domain-specific planners from example plans. In: Workshop on AI Planning and Learning, in conjunction with ICAPS-07.

Yaman, F., Oates, T., 2007. Workflow inference: What to do with one example and no semantics. In: Proc. of the AAAI Workshop on Acquiring Planning Knowledge via Demonstration.