# Foundations of Generalized Planning

Siddharth Srivastava Neil Immerman Shlomo Zilberstein

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

**Abstract**

Learning and synthesizing plans that can handle multiple problem instances are longstanding open problems in AI. We present a framework for *generalized planning* that captures the notion of algorithm-like plans, while remaining tractable compared to the more general problem of automated program synthesis. Our formalization facilitates the development of algorithms for finding generalized plans using search in an abstract state space, as well as for generalizing example plans. Using this framework, and building on the TVLA system for static analysis of programs, we develop algorithms for plan search and plan generalization. We also identify a class of domains called extended-LL domains where we can precisely characterize the set of problem instances solved by our generalized plans. Finally, we use our formalization to develop measures for evaluating and comparing generalized plans. We use these measures to evaluate the outputs of our implementation for generalizing example plans.

## 1 Introduction

Planning is among the oldest problems to be studied in AI. From a restricted formulation in the original STRIPS framework (Fikes and Nilsson, 1971), advances in knowledge representation and planning techniques have enabled planners to deal with situations involving uncertainty, non-determinism and probabilistic dynamics. While these advances strive to make planners more robust, their scope has been restricted to finding the solution of a single given planning problem. In practical terms, this is restrictive and necessitates renewed problem solving effort for every new problem, however close it may be to one previously solved. Our goal is to be able to search for, and learn from examples, plans that can solve classes of related problems. For example, consider the Unit Delivery problem (Fig. 1).

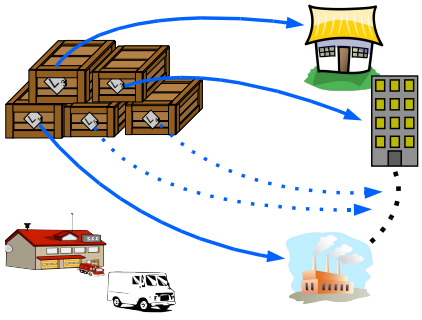

Figure 1: The unit delivery problem

*Move Truck to Dock*
*While #(undelivered crate)>0*
  *Load a crate*
  *Find crate's destination*
  *Move Truck to destination*
  *Unload crate*
  *Move Truck to Dock*
*Move Truck to Garage*

Figure 2: A handwritten algorithm for solving the unit delivery problem

There are some crates marked with their destinations at a dock, and a truck in a garage. For the purpose of this problem, we will consider each crate as representing a unit of cargo equal to the truck's capacity. The goal is to determine each crate's destination using a sensing action, and deliver it using

the available truck. A simple, hand written algorithm for solving this problem is shown in Fig. 2. A few questions central to our research are: what sets this algorithm apart from a plan? Is it possible to find plans resembling this algorithm for solving classes of different classical planning problems? Will this task be as intractable as the general problem of automated program synthesis?

Simple as the presented example is, numerous factors make solutions like the one in Fig. 2 beyond the scope of current planners. In fact, the problem itself, as stated, cannot be handled by current classical planners because it does not specify the exact numbers of crates and destinations. Yet, it is a very practical statement, and it is not unreasonable to expect it to be solvable by state-of-the-art AI Planners. Complicating this problem is the fact that every instance, with a fixed number of crates and destinations requires a conditional planner in order to utilize information that can only be obtained during plan execution (the actual destination for a crate). Note that there is no uncertainty in the outcomes of the non-sensing actions.

If the problem is difficult to represent using the current planning paradigms, the solution is even more out of reach: plans like the one shown in Fig. 2 are particularly difficult to find because of the included loop. Not only are loops difficult to recognize while searching for a plan, the presence of a loop makes it difficult to reason about the effects of a plan, or more precisely, whether it will terminate for all possible inputs, and if so, with the desired results. A plan without any indication of applicability would be of little utility when the expected usage is over a large class of problems.

Despite persistent interest in this problem, previous research efforts have resulted in little progress. An early approach to this problem (Fikes et al., 1972) worked by building databases of parametrized action-sequences and effects for use or adaptation in similar problems. These ideas have led to current approaches like *case based planning* (Spalzzi, 2001), where problem cases are stored along with parametrized plans; a significant amount of research and processing effort in these approaches is devoted to the efficient storage, retrieval, and adaptation of relevant problems and plans. However, these approaches cannot find algorithm-like plans which readily work across a class of problems. Another approach, based on *explanation based learning*, is to create proofs or explanations of plans that work for sample problem instances; these proofs can then be generalized, and more general plans can be extracted from the generalized proofs (Shavlik, 1990). This approach however, requires hand coded theories for all the looping concepts that are to be discovered in observed plans. KPLANNER (Levesque, 2005) is an iterative planner with goals similar to ours. It takes a single integer planning parameter, and iteratively produces plans for problem instances with increasing values of this parameter. It searches for patterns resembling loop iterations in these plans, and is able to produce plans with loops in several interesting problems. This approach is limited by the single planning parameter and an absence of methods to reason about the effects of the learned plans. Hybrid approaches like DISTILL (Winner and Veloso, 2003, 2007) derive a partial order on the actions in an observed plan using plan annotations; patterns in parametrized, partially ordered example plans are then used to construct plans with loops and branches. This approach is oriented towards solving all problems in a domain, but based on what has been published so far, the extent of its loop extraction capabilities is unclear. Classification of solved instances are also not provided under this approach.

An interesting aspect of research in this area is that although they work with similar objectives, results from different approaches are seldom comparable. With so many different factors influencing research goals and output quality, the lack of a unifying framework makes it difficult to assess research progress. Another common aspect across all of these approaches for computing such plans is that they rely upon post-facto plan generalization. To our knowledge, in spite of continued interest, there is currently no paradigm for directly computing plans that solve rich classes of problems, without using automated deduction. Does this problem require the use of logical inferences from an axiomatic knowledge base with its associated incomputability? A key objective of our new planning paradigm is to dispel this notion.

In this paper we develop a formalization for *generalized planning problems*, which captures the concept of "planning problem classes" (Section 2). They consist of a general goal condition and various problem instances which could vary not only in their initial states and specific goals, but also in the domain structure, like the graph topology in path planning. Solutions to generalized planning problems are algorithm-like *generalized plans*, a formalization and extension of the kind of operation structure

seen in Fig. 2. Although the notions of generalized plans and algorithms are similar, our framework shows that a large and useful class of generalized planning problems is tractable and does not suffer from undecidability of the halting problem (Prop. 1). Consequently, given a generalized plan, in these problems it is possible to determine if it will work for a given problem instance. This makes methods for lazily expanding generalized plans to solve more problem instances possible by precluding the incomputability of full-fledged program synthesis. In a restricted setting, we also show that we can easily determine the exact number of times we need to go around each loop in a generalized plan in order to reach the goal (Cor. 3).

We present a new, unified paradigm for searching for generalized plans via search in a finite, abstract state space, and for generalizing example plans (Section 3). We provide a detailed description of one such framework that we have developed using a well established technique for state-abstraction from research in software model checking (Sagiv et al., 2002; Loginov et al., 2006). The abstraction technique is described in Section 4, and our algorithms for generalized planning are discussed in Section 5. These algorithms can find the *most general* plans in a wide class of problems (Theorem 3), with ongoing extensions to other problem classes. Section 5 also includes a theoretical analysis of our methods to develop an algorithm for finding preconditions of the generalized plans that we compute. This analysis also provides a classification of the kind of domains where our algorithms are proven to work. Our techniques are illustrated with several working examples, of which the unit delivery problem is the least complicated. We discuss other approaches directed at solving similar problems in Section 6, and illustrate the practicality of our own approach with an implementation of the plan generalization module (Section 7).

The formalization of generalized planning problems provides a basis for developing measures for evaluating plans that solve multiple problems. While approaches for finding such plans have been studied and developed over decades, there is no common framework for comparing their results. As any field matures, measures of the quality of results become necessary for focused improvements and the overall coherence of research. As a part of the section formalizing generalized planning (Section 2.1), we develop a set of measures for evaluating the ability of a generalized plan (or plan database) for solving classes of problems. While these measures and their objectives are significantly different from those used in classical planning, they are general enough to be applicable to the results of any approach for generalizing plans, including those discussed above.

## 2 Generalized Planning

We take the state-based approach to planning, so that action dynamics are described by state transition functions. The planning problem is to find a sequence of actions taking an initial state to a goal state. Any planning problem is thus defined by an initial state, the set of action operators, and a set of goal states. To capture classes of planning problems, we first formalize the concept of a state as a logical structure. We express all relevant properties in the language of first-order logic with transitive closure (FO(TC)). This formalization most practical planning situations including all the classical planning benchmarks. The relations used to define such structures, together with the action operators and a set of integrity constraints defining legal structures constitute a domain-schema. Intuitively, a domain-schema lays out the world of interest in a generalized planning problem. Formally,

**Definition 1** *(Domain schema)* A domain-schema is a tuple $\langle \mathcal{V}, \mathcal{A}, \mathcal{K} \rangle$ where the vocabulary $\mathcal{V} = \{R_1^{n_1}, \ldots, R_k^{n_k}\}$ is a set of predicate symbols together with their arities, $\mathcal{A}$ is a set of action operators, and $\mathcal{K}$ is a collection of *integrity constraints* about the domain. Each action operator $a(x_1, \ldots, x_n)$ consists of a precondition $\varphi_{pre}$ and a state transition function $\tau_a$ which maps a legal (wrt integrity constraints) state with an instantiation for the operands satisfying the preconditions to another legal state:

$$\tau_a(\bar{x}) : \{(S, i) | S \in ST[\mathcal{V}]^{\mathcal{K}} \wedge (S, i) \models \varphi_{pre}\} \rightarrow ST[\mathcal{V}]^{\mathcal{K}}$$

where $\bar{x} = (x_1, \ldots, x_n)$, $ST[\mathcal{V}]^{\mathcal{K}}$ is the set of finite structures over $\mathcal{V}$ satisfying $\mathcal{K}$, and $(S, i)$ denotes the structure $S$ together with an interpretation for the variables $x_1, \ldots, x_n$.

An instance from a domain-schema is a structure in its vocabulary satisfying the integrity constraints. With this notation, we define a generalized planning problem as follows

**Definition 2** *(Generalized planning problem)* A generalized planning problem is a tuple $\langle \mathcal{I}, \mathcal{D}, \varphi_g \rangle$ where $\mathcal{I}$ is a possibly infinite set of instances, $\mathcal{D}$ is the domain-schema, and $\varphi_g$ is the common goal formula. $\mathcal{I}$ is typically expressed via a formula in FO(TC).

**Example 1** We illustrate the expressiveness of generalized planning problems using the graph 2-coloring problem. The vocabulary $\mathcal{V}_G = \{E^2, R^1, B^1\}$, consists of the edge relation and the two colors. The set of actions is $\mathcal{A}_G = \{colorR(x), colorB(x)\}$, for assigning the red and blue colors. Their preconditions are respectively $\neg B(x), \neg R(x)$. Integrity constraints stating our graphs are undirected, the uniqueness of color labels, and the coloring condition are given by

$$\begin{aligned} \mathcal{K}_G \quad = \quad & \{\forall x(\neg(R(x) \wedge B(x)) \wedge \\ \forall y \quad & (E(x,y) \rightarrow (E(y,x) \wedge \neg(R(x) \wedge R(y)) \\ & \wedge \neg(B(x) \wedge B(y))))\} \end{aligned}$$

We consider the planning problem with $\mathcal{I}_G = ST[\mathcal{V}_G]^{\mathcal{K}_G}$ (represented by the formula **true**), the domain schema $\langle \mathcal{V}_G, \mathcal{A}_G, \mathcal{K}_G \rangle$, and the goal condition $\varphi_{2-color} = \forall x(R(x) \vee B(x))$.

In addition to the relations from a problem's domain schema, solving a generalized planning problem may require the use of some auxiliary relations for keeping track of useful information. For example, in a transport domain such relations can be used to store and maintain shortest paths between locations of interest. These paths can then be used to move a transport using the regular state-transforming actions. In order to maintain or extract such information, a generalized plan can include *meta-actions* which update the auxiliary relations. The action of choosing an instantiation of operands is also a simple kind of meta-action. A choice meta-action is specified using the variables to be chosen, and a formula describing the constraints they should meet, e.g. *choose* $x, y : (E(x,y) \wedge \neg R(y))$. Solutions to generalized planning problems are called *generalized plans*. Intuitively, a generalized plan is a full fledged algorithm. Formally,

**Definition 3** *(Generalized plan)* A generalized plan $\Pi = \langle V, E, \ell, s, T, \mathcal{M} \rangle$ is defined as a tuple where $V, E$ are the vertices and edges of a connected, directed graph; $\ell$ is a function mapping nodes to actions and edges to conditions; $s$ is the start node and $T$ a set of terminal nodes. The tuple $\mathcal{M} = \langle R_m, A_m \rangle$ consists of the vocabulary of auxiliary relations ($R_m$) and the set of meta-actions ($A_m$) used in the plan.

During an execution, the generalized plan's *run-time configuration* is given by a tuple $\langle pc, S, i, \mathcal{R} \rangle$ where $pc \in V$ is the current node to be executed; $S$, the current state on which to execute it; $i$, an instantiation of the free variables in $\ell(pc)$, and $\mathcal{R}$, an instantiation over $S$ of the auxiliary relations in $R_m$. In general, compound node labels consisting of multiple actions and meta-actions can be used for ease of expression. For simplicity, we allow only a single action per node and require that all of an action node's operands be instantiated before executing that node. Unhandled edges such as those due to an action's precondition not being satisfied and due to non-exhaustive edge labels are assumed to lead to a default terminal (trap) node.

A generalized plan is executed by following edges whose conditions are satisfied, starting with $s$. After executing the action at a node $u$, the next possible actions are those at neighbors $v$ of $u$ for which the label of $\langle u, v \rangle$ is satisfied by $(S, i)$. Non-deterministic plans can be achieved by labeling a node's outgoing edges with mutually consistent conditions. A generalized plan **solves** an instance $i \in \mathcal{I}$ if every possible execution of the plan on $i$ ends with a structure satisfying the goal.

**Example 2** A generalized plan for the 2-coloring problem discussed above can be written using an auxiliary labeling relation $D(x)$ for nodes whose processing is done. $D$ is initialized to $\phi$ and is modified using the meta-action $setD(x)$. The generalized plan is shown in Fig. 3. We use the abbreviation $C(x)$ for $B(x) \vee R(x)$.

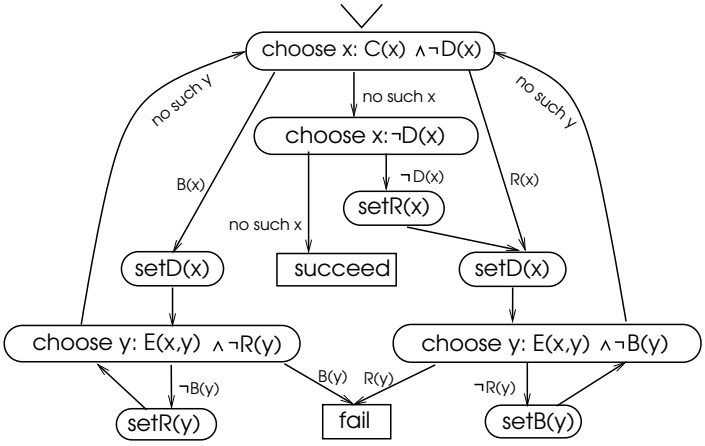


Figure 3: Generalized plan for 2-coloring a graph

We call a generalized planning problem "finitary" if for every instance $i \in \mathcal{I}$, 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 are practical because they capture most real-world situations and are decidable: in finitary domains, the language consisting of instances that a generalized plan solves is decidable. This is because in these domains we can maintain a list of visited states (which has to be finite), and determine when the algorithm goes into a non-terminating loop. Finitary domains thus have a solvable halting problem. We formalize this notion with the following proposition:

**Proposition 1** (Decidability in finitary domains) *The halting problem for any generalized plan in a finitary domain is decidable.*

Therefore, while generalized plans bring the full power of algorithms, finitary domains capture a widely applicable class of planning problems that are potentially tractable. This makes it possible for us to determine preconditions for generalized plans and determine when the search for a more general plan can be useful.

We define the *domain coverage* ($DC$) of a generalized plan as the set of instances that it solves. Generalized plans for a particular problem can be partially ordered on the basis of their domain coverage: $\Pi_1$ is more general than $\Pi_2$ ($\Pi_2 \preceq \Pi_1$) iff $DC(\Pi_2) \subseteq DC(\Pi_1)$. A *most general plan* is one which is more general than every other plan. Thus, if plan preconditions are available and can be determined or sensed, then any plan that is not most general can be extended by adding a branch for a solvable instance not covered before. In finding most general plans therefore, the problem is that of finiteness – when can we compress the complete set of plans into a finite generalized plan?

Common restrictions on plan representations such as disallowing loops can make the problem such that a most general plan can never be found – for example, for unstacking a tower of blocks.

The question of plan preconditions is particularly important for generalized plans since it determines their usability. We call a generalized plan *speculative* if it does not come with a classification of the problem instances that it solves. Given a problem instance, such plans would have to either be tested through simulations prior to application, or allow for re-planning during execution.

## 2.1 Evaluation of Generalized Plans

Our theoretical framework allows us to address a serious hurdle in the development of generalized planners. Contrary to classical planning which has seen a development of rich plan metrics, the only demonstrated comparisons in approaches finding generalized plans have been over the time taken to

generate them. However, solutions generated by generalized planners are usually partial and can differ over a variety of factors. Time comparisons provide no information about the quality of results. This hampers research progress due to lack of clear objectives for successive approaches to improve upon: current generalized planners work with the very generic objective of finding *a* generalized plan. Our framework for generalized planning provides a setting for comparing generalized plans.

The quality of a generalized plan depends on various factors, some of which are not applicable to classical plans. In this section we present some measures for evaluating generalized plans. With the exception of plan optimality, these measures are new. It is important to note that no single measure of a generalized plan suffices as an accurate assessment of its quality. Different generalized plans can represent trade-offs between factors such as the cost of testing if a plan will work for a given instance, the range of instances that it actually covers, and its optimality. Considered together, the measures presented below can help in comparing the most important aspects of different available generalized plans. When analytical computation is not possible, estimates of these measures can be computed by sampling problem instances from an appropriate distribution over the input problem class, and taking an average of the measure. Also, although we define most measures as functions over the class of initial instances, constant upper and lower bounds on these functions may be easily obtainable. Such bounds can also be effective in evaluating a generalized plan. We illustrate the general nature of these measures by applying them on the algorithm like plan for the unit delivery problem (Fig. 2) and for the solution of the graph 2-coloring problem (Fig. 3).

#### 2.1.1 Domain coverage

This measure is a generalization of plan correctness. In classical planning, correctness is ensured if a plan is found using the right domain semantics. In generalized planning, the issue of correctness translates to determining *when* the generalized plan will work, or its domain coverage. The domain coverage of a generalized plan can be expressed either explicitly as the set of instances covered (derived analytically), or as a fraction of the total set of input instances. An estimate of this fraction can also be obtained by sampling, and can be used to approximate the likelihood that an instance of the generalized planning problem will be solved by the given generalized plan. Since the algorithm in Fig. 2 solves all instances, its domain coverage is 100%. The same holds for the generalized plan in Fig. 3.

#### 2.1.2 Computational cost of testing applicability

Given a problem instance, the applicability of a generalized plan needs to be tested prior to application. Such a test may vary from a condition to be evaluated on the input structure, to a simulation of the plan on a model of the input problem instance. If simulation is used as a test, then the cost of testing applicability is the total amount of computation required, including that required for instantiation. Since the unit delivery algorithm and the 2-coloring plan work for all possible problem instances, they incur no cost of testing applicability.

#### 2.1.3 Measures of plan performance

As in classical planning, the performance of a generalized plan is an important factor in assessing its quality. For classical plans, this can be measured simply by counting the total number of actions executed, or the sum of their appropriately assigned costs. The execution of a generalized plan includes more features which can be measured and improved in subsequent planning efforts. A simplistic method for assessing the performance of a generalized plan would be to compare it with the best classical plan for an instance. However, the total execution cost of a generalized plan on an instance includes the costs for instantiating the choice actions and executing any meta-actions required for executing subsequent actions. A classical plan on the other hand, is designed for a particular instance and knows a-priori all the operand choices. Considering actions useable by concrete plans for the given domain as "basic actions", we separate the performance of a generalized plan into two parts: the cost of instantiation (which is not incurred by concrete plans), and the cost associated with the basic actions in the generalized plan.

With this terminology, we can easily compare the performance of concrete plans and its appropriate counterpart for generalized plans.

**Computational cost of instantiating the generalized plan** As discussed above, generalized plans may use choice actions and other meta-actions in order to be able to deal with multiple problem instances. In particular, a generalized plan must use variables in place of action operands and in branch or loop conditions. These variables are instantiated using the instantiation meta-actions, which may incur a significant cost in searching suitable elements for instantiation. The cost of instantiation, $\chi_\Pi(S)$ for a generalized plan $\Pi$ measures the total number of operations required for meta-actions when executing $\Pi$ on $S$. For example, the cost of instantiation of the action "choose $x, y, z$" is $O(1)$ for any problem instance. On the other hand, the cost of "choose $x, y, z : \varphi(x, y, z)$" is bounded above by $O(n^3 \times \alpha(n))$ where $n$ is the number of elements in the problem instance, and $\alpha(n)$ is the number of operations required for testing if $\varphi$ holds in a problem instance of size $n$. In implementation, auxiliary relations would be used in order to make such choices efficient. Automatically deriving such relations and their associated algorithms is a related problem to be addressed in future research.

In the unit delivery algorithm, the only variable instantiation is that of the next chosen crate, which has cost $O(1)$ per instantiation and thus a total of at most $O(n)$. In the generalized plan for 2-coloring, we have the meta-action setD(x) in addition to the choice actions. Depending on the implementation, the cost of the choice actions can range between $O(n(n + m))$ and $O(n + m)$ where $n = |V|$ and $m = |E|$. The former is achieved when no auxiliary relation is used: every choice must cycle through all the vertices. For the latter, we use a data structure that keeps track of choices made, so that each edge and vertex is examined at most twice. The meta-action $setD(x)$ is executed at most once for each node, so its total cost is $O(V)$, which is subsumed by the upper bounds for the choice actions.

**Plan optimality** This aspect of evaluation measures the performance of a generalized plan in relation to the class of optimal plans for $\mathcal{I}$. We define the optimality of a generalized plan as a function $\eta_\Pi(S_i) = \frac{Cost(OPT(S_i))}{Cost(\Pi(S_i))}$, where $S_i \in \mathcal{I}$. $\Pi(S_i)$ represents the instantiation of $\Pi$ for $S_i$. Depending upon the application, the $Cost$ of a plan is the number of basic actions it contains, their makespan, or a cumulative cost derived by adding the cost of each basic action. In some cases, it may only be possible to provide bounds on $\eta_\Pi$, because of lack of availability of optimal plans. The optimality of the unit delivery algorithm is 1; for the 2-coloring problem, it is 1, because each node is colored exactly once (when the graph is 2-colorable).

In addition to the above criteria, another measure becomes important when dealing with learned generalized plans.

### 2.1.4 Loss of optimality due to generalization

A generalized plan may be learned using a set of example plans as input. The generalized plan could significantly differ in performance compared to the input. The loss of optimality due to generalization is measured as $\eta_{loss}(S_i) = 1 - \frac{\eta_\Pi(S_i)}{avg(\eta_{input})}$. This measure provides an explicit comparison of the generalized plan's performance with the given example plans. It also gives an indication of the quality of the learning process used.

Various trade-offs among the measures presented above come into play when choosing or creating a generalized plan. For instance, precise characterizations of domain coverage may increase the cost of applicability tests. In such situations it may actually be desirable to leave out some less frequent instances in the stated domain coverage of a plan, in favor of a simpler test for applicability.

# 3 Generalized Planning Using State Abstraction

Since solving individual planning problem instances in itself can be hard, a naive approach to generalized planning would be intractable. However, it is possible to convert the generalized planning problem into a search problem in a finite state space. Our paradigm for solving generalized planning problems is as follows:

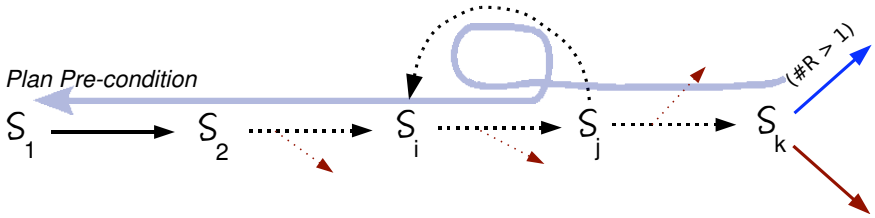


Figure 4: Finding preconditions of plan branches

**Step 1: Abstraction** Use an abstraction mechanism to collapse similar states reachable from any of the problem instances in $\mathcal{I}$ together. The total number of abstract states should be bounded. This will permit tractable handling of infinitely many states by working in the abstract state space. The abstraction should provide sufficient information to discern when an abstract state satisfies the goal.

The concrete action mechanism may also have to be enhanced to work with abstract states. In particular, the abstract action mechanism should be able to produce appropriate branching effects when the result of an action on members of an abstract state consists of concrete states belonging to different abstract states.

**Step 2: Computing preconditions** Because of the branching effects described above, a plan given by a sequence of actions will work for those initial concrete states that always take the branch that goes towards the goal. The computability of such states depends on the abstraction and action mechanisms used: we need to be able to compute the conditions under which a concrete state will take the desirable branch of an action and result in a desirable part (defined by a similar condition, for the later actions) of the resulting abstract state. By doing this all the way to the start action we can find the preconditions under which the given sequence of actions will lead to the goal (Fig. 4).

**Step 3: Plan synthesis and generalization**

**Plan Synthesis** The abstraction mechanism and the procedure for computing preconditions yield an algorithm for searching for generalized plans: the abstraction mechanism is used to come up with a representation $\mathcal{S}_0$ of the input class of instances. This representation should be finite. Starting with $\mathcal{S}_0$ we simulate a simultaneous search for a goal structure over infinitely many state spaces by using the abstract action mechanism on $\mathcal{S}_0$. Unlike search in a concrete domain, paths with loops in the abstract state space can be very useful for the plan's domain coverage. Although we are extending our work to nested loops, we focus on paths with non-nested ("simple") loops in this paper. Once a path with simple loops to a goal structure is found, its preconditions are computed (this also determines if the included loops make progress). If the preconditions increase the coverage beyond any existing plan branches, the path is included in the repository of condition-indexed plan branches.

**Plan Generalization** An abstraction mechanism can also be used to generalize example plans by a process we call *tracing*. In essence, the abstract action mechanism can be used to translate the actions from an example plan to those in the abstract state space. The resulting sequence of actions can be applied on an abstract input structure. Since abstraction collapses similar states from the point of view of action preconditions, any loop unrollings in the example plan become obvious as repeated state and action subsequences. A new plan can then be extracted by collapsing these subsequences into loops. This gives a generalization of the original plan; in addition, if precondition evaluation is possible, a classification of the solved instances can be generated.

In the next two sections we discuss an application of this paradigm built using TVLA (Sagiv et al., 2002), a tool for static analysis of programs. We begin by providing a summary of the abstraction

techniques developed in TVLA with emphasis on how we use them, in Section 4.

# 4 State Abstraction Using 3-Valued Logic

## 4.1 Representation

As discussed in Section 2, we represent states of a domain by traditional (two-valued) logical structures. State transitions are carried out using action operators specified in FO[TC], defining new values of every predicate in terms of the old ones. We use the terms "structure" and "state" interchangeably.

We represent abstract states using 3-valued structures also called "abstract structures". In a 3-valued structure, each tuple may be present in a relation with logical value 1 (present), 0 (not present), or $\frac{1}{2}$ (perhaps present). See Appendix A for more detail about 3-valued structures.

**Example 3** The delivery domain can be modeled using the following vocabulary: $\mathcal{V} = \{crate^1, dock^1, garage^1, location^1, truc$ An example structure, $S$, for the unit delivery problem discussed above can be described as: the universe, $|S| = \{c, d, g, l, t\}$, $crate^S = \{c\}$, $dock^S = \{d\}$, $garage^S = \{g\}$, $location^S = \{l\}$, $truck^S = \{t\}$, $delivered^S = \emptyset$, $in^S = \emptyset$, $at^S = \{(c, d), (t, g)\}$, $destination^S = \{(c, l)\}$.

## 4.2 Action Mechanism

The action operator for an action (e.g., $a(\bar{x})$) consists of a set of preconditions and a set of formulas defining the new value $p'$ of each predicate $p$.

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

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

The RHS of this equation consists of two conjunctions, the first of which holds for arguments on which $p_i$ is changed to true by the action; the second holds for arguments on which $p_i$ was already true, and remains so after the action. These update formulas resemble successor state axioms in situation calculus. However, we use query evaluation on possibly abstract structures rather than theorem proving to derive the effect of an action.

We use $a(\bar{x})$ to denote a complete action operator including the precondition test and the action update (more components will be added to deal with abstract structures); $\tau_a(\bar{x})$ denotes just the predicate update part of $a(\bar{x})$. This separation will be helpful when we deal with abstract structures.

**Example 4** The delivery domain has the following actions: $Move(x_1, x_2)$, $Load(x_1, x_2)$, $Unload(t)$, $findDest(x, l)$. With obj as the vehicle and loc as the location to move to, update formulas for the $Move(\text{obj}, \text{loc})$ action are:

$$\begin{aligned} at'(u, v) \quad := \quad & \{at(u, v) \wedge (u \neq \text{obj} \wedge \neg in(u, \text{obj}))\} \vee \\ & \{\neg at(u, v) \wedge (v = \text{loc} \wedge ((u = \text{obj}) \vee in(u, \text{obj})))\} \end{aligned}$$

The goal condition is represented as a formula in FO[TC]. For example, $\forall x(crate(x) \rightarrow delivered(x))$. The predicate *delivered* is updated to True for a crate when the *Unload* action unloads it at its destination.


Figure 5: Abstraction in the delivery domain

### 4.3 Abstraction Using 3-valued Logic

Given a domain-schema, some special unary predicates are classified as *abstraction predicates*. The special status of these predicates arises from the fact that they are preserved in the abstraction. We define the *role* an element plays as the set of abstraction predicates it satisfies:

**Definition 4** *(Role)* A role is a conjunction of literals consisting of every abstraction predicate or its negation.

**Example 5** In all our examples, we let the set of abstraction predicates be all the unary predicates used for the problem representation. In the delivery domain, the role representing crates that are not delivered is $crate(x) \wedge \neg delivered(x) \wedge \neg dock(x) \wedge \neg garage(x) \wedge \neg location(x) \wedge \neg truck(x)$. For simplicity, we will omit the negative literals and the free variable when expressing a role. The role above then becomes *crate*.

We perform state abstraction using *canonical abstraction* (Sagiv et al., 2002). This technique abstracts a structure by merging all objects of a role into a *summary object* of that role. The resulting abstract structure is used to represent concrete structures having a positive number of objects for every summary object's role. Since the number of roles for a domain is finite, the total number of abstract structures for a domain-schema is finite; we can tune the choice of abstraction predicates so that the resulting abstract structures effectively model some interesting generalized planning problems and yet the size and number of abstract structures remains manageable.

The imprecision that must result when objects are merged together is modeled using three-value logic. In a three-valued structure the possible truth values are $0, \frac{1}{2}, 1$, where $\frac{1}{2}$ means "don't know". If we order these values as $0 < \frac{1}{2} < 1$, then conjunction evaluates to minimum, and disjunction evaluates to maximum. The universe of the canonical abstraction, $S'$, of structure $S$, is the set of nonempty roles of $S$. The truth values in canonical abstractions are as precise as possible: if all embedded elements have the same truth value then this truth value is preserved, otherwise we must use $\frac{1}{2}$.

**Example 6** Suppose the destination relation in the delivery domain is known. Fig. 6 shows the result of abstraction on this relation when only *Crate* and *Location* are used as abstraction predicates in the delivery domain. The dotted edge represents the $\frac{1}{2}$ truth value, and the encapsulated nodes denote summary objects. This abstract representation captures all possible configurations of the destination function, with any number of crates and locations.

Uncertainty in the concrete state itself can also be modeled, by assigning a tuple the truth value $\frac{1}{2}$ for a relation.

The set of concrete structures that are represented by an abstract structure $S$ is called the *concretization* of $S$, and written as $\gamma(S)$. A formal definition of canonical abstractions and embeddings can be found in Appendix A.

With such an abstraction, the update formulas for actions might evaluate to $\frac{1}{2}$. We therefore need an effective method for applying action operators while not losing too much precision. This is handled in TVLA using the *focus* and *coerce* operations.

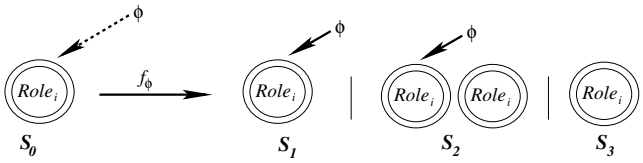


Figure 6: Effect of focus with respect to $\phi$.

#### 4.3.1 Focus and Coerce

The focus operation on a three-valued structure $S$ with respect to a formula $\varphi$ produces a set of structures which have definite truth values for $\varphi$ in every possible instantiation of the variables in $\varphi$, while collectively representing the same set of concrete structures, $\gamma(S)$. This process could produce structures that are inherently infeasible because they violate the integrity constraints. Such structures are later removed by TVLA's coerce operation. Using integrity constraints of a domain, coerce either refines them by making their relations' truth values precise, or discards them when a refinement reconciling a focused structure with integrity constraints is not possible. We use focus and coerce for the purpose of making updates to abstraction predicates precise, and also for drawing individual elements to be used as action arguments out of their roles.

A focus operation with a formula with one free variable on a structure which has only one role ($Role_i$) is illustrated in Fig. 6: if $\phi()$ evaluates to $\frac{1}{2}$ on a summary element, $e$, then either all of $e$ satisfies $\phi$, or part of it does and part of it doesn't, or none of it does. Fig. 6 also illustrates how we can use the focus and coerce operations to model the "drawing-out" of individuals from their summary elements prior to an action update, and for action argument selection. If integrity constraints restricted $\phi$ to be unique and satisfiable, then structure $S_3$ in Fig. 6 would be discarded and the summary elements for which $\phi()$ holds in $S_1$ and $S_2$ would be replaced by singletons. These two structures denote situations where (a) $\phi()$ holds for a single object of role $Role_i$, and that this is the only object of this role, and (b) $\phi()$ holds for a single object of role $Role_i$, but there are other objects of $Role_i$ as well.

Since the coerce operation refines a structure to be consistent with the integrity constraints, it effectively establishes the properties of objects drawn out by focus. The focus operation wrt a set of formulas works by successive focusing wrt each formula in turn. In an effort to make this paper self-contained, more details about the focus and coerce operations are provided in Appendix B. Further descriptions of these operations can be found at (Sagiv et al., 2002).

#### 4.3.2 Choosing Action Arguments

Action arguments are chosen using the focus and coerce operations described above. Note that if $\phi$ were constrained to be unique and satisfiable in Fig. 6, the focus operation would effectively draw-out an object from $Role_i$. We illustrate this process with the following example.

**Example 7** Consider the sequence of operations in Fig. 7 in a simplified version of the delivery domain (we ignore the object locations). $Chosen(x)$ is initialized to $\frac{1}{2}$ for all objects with the role *crate* in this figure. These operations illustrate how we use the focus and coerce operations to (a) choose an operand using the drawing-out mechanism described above, and (b) to model sensing actions by producing all possible structures with definite truth values of the formula being sensed. The effects of these operations are also illustrated in a concrete structure with known crate destinations. In this example, the second focus operation takes place over the formula $\exists x(Chosen(x) \wedge dest(x, y))$. Integrity constraints are used to assert that (a) $Chosen(x)$ must hold for a unique element, and (b) every crate has a unique destination, so that coerce discards structures where the chosen crate none, or non-unique destinations. Note that for every action, the different possible outcomes can be easily differentiated on the basis of the number of elements of some role. For instance, after the "select crate" action on the abstract starting state, the two possible outcomes are characterized by whether or not there are at least two objects with the role *crate*. This becomes useful when we need to find preconditions for action branches (Section 5.1).

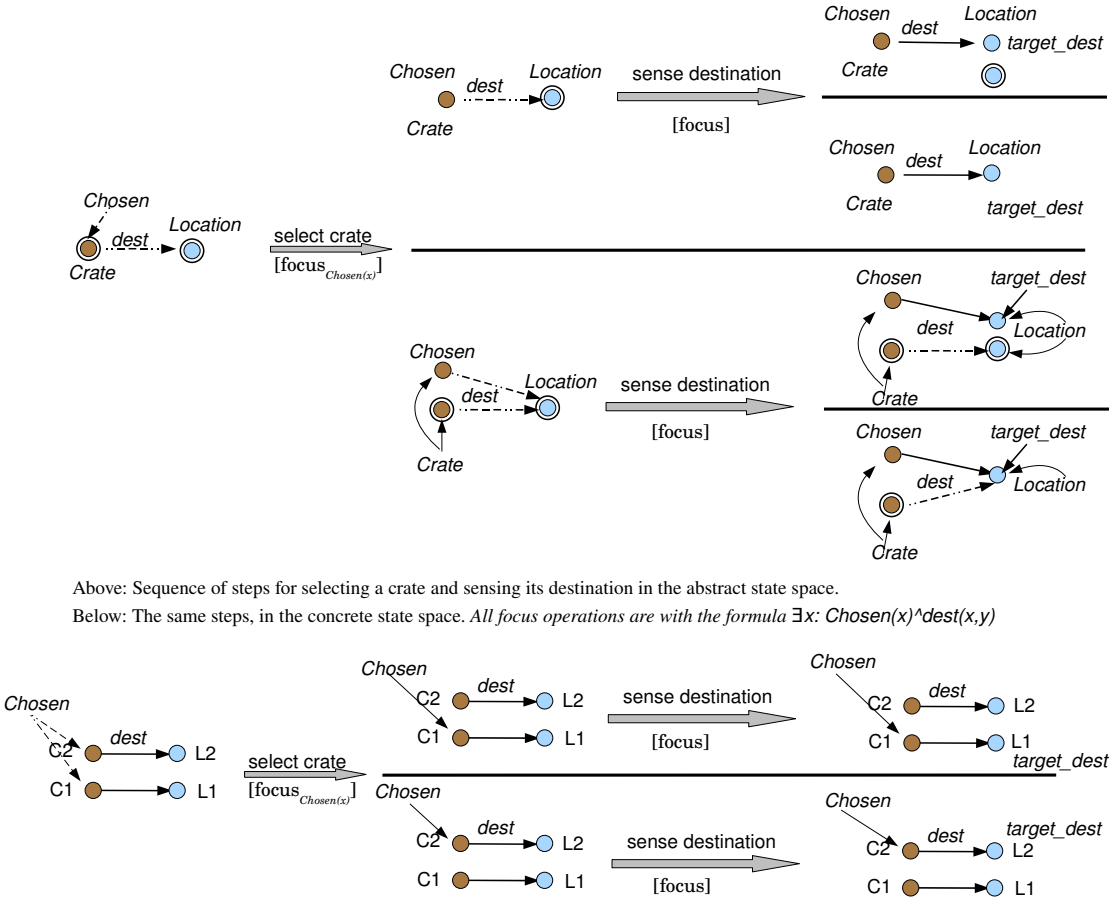


Figure 7: An action sequence in the delivery domain, shown in abstract and concrete representations. Different action outcomes are separated by horizontal lines.

### 4.3.3 Action Application

Imprecise truth values of non-abstraction predicates can lead to imprecise values of abstraction predicates after an action update (Eqn.1). In order to address this, TVLA's mechanism for action application on an abstract structure $S_1$ first focuses it using a set of user-defined, action specific focus formulas (Fig. 8). The resulting focused structures are then tested against the preconditions, and action updates are applied to those that qualify. The resulting structures are then canonically abstracted (this is called "blur" in TVLA), yielding the abstract result structures (Fig. 8). For our purposes, the most important updates are for (unary) abstraction predicates. Recall that the predicate update formulas for an action operator take the form shown in equation 1. For unary predicate updates, expressions for $\Delta_i^+$ and $\Delta_i^-$ are *monadic* (i.e. have only one free variable apart from action arguments). This completely determines the choice of focus formulas in our application of TVLA: if an abstraction predicate's update formula involves non-abstraction predicates due to which it may evaluate to $\frac{1}{2}$, we include this predicate's $\Delta_i^+$ or $\Delta_i^-$, as required, in the set of focus formulas for this action. We use $\psi_a$ to denote this set of focus formulas for an action $a$. We illustrate this choice of focus formulas using an example from the blocks world, since non-choice actions in the unit delivery problem do not need focus formulas.

**Example 8** Consider a blocks world domain-schema with the vocabulary $\mathcal{V} = \{on^2, topmost^1, onTable^1\}$,

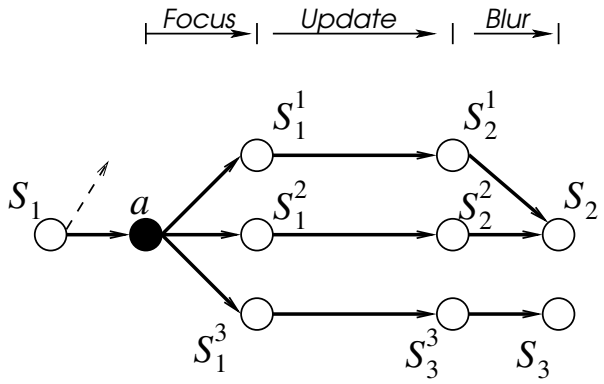


Figure 8: Action update mechanism

and abstraction predicates $\{topmost, onTable\}$. Consider the *move* action which has has two arguments: $\text{obj}_1$, the block to be moved, and $\text{obj}_2$, the block it will be placed on. Update formulas for *on* and *topmost* are:

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

Following the discussion above, the update formula for *topmost* can evaluate to $\frac{1}{2}$ because $on(obj1, x)$ can evaluate to $\frac{1}{2}$. Consequently, $on(\text{obj}_1, x) \wedge (x \neq \text{obj}_2)$ is included in the focus formula for $move()$. Effectively, 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 $\text{obj}_2$ will be a singleton element for which an abstraction predicate denoting the chosen argument will hold. For more details about the focus operation in this example, see Appendix B.

# 5 Algorithms for Generalized Planning

In this section we use the abstraction and action mechanisms presented above to develop algorithms for generalized planning. We call this system for generalized planning ARANDA[1].

## 5.1 Plan Preconditions

As discussed in Section 3, we need to be able to find the concrete states that a given sequence of actions, possibly with loops, can take to the goal. In order to accomplish this, we need a way of representing subsets of an abstract state's constituent concrete states – in particular, those that are guaranteed to take a particular branch of an action's focus operation. We also need to be able to pull these subsets backwards through action edges in the given path all the way up to the initial abstract state – thus identifying its "solved" concrete members. In order to represent subsets of an abstract structure's concretization, we will use conditions from a chosen constraint language. Formally,

**Definition 5** *(Annotated Structure)* Let $\mathcal{C}$ be a language for expressing constraints on three-valued structures. A $\mathcal{C}-$*annotated structure* $S|_C$ is a restriction of $S$ consisting of structures in $\gamma(S)$ that satisfy the condition $C \in \mathcal{C}$. Formally, $\gamma(S|_C) = \{s \in \gamma(S) \mid s \models C\}$.

[1]After an Australian tribe whose number system captures a similar abstraction.

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

We define the following notation to discuss action transitions.

**Definition 6** *(Action Transition)* Let $a$ be an action and $S_1$ a three-valued structure with instantiations (as singleton elements) for 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 $S_1^1 \in f_{\psi_a}(S_1)$ s.t. $S_2 = blur(\tau_a(S_1^1))$. The transition $S_1 \xrightarrow{a} S_2$ can be decomposed into a set of transition sequences for each result of the focus operation: $\{(S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2) | S_1^i \in f_{\psi_a}(S_1) \wedge S_2^i = \tau_a(S_1^i) \wedge S_2 = blur(S_2^i)\}$.

In order to find preconditions, we need to be able to identify the (annotated) pre-image of an annotated structure under any action. This requirement on a domain is captured by the following definition:

**Definition 7** *(Annotated Domain)* An *annotated domain-schema* is a pair $\langle \mathscr{D}, \mathcal{C} \rangle$ where $\mathscr{D}$ is a domain-schema and $\mathcal{C}$ is a constraint language. An annotated domain-schema is *amenable to back-propagation* if for every transition $S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2$ and $C_2 \in \mathcal{C}$ it is possible to compute a $C_1^i \in \mathcal{C}$ such that $\tau_a(\gamma(S_1|_{C_1^i})) = \tau_a(\gamma(S_1^i))|_{C_2}$.

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

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

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

**Proposition 2** (Linear backup) *Suppose $\langle \mathscr{D}, \mathcal{C} \rangle$ is an annotated domain-schema that is amenable to back-propagation and $S_1, \ldots, S_k \in \mathscr{D}$ are distinct structures such that $S_1 \xrightarrow{\tau_1} S_2 \cdots \xrightarrow{\tau_{k-1}} S_k$. Then for all $C_k$ there exists $C_1$ such that $\tau_{k-1\ldots1}(\gamma(S_1|_{C_1})) = (\tau_{k-1\ldots1}(\gamma(S_1)) \cap S_k)|_{C_k}$*

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

Amenability for back-propagation can be seen as being composed of two different requirements. In terms of definition 7, we first need to translate the constraint $C_2$ into a constraint on $S_1^i$. Next, we need a constraint selecting the structures in $S_1$ that are actually in $S_1^i$. Composition of these two constraints will give us the desired annotation, $C_1^i$. The second part of this annotation, classification, turns out to be a strong restriction on the focus operation. We formalize it as follows:

**Definition 8** *(Focus Classifiability)* A focus operation $f_\psi$ on a structure $S$ satisfies *focus classifiability with respect to a constraint language $\mathcal{C}$* if for every $S_i \in f_\psi(S)$ there exists an annotation $C_i \in \mathcal{C}$ such that $s \in \gamma(S_i)$ iff $s \in \gamma(S|_{C_i})$.

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

### 5.1.1 Inequality-Annotated domain-schemas

Let us denote by $\#_R(S)$ the number of elements of role $R$ in structure $S$. In this section, we use $\mathcal{C}_I(\mathcal{R})$, the language of inequalities between constants and role-counts $\#_{R_i}(S)$. When we handle paths with loops, we will extend this language to include terms for loop iterations in the inequalities.

In order to make domain-schemas annotated with constraints in $\mathcal{C}_I(\mathcal{R})$ amenable to back propagation, we first restrict the class of permissible actions to those that change role counts by a fixed amount:

**Definition 9** *(Linear Change)* An action shows *linear change w.r.t a set of roles* $\mathcal{R}$ iff whenever $S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2$ then $\forall s_1 \in \gamma(S_1^i), s_2 \in \gamma(S_2^i)$ such that $s_1 \xrightarrow{a} s_2$ and $R_j \in \mathcal{R}$, we have $\#_{R_j}(s_2) = g_j(\#_{R_1}(s_1), \ldots \#_{R_l}(s_1))$ where $g_j$ is a linear function. In the special case where changes in the counts of all roles are constants, we say the action shows *constant change.*

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

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

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

More precisely, suppose $\#_{R_j}(S_2^i) = g_j(\#_{R_1}(S_1), \ldots \#_{R_l}(S_1))$. Then we obtain the corresponding inequalities for $S_1$ by substituting $g_j(\#_{R_1}(S_1), \ldots \#_{R_l}(S_1))$ for $\#_{R_j}(S_2^i)$ in all inequalities of $C_2$ and using the linearity of $g_j$ to rearrange the terms. Let us call the resulting set of inequalities $C_1$. Because of linear change, if $C_1$ is satisfied in $S_1$, $C_2$ will be satisfied in $S_2^i$. The conjunction of $C_1$ and $C_i$ thus gives us the desired annotation $C_1^i$. That is, $\tau_a(\gamma(S_1|_{C_1^i})) = \tau_a(\gamma(S_1^i|_{C_1})) = \tau_a(\gamma(S_1^i))|_{C_2}$. □

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

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

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

For constant change we need to show that $\#_{R_i}(s_2) = \#_{R_i}(s_1) + \delta$ where $\delta$ is a constant. Recall that a role is a conjunction of abstraction predicates or their negations. Further, because the focus formula $f_{\psi_a}$ consists of pairs of formulas $\Delta_i^+$ and $\Delta_i^-$ 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 $s_1$ is embeddable in $S_1^i$ and embeddings are surjective, the number of elements leaving or entering a role in $s_1$ is the number of those singletons which enter or leave it in $S_1^i$. Hence, this number is the same for every $s_1 \in \gamma(S_1^i)$, and is a constant determined by $S_1^i$ and the focus formulas. □

### 5.1.2 Quality of Abstraction

In order for us to be able to classify the effects of focus operations, we need to impose some quality-restrictions on the abstraction. Our main requirement is that the changes in abstraction predicates

should be characterized by roles: given an abstract structure, an action should be able to affect a certain abstraction predicate only for objects with a certain role. We formalize this property as follows: a formula $\varphi(x)$ is said to be *role-specific in* $S$ iff only objects of a certain role can satisfy $\varphi$ in $S$. That is, there exists a role $R$ such that for every $s \in \gamma(S)$, we have $s \models \forall x(\varphi(x) \rightarrow R(x))$.

We therefore want our abstraction to be rich enough to make the focus formulas, $\psi_a$, role-specific in every structure encountered. The design of a problem representation and in particular, the choice of abstraction predicates therefore needs to carefully balance the competing needs of tractability in the transition graph and the precision required for back propagation.

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

**Proposition 3** *If $\psi$ is uniquely satisfiable in all $s \in \gamma(S)$ and role-specific in $S$ then the focus operation $f_\psi$ on $S$ satisfies focus classifiability w.r.t $\mathcal{C}_I(\mathcal{R})$.*

Proof Since the focus formula must hold for exactly one element of a certain role, the only branching possible is that caused due to different numbers of elements (zero vs. one or more) satisfying the role while not satisfying the focus formula (see Fig. 6). The branch is thus classifiable on the basis of the number of elements in the role. □

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

Therefore any domain-schema whose actions only require the kind of focus formulas specified by the corollary above is amenable to back-propagation. As discussed in the previous section (Theorem 1), focus-classifiability is the main requirement for back propagation. Cor. 1 above provides sufficient conditions for focus classifiability, and thus for back propagation. We exploit this to define a class of domains that allow back propagation. Because these domain-schemas look like linked-lists on abstraction, we call them extended-LL domains:

**Definition 10** *(Extended-LL domains)* An *Extended-LL domain with start structure $S_{start}$* is a domain-schema such that its actions' focus formulas $\psi_{a_i}$ are role-specific, exclusive when not equivalent, and uniquely satisfiable in every structure reachable from $S_{start}$. More formally, if $S_{start} \rightarrow^* S$ and $\Delta_i^{\pm}$ are the focus formulas, then $\forall i, j, \forall e, e' \in \{+, -\}$ we have $\Delta_i^e$ role-specific and either $\Delta_i^e \equiv \Delta_j^{e'}$ or $\Delta_i^e \implies \neg\Delta_j^{e'}$ in $S$.

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

Intuitively, these domain-schemas are those where 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.

**Unary Domains** Another useful class of domains where we can compute preconditions for action sequences can be defined by restricting the domain vocabularies to unary predicates. Such domains satisfy focus classifiability vacuously: all their predicates can be used as abstraction predicates, eliminating the need for focus operations during action application. Action application in these domains therefore cannot result in branching; the only use of focus operations in these domains may be for choosing action operands as illustrated in the first operation in Fig. 7, or sensing. If the initial structure in such domains is a canonical abstraction (i.e. no predicate evaluates to $\frac{1}{2}$ for any element) actions show linear change wrt all the roles. This is because the action update formulas for each predicate evaluate to 0 or 1 for entire roles. Thus, unary domains show linear change and are amenable to back propagation (Theorem 1). For example, suppose our unary predicates were $Red, Blue,$ and $Green$. A "Stain" action that changed the $Green$ predicate to $Blue$ would change role counts as follows:

| Original Role | After Staining | $\Delta$ in count of resultant role |
|---|---|---|
| $Green$ | $Blue$ | $\#_{Green}(S) + \#_{Green \wedge Blue}(S)$ |
| $Green \wedge Red$ | $Blue \wedge Red$ | $\#_{Green \wedge Red}(S)$ |
| $Green \wedge Blue$ | $Blue$ | $\#_{Green}(S) + \#_{Green \wedge Blue}(S)$ |

Algorithms for computing preconditions across linear sequences of actions in unary domains can be developed just like those for extended-LL domains. The only difference is that in unary domains changes in role counts are expressions in terms of old role counts rather than constants. Computation of preconditions for loops in unary domains is an interesting subject for future work.

Unary domains allow us to hard-code interesting problem structure captured by relations of greater arity into simple problem domains that we can handle. This process is syntactically similar to propositionalization, which is commonly used in current classical planners. For example, the transport domain discussed later is encoded as a unary domain.

### 5.1.3 Algorithms for Single Step Backup

Algorithm 1 shows a simple algorithm for computing the role changes due to an action on an abstract structure. 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 works in $O(s)$ operations, where $s$ is the number of distinct roles in the two structures.

**Algorithm 1**: ComputeSingleStepChanges

**Input**: Action transition $S_1 \xrightarrow{f_{\psi_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2$
**Output**: Role-change vector $\Delta$
1 $R \leftarrow$ roles in $S_1^i$ or $S_2^i$
**for** $r \in R$ **do**
2 $\quad count_{old}(r)$= No. of elements with role $r$ in $|S_1^i|$
3 $\quad count_{new}(r)$= No. of elements with role $r$ in $|S_2^i|$
4 $\quad \Delta_r = count_{new}(r) - count_{old}(r)$

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

### 5.1.4 Handling Paths with Loops

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

Let us consider the path from $S$ to $S_f$ including the loop in Fig. 9; analyses of other paths including the loop are similar.

The following proposition formalizes back-propagation in loops:

**Proposition 4** (Back-propagation through loops) *Suppose* $S_1 \xrightarrow{a_2} S_2 \xrightarrow{a_3} \ldots \xrightarrow{a_n} S_n \xrightarrow{a_1} S_1$ *is a loop in an extended-LL domain with a start structure* $S_{start}$*. Let the loop's entry and exit structures be* $S$ *and* $S_f$ *(Fig. 9). We can then compute an annotation* $C(l)$ *on* $S$ *which selects the structures that will be in* $S_f|_{C_f}$ *after* $l$ *iterations of the loop on* $S$*, plus the simple path from* $S$ *to* $S_f$*.*

Proof Our task is to find an annotation for the structure $S$ which allows us to reach $S_f|_{C_f}$, where $C_f$ is a given annotation. Since we are in an extended-LL domain, every action satisfies constant change.

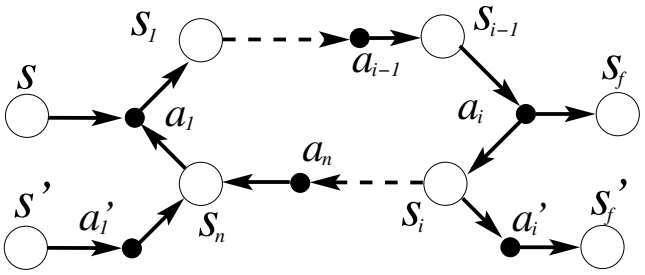


Figure 9: Paths with a simple loop. Outlined nodes represent structures and filled nodes represent actions.

Let the vector $\bar{R} = \langle \#R_1, \#R_2, \ldots, \#R_m \rangle$ consist of role-counts. Conceptually, we can include counts for all the roles; in practice, we can omit the irrelevant ones. 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 the count of the branch-role at action $a_i$ is $\bar{R}_{b_i}$. If there is no branch at action $a_i$, we let $b_i = d$, some unused dimension. Let $\Delta^i$ denote the role-count change vector for action $a_i$. Let $\Delta^{1..i} = \Delta^1 + \Delta^2 + \cdots + \Delta^i$.

Before studying the loop conditions, consider the action $a_n$ in Fig. 9. Suppose that the condition that causes us to stay in the loop after action $a_n$ is that $\#R_{a_n} = 1$. Then the loop branch is taken during the first iteration starting with role-vector $\bar{R}^0$ if $(\bar{R}^0 + \Delta^{1..n})_{b_n} = 1$. This branch is taken in every subsequent loop iteration iff $\Delta^{1..n}_{b_n} = 0$, in which case such actions effectively do not have any branches leaving the loop.

The condition for a full execution of the loop starting with role-count vector $\bar{R}^0$ then is:

$$\begin{array}{lcl} (\bar{R}^0 + \Delta^{1..1})_{b_1} & \circ & c_1 \\ (\bar{R}^0 + \Delta^{1..2})_{b_2} & \circ & c_2 \\ & \vdots & \\ (\bar{R}^0 + \Delta^{1..n})_{b_n} & \circ & c_n \end{array}$$

$\circ$ is one of $\{>, =, <\}$ depending on the branch that lies in the loop. The only execution dependent term in this set of inequalities is $\bar{R}^0$. Let us call these inequalities $\mathsf{LoopIneq}(\bar{R}^0)$. For executing the loop $l$ times, the condition becomes

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

where $\bar{R}^{l-1} = \bar{R}^0 + (l-1) \times \Delta^{1..n}$. These two sets of conditions ensure all the intermediate loop conditions hold, because the changes are linear. For an exit during the $l^{th}$ iteration, we have the conditions:

$$\begin{array}{lcl} & \mathsf{LoopIneq}(\bar{R}^0) & \\ & \mathsf{LoopIneq}(\bar{R}^{l-2}) & \\ (\bar{R}^{l-1} + \Delta^{1..1})_{b_1} & \circ & c_1 \\ (\bar{R}^{l-1} + \Delta^{1..2})_{b_2} & \circ & c_2 \\ & \vdots & \\ (\bar{R}^{l-1} + \Delta^{1..i})_{b_i} & \bullet & c_i \end{array}$$

where the last inequality corresponds to the condition for the desired branch. These conditions on the role vector $\bar{R}^0$ at $S$ constitute $C(l)$. Note that in order to compute this set of conditions we only need to compute at most $n$ different $\Delta^{1..i}$ vectors. Using the algorithm for computing one step change vectors $\Delta^i$ (Algorithm 1), $C(l)$ can be computed in $O(s \cdot n_l)$ time, where $s$ is the maximum number of roles in a structure in this loop, and $n_l$ is the number of actions in the loop.

□

Note that final set of inequalities in the proof given above include the exact role counts for all roles after $l$ iterations of the loop. Together with the ability to compute changes in role counts across linear sequences of actions (Cor. 2,Prop. 2), this means we can compute not only whether a path with simple loops can take a certain concrete structure to a desired goal structure, but also the *exact* number of times we need to go around each loop in the path, in order to reach the desired structure with desired role counts.

**Corollary 3** (Number of loop iterations) *In extended-LL domains, we can find the exact number of iterations of each loop required to reach goal states using generalized plans with simple loops.*

## 5.2 Synthesis of Generalized Plans

In this section we present the fundamental algorithm for searching for generalized plans with precondition finding routines for extended-LL domains and show how it finds generalized plans for the unit delivery problem. The overall algorithm can be applied to any system of state abstraction where methods for determining preconditions of action sequences are available. To our knowledge, this is the first approach for finding algorithm-like plans without using automated deduction. We present the algorithm and illustrate how it works. An efficient implementation of the algorithm is underway and a comprehensive evaluation of its performance is thus left for future work. Heuristics for finding paths to the goal in the abstract state space would also be useful in improving the algorithm's performance in practice. Such heuristics for search in an abstract state space also present a new area for future research.

Once we have the state abstraction and abstract action mechanisms, they can be used to conduct a search in the finite abstract state space. The size of this space is determined by the number of abstraction predicates and relations in the domain. Although the number of possible roles is exponential in the number of abstraction predicates, not all of those roles can coexist. For example, in the unit delivery problem, there are 4 possible roles for crates ($crate$; $crate \wedge chosen$; $crate \wedge delivered$; $crate \wedge chosen \wedge delivered$), 1 possible truck role, and 2 possible location roles ($location$; $location \wedge targetDest$). Consequently, the $at()$ relation can be instantiated over at most $(4 + 1) \times 2 = 10$ combinations of operands. Since only a chosen crate can be loaded, the $in()$ relation has only 1 possible instantiation. Each of these 11 total instantiations can have one of three truth values, and for each complete assignment of truth values, there can be at most $3^6$ ways of assigning an element-type (singleton, summary, or non-existent) to the six crate and location roles. This gives us an upper bound of $3^{17}$ reachable abstract states.

On the other hand, if we do not use abstraction, a loose lower bound on the number of reachable states is $(n_l + 1)^{n_c+1}$, assigning one of $n_l$ delivery locations and the dock to the $n_c$ crates and the truck. The number of abstract states is smaller even if we have only 8 crates and 8 delivery locations! In the transport example discussed later, the number of concrete states far exceeds the number of abstract states if there are more than 9 total items (monitors or servers). Since current classical planners can solve even larger instances of this problem, abstract state search presents a viable and more efficient alternative.

The algorithm for synthesis of generalized plans using search is presented as ARANDA-Synth (Algorithm 2). It proceeds in phases of search and annotation. During the search phase it uses the procedure $getNextSmallest$ to find a path $\pi$ (with non-nested loops only) from the start structure to a structure satisfying the goal condition. Heuristics can be used to aid the efficiency of the search. During the annotation phase, it uses the procedure $findPrecon$ to find the precondition for $\pi$, if we are in an

**Algorithm 2**: ARANDA-Synth

**Input**: Generalized Planning Problem $\langle S_0, \mathcal{D}, \varphi_g \rangle$, where $S_0$ is an abstract structure
**Output**: Generalized Plan $\Pi$
$\Pi \leftarrow \emptyset$
**while** $\Pi$ *does not cover* $\mathcal{I} = \gamma(S_0)$ *and* $\exists \pi$*: unchecked path to goal* **do**
  $\pi \leftarrow$ `getNextSmallest`$(S_0, \varphi_g)$ $C_\pi \leftarrow$ `findPrecon`$(S_0, \pi, \varphi_g)$
  **if** $C_\pi$ *is not subsumed by labels on current edges from* $s_\Pi$ **then**
    Add edge from $s_\Pi$ to $s_\pi$ with label $C_\pi$

extended-LL domain. Methods for $findPrecon$ were described in the previous section. The resulting algorithm can be implemented in an any-time fashion, by returning plans capturing more and more problem instances as new paths are found.

Surprisingly, in the examples used in this paper we found that only a few plan branches (three for the unit delivery and transport problems, and five for the more complicated blocks word problem) were sufficient to cover the entire space of solvable instances.

Procedure $getNextSmallest$ (Algorithm 3) works by passing path messages along action edges between structures. A path message consists of the list of structures in the path, starting from the initial structure $S_0$. Loops in the path are represented by including the loop's structure list, in the correct order, as a member of the path message. For example, the path message for the path from $S$ to $S_f$ including the loop in Fig. 9 is represented as $[S, [S_1, S_2, \ldots, S_n], S_{i-1}, S_f]$. The first structure after a loop in the message denotes the exit from the loop.

$MsgInQ(S)$ and $MsgOutQ(S)$ functions index into the message queues for the given structure. The procedure starts by sending the message containing the empty path ([ ]) to $MsgInQ(S_0)$. When $getNextSmallest$ is called, structures are activated, i.e. their messages are processed and dispatched until a path to the goal structure is found (lines 1-4). Path messages are generated by $genMsg(S, m)$ which takes a structure and the incoming message, and dispatched by $sendMsg(S, m)$, which takes the destination structure and the message to be sent to it as arguments. Message size is not a serious overhead for message passing, which can be effectively executed using only the message pointers.

During every activation, all the messages in every structures $MsgInQ$ and $MsgOutQ$ are processed using $genMsg$ and $sendMsg$ (lines 5-8). These operations can be carried out in parallel. $sendMsg$ makes sure that messages only go to those structures which won't create a nested loop. $genMsg(S, m)$ checks if the incoming message creates a simple loop by checking if $S$ is already present in the message without its loop elements (line 9). If a simple loop is created, the appropriate loop element is added to the path message to obtain $m'$. In this case, two kinds of messages are sent out: one ending with the newly formed loop (line 11), which is sent only to $S$'s successor in the loop, and the other denoting an exit from the loop at $S$ itself (line 12), which is only sent to the non-loop neighbors of $S$ by $sendMsg$. If a loop is not formed, $S$ is simply added to the end of the path message and placed in $MsgOutQ(S)$ (lines 14,15). At this stage, it is also checked if $S$ is a goal structure. If it is, the newly formed path is returned, and this phase of activation ends. In practice, this phase can be optimized by interleaving precondition evaluation and the search for new paths: if a newly found path includes structures that occur in paths found earlier, its preconditions can first be found only up till those structures. Unless they increase the coverage on those structures over the earlier paths, this path can be discarded and the activation phase can continue until a useful path is found.

Once the messages have been generated, subroutine $sendMsg$ looks at each message, and sends it to successor structures that will not create a nested loop (lines 18-20), or to successor structures within the newly formed loop (line 22-24).

This algorithm generates all paths with simple loops to the goal, while ordering them across activation phases in a non-decreasing order of lengths. The length of a path with a loop is counted by including one full iteration of the loop, and a partial iteration until the structure where it exits.

Together with the facts that the state space is finite and that we can find preconditions in extended-LL domains, Algorithm 2 realizes the following theorem.

**Algorithm 3**: getNextSmallest($S_0, \varphi_g$)

```
   { /* All data structures are global, and maintained across calls
   Initialize:
   MsgInQ(S_0) ← {[ ]}
   MsgOutQ(S_0) ← ∅
   Do this initialization for every structure when encountered for the first time.   */
 1 π ← ∅; Pause← False
 2 while π is not a path to φ_g do
 3 |   forall S with non-empty MsgInQ or MsgOutQ do
 4 |   |   Activate(S)
   }
Procedure Activate(S) {
 5 while MsgInQ(S) ≠ ∅ and not(Pause) do
 6 |   m ← PopQ(MsgInQ(S)); genMsg(S,m)
 7 forall MsgOutQ(S) ≠ ∅ do
 8 |   m ← PopQ(MsgOutQ(S)); sendMsg(S,m)
   }
Procedure genMsg(S, m) {
 9 if S ∈ m \ loops then
   |   /* m \ loops = [S_0, S_1, ..., S_{k-1}, S, ..., S_l]; A loop is formed   */
10 |   m'← createLoop(m,S)
   |   /* m' ← [S_0, S_1, ..., S_{k-1}, [S, S_{k+1}, ..., S_l]]   */
11 |   PushQ(MsgOutQ(S),m')
12 |   m_S ← append(m',S)
13 else                                                    /* Loop not formed */
14 |   m_S ← append(m,S)
15 PushQ(MsgOutQ(S), m_S)
16 if S ⊨ φ_g then
   |   π ← m_S;Pause← True                                  /* The next path */
   }
Procedure sendMsg(S, m) {
17 if not newLoop(m) then
18 |   for S_n ∈ possNextStruc(S) do
19 |   |   if afterLastLoop(m, S_n) then
   |   |   |   /* Adding S_n won't cause a nested loop   */
20 |   |   |   rcvMsg(S_n, m)
21 else                                                /* m's last element is a loop */
22 |   if S not last in loop then
23 |   |   S_n ← successor of S in the loop
24 |   |   rcvMsg(S_n,m)
   }
Procedure rcvMsg(S, m) {
25 PushQ(MsgInQ(S), m)
   }
```

**Algorithm 4**: ARANDA-Learn

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

$S_0 \leftarrow canAbs(S_0^\#)$; $\Pi \leftarrow \pi$; $C_\Pi \leftarrow \top$
$\{S_0; a_1, \ldots, S_{n-1}; a_n, S_n\} \leftarrow Trace(\{S_0^\#; a_1, \ldots, S_{n-1}^\#; a_n, S_n^\#\})$
$\Pi \leftarrow$ formLoops$(S_0; a_1 \ldots, S_{n-1}; a_n, S_n)$
**if** $\exists C \in \mathcal{C}_I(\mathcal{R}) : S_n|_C \models \varphi_g$ **then**
  $C_\Pi \leftarrow$ findPrecon$(S_0, \Pi, \varphi_g)$
  $\Pi_r \leftarrow$ refineLoops$(\Pi)$
return $\Pi_r, C_\Pi$

**Theorem 3** (Generalized planning for extended-LL domains) *Algorithm 2 finds most general plans amongst those with simple loops in extended-LL domains.*

**Example 9** Fig. 10 shows the structures and actions in a path found in the unit delivery domain using the ARANDA-Synth algorithm. The initial structure represents all possible problem instances with any number of crates and locations (at least one of each). The abstract structure after unloading 3 crates is identical to the one after unloading 4 crates. This appears in the path as a simple loop. While this path to the goal works for instances with at least 4 crates, paths for fewer crates do not include loops, have fewer action edges and are thus found before the shown path.

Possible exits due to branching effects are not shown in the figure except for the last $choose(Crate)$ action; they have to be taken into account during precondition evaluation.

If speculative plans are acceptable, this algorithm can be applied more broadly without the need for finding preconditions. All we need to guarantee is that the loops in the generalized plan always terminate. Static analysis methods can be used for this purpose. Methods such as Terminator (Cook et al., 2006) have been shown to determine termination for a wide class of program loops. Their use of linear inequalities for representing changes caused by program statements are also suitable for representing the changes in role counts. Such methods could also prove useful for future work in calculating loop preconditions under linear change.

## 5.3 Generalizing Example Plans

In this section we present our approach for computing a generalized plan from a plan that works for a single problem instance. An earlier version of this algorithm was described in (Srivastava et al., 2008). As discussed in section 3, our approach for plan generalization is to apply the given plan in the abstract state space, starting with an abstract start state and extracting loops in the resulting state-action trace. Because of abstraction, recurring properties become easily identifiable as repeating abstract states. The procedure is shown in Algorithm 4.

Suppose we are given a concrete example plan $\pi = (a_1, a_2, \ldots, a_n)$ for a concrete state $S_0^\#$. Let $S_i^\# = a_i(S_{i-1}^\#), i > 0$. Let $S_0$ be any structure which makes the resulting domain extended-LL, and for which we need a generalized plan; the canonical abstraction of $S_0^\#$ forms a natural choice. To obtain the generalized plan, we first convert $\pi$ into a sequence of action operators which use as as their argument(s) any element having the role (or even just the type) of the concrete action's argument(s). We successively apply the operators from this sequence to $S_0$, at every step keeping only the abstract structure $S_i$ that embeds the concrete structure $S_i^\#$ for that step. This process, which we call *tracing*, is implemented in the $Trace$ subroutine.

The $formLoops$ subroutine converts a linear path 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. One way of implementing this routine is by making a single pass over the input sequence

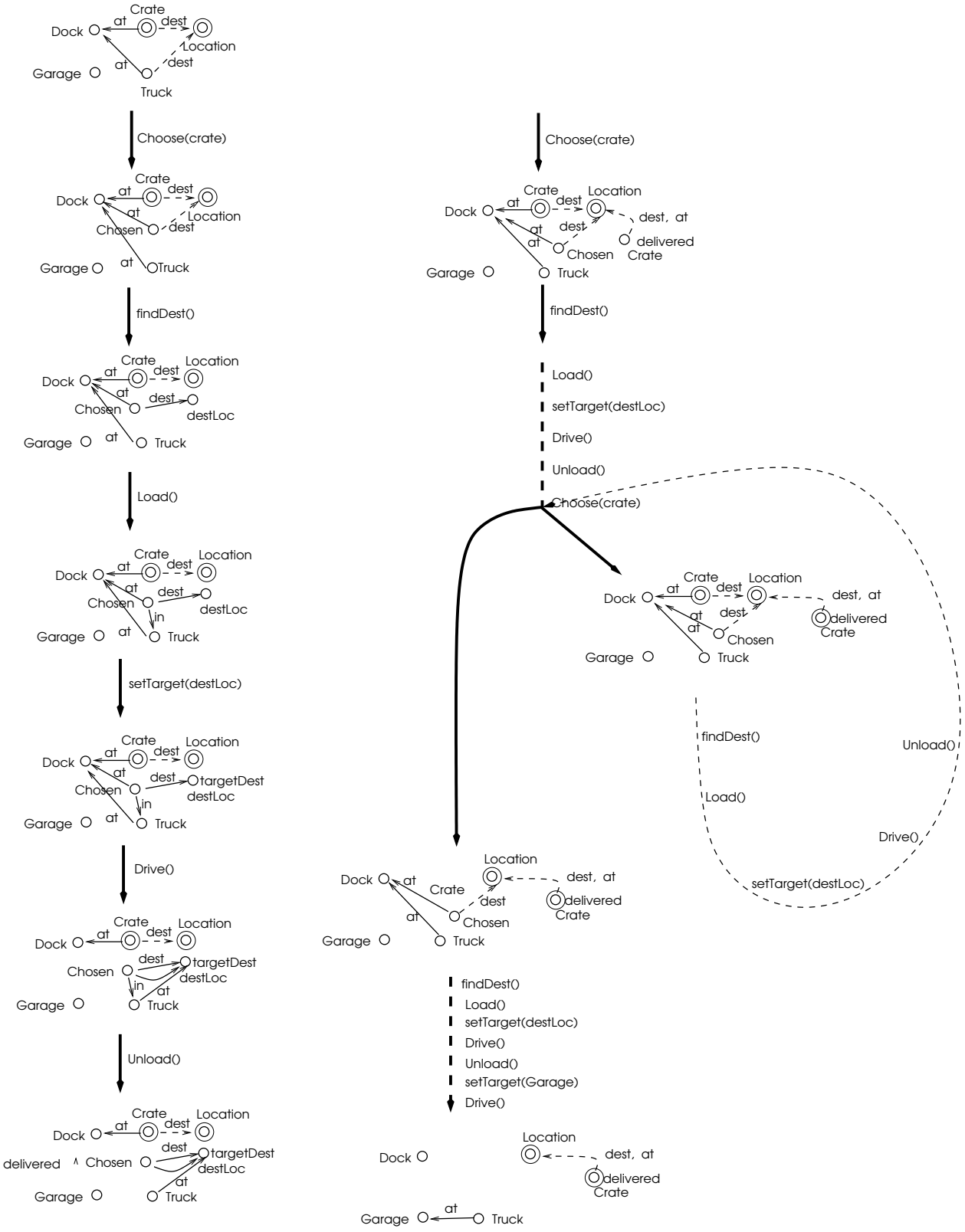


Figure 10: Example for ARANDA-Synth

of abstract-state and instantiated action pairs, and adding back edges whenever a pair $\langle S_j, a\rangle$ is found such that $\langle S_i, a\rangle = \langle S_j, a\rangle (i < j)$, and $\langle S_i, a\rangle$ is not part of, or behind a loop. Note that the instantiated actions include their role arguments in these pairs. Such a repeated pair indicates that some properties that held in the concrete state after application of $a_i$ were true again after application of $a_j$ as witnessed by $S_j$, and further, that in the example plan, the same action $a$ was applied at this stage. This is our fundamental cue for identifying an unrolled loop – as long as an identical abstract state can be reached again, the same actions can be applied. This is true because the abstraction affords accurate evaluations for action preconditions. Loops identified in this manner are kept only if they make progress, which is determined during the precondition evaluation phase.

Continuing with the $formLoops$ algorithm, structures and actions following $S_j$ are merged with those following $S_i$ if they are identical; otherwise, the loop is exited via the last action edge. This method produces one of the possibly many simple-loop paths from $\pi$; we could also produce all such paths. $formLoops$ thus gives us a generalization $\Pi$ of $\pi$. If the final abstract structure $S_n$ or its restriction satisfies the goal, we have a general plan and we can proceed to find its preconditions (steps 4 and 5) using the $findPrecons$ subroutine. Note that non-progressive loops will be identified in this phase as they will register a net change of zero in the role counts. The $refineLoops$ procedure uses this information to remove any loops that do not make progress.

**Example 10** Consider the unit delivery problem. Suppose we were given an example plan that delivered five crates to at least two different destinations using the following sequence of actions $Choose, findDest, Load, setTarget, Drive, Unload$. Replacing real arguments by their roles, and tracing out the plan on the canonical abstraction of the initial structure yields exactly the path shown in Fig. 10! The included loop can be identified using $formLoops$, as described above.

To summarize our instantiation of the paradigm for generalized planning presented in this paper, we use abstraction based on 3-valued logic to convert the infinite set of state spaces into a finite state space for searching for generalized plans. The concrete action mechanism is enhanced with focus and coerce operations in order to obtain a more precise action mechanism for the abstraction. An abstract structure is used to capture the class of initial states. Unions of abstract structures can be used to capture richer classes, but for simplicity we only deal with single initial abstract structures in this paper.

# 6 Related Work

There have been very few directed efforts towards developing planners 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 been developed handle this problem by extracting useful information from existing plans. However, to our knowledge no approach provides approaches for direct search for generalized plans and for plan generalization in a unified manner. In fact, few approaches for directly searching for generalized plans exist at all. In this section we discuss other work on finding plans that could be understood as generalized plans. We also related uses of abstraction, both in planning and software model checking.

## 6.1 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 simultaneously searching in infinitely many state spaces; further, our abstraction is not propositional and does not satisfy some of their planning graph based requirements.

## 6.2 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 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.

## 6.3 Case Based Planning

Case based planning (CBP) (Hammond, 1996; Spalzzi, 2001) attempts to solve the problem of handling instances of conceptually similar problems by maintaining databases of past planning problems and their solutions. To this end, CBP research focusses on methods for plan storage for efficient retrieval of related plans and plan adaptation or repair so as to make retrieved plans applicable to new problems. CBP approaches typically incur significant costs for plan adaptation and instantiation. In contrast, generalized plans are directly applicable to the problem instances they are designed for and require minimal applicability tests.

## 6.4 Plans With Loops

Perhaps the closest, and the most significant amongst these approaches is DISTILL (Winner and Veloso, 2003) and its more recent looping version, LoopDISTILL(Winner and Veloso, 2007). DISTILL is a system for learning domain specific planners (dsPlanners) through examples. A dsPlanner can be used to generate plans for different problem instances, bringing it close to the goal of generalized planning. DISTILL works by annotating example plans with partial orderings reflecting every operator's needs and effects. This annotation is used to compile parametrized versions of example plans into a dsPlanner. This approach is limited to plan learning however, and only creates speculative plans.

KPLANNER (Levesque, 2005) is an approach for iterative planning. It proceeds by iteratively finding plans for bigger problem instances and attempts to find loops by observing recurring patterns. This approach is limited to problems with a single planning parameter, the only variable whose value can differ across different instances. KPLANNER therefore cannot handle any of the problems demonstrated in this paper; however, it is better at dealing with numeric variables.

## 6.5 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 could also be classified as speculative.

## 6.6 Contingent Planning

Contingent planning (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005) can be seen as an instance of generalized planning. The class of initial instances now represents a 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. After abstraction, the search problem for contingent planning is similar to

that for generalized planning, with the exception of including loops and finding preconditions. Finding preconditions is not necessary because contingent planners expect the entire initial belief state to be solvable.

The input expected by current contingent planners differs significantly from a generalized planning problem, which makes them unsuitable for generalized planning. Contingent planners typically expect a finite set of possible states. It is also difficult to model states with different numbers of objects as belief states. One way of doing this could be by using an upper bound to include a fixed number of objects in the domain, for each of which a *present* predicate may or may not be true. However, during planning it is difficult to generate properties for newly discovered *present* objects and their relationships with older objects. Because they typically exclude cycles from search, current contingent planners can *never* find a most general plan for the problems discussed in this paper.

### 6.7 Abstraction in Software Model Checking

Software model checking is the problem of verifying that the behavior of a formal specification or program satisfies desirable properties. 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).

## 7 Implementation and Results

We implemented a prototype for ARANDA-Learn using TVLA as an engine for computing results of action sequences. The implementation proceeds in four phases. In the first phase, the example plan is executed on the concrete initial state that it came with. This gives us a sequence of concrete state and action pairs. In the second phase, the example plan is executed on an abstract start state, with the real operands replaced by their roles. In the third phase, for every action we select the unique abstract result structure that embeds the corresponding result structure from the concrete run (tracing). Finally, loops are identified as described in Section 5.3.

The implementation was developed using Python. We used the `pydot` utility based on the `Pyparsing` module for parsing the transition graphs produced by TVLA for the third phase. The parsing operation is the most expensive in our implementation, taking about 5 minutes to parse a file containing $2,500$ structures. This is extraneous to our algorithm: an exponentially faster system can be implemented by following the tracing procedure described in the section on our Algorithm. This requires a better interface with TVLA.

### 7.1 Results

We ran ARANDA-Learn on some practical problems derived from classical planning benchmarks. We summarize the problems and the results below. In each of these problems, the class of initial instances was represented using a three-valued structure. Each of these generalized plans can also be found directly by ARANDA-Synth. In fact, ARANDA-Synth finds complete generalized plans for these problems by finding the extra 2 to 3 branches required to cover the smaller problem instances missed by the generalizations presented below.

#### 7.1.1 Delivery

The input example plan delivered five objects to two different locations. The abstract start structure was shown in Fig. 10. 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 5.1 to compute the preconditions

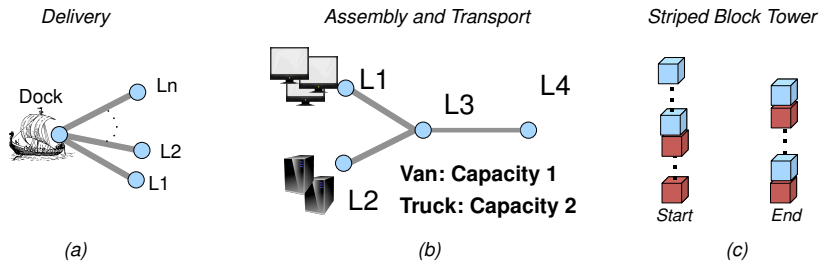


Figure 11: Example problem domains

for this plan as $\#(item) \geq 2$. In fact, here and in all the following examples the preconditions also show how many loop unrollings there will be in a plan execution (e.g. $\#(item) = l + 2$, where $l \geq 0$ is the number of loop iterations). In finding preconditions, we assumed that the sensing action for finding a crate's destination does not effect goal reachability. This is well founded because every crate is known to have a unique destination.

### 7.1.2 Assembly and Transport

We have two source locations L1 and L2, which have a variable number of monitors and servers respectively (Fig. 11(b)). There are two types of transport, a van at L1 and a truck at L2 with capacities 1 and 2 respectively. For simplicity, in this example we have exactly one vehicle of each kind. 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. The example plan for six pairs of such items worked as follows: the van moves a monitor from L1 to L3 and returns to L1; the truck then takes a server from L2 to L4, while picking up the monitor at L3 on the way.

We represented this domain without using any binary relations. Fig. 14 shows the abstract structure used for tracing. Fig. 13 shows the main loop discovered by our algorithm. Preconditions can be computed as $\#(monitor \wedge atL_1) = \#(server \wedge atL_2) = l + 2$, where again, $l$ is the number of iterations of the loop. This is a good illustration of how preconditions based on just role-counts can capture required relationships between objects of different roles.

### 7.1.3 Striped Block Tower

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 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 misplaced block, or 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 indicate that for the plan to work, we must have $\#(Red) = \#(Blue) = l + 4$, where $l$ is the number of iterations of the loop for alternating stacking red and blue blocks. The representation for this problem can also be used to model, and solve program synthesis problems like reversing and traversing link lists. This gives us a method for solving such problems without the computational overheads of automated deduction.

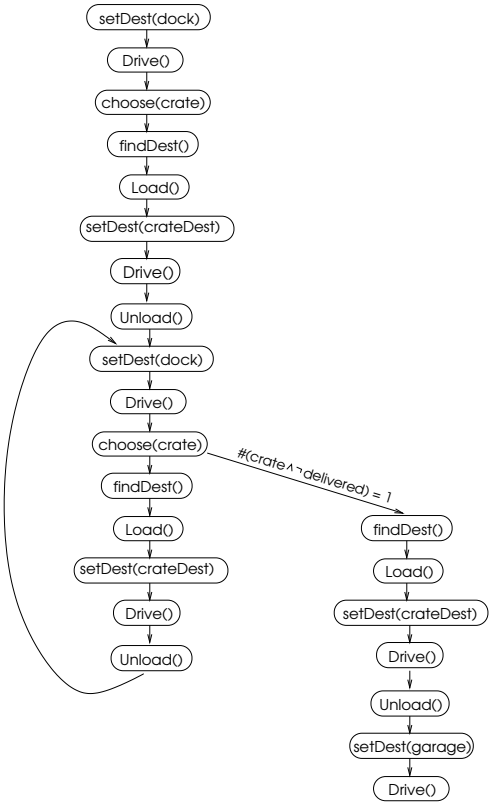


Figure 12: Generalized plan for unit delivery problem instances with at least 2 crates.

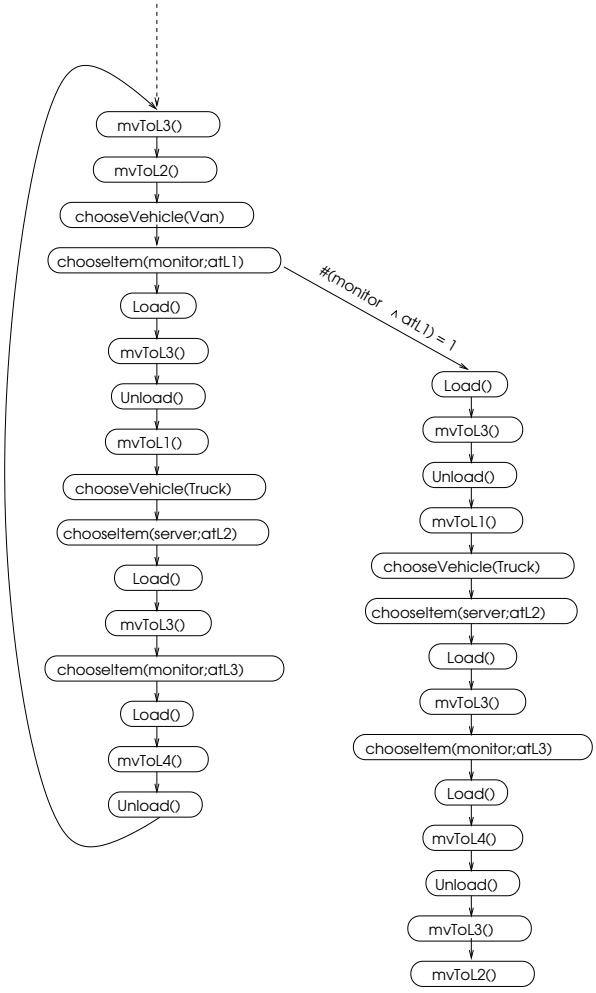


Figure 13: Main loop for Assembly and Transport

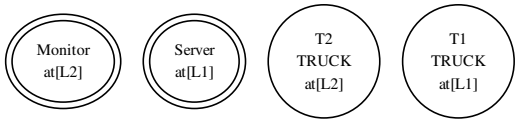


Figure 14: Start structure for assembly and transport

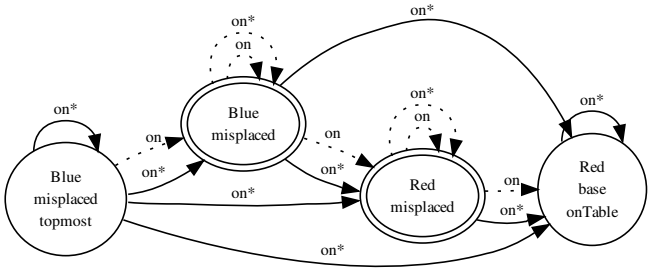


Figure 15: Start structure for striped block tower

## 7.2 Comparison

A summary of the timing results for ARANDA-Learn can be seen in Fig. 17. The parsing time for phase 3 (extraneous to our algorithm) in ARANDA-Learn was between 4 and 6 minutes for these examples and is not shown in the plots. Loop finding times were less than a second.

We compared our prototype's plan-generalization times with planning times of winners of the last International Planning Competition (IPC-5, 2006): SGPlan5 (a non-optimal planner) and SATPLAN06 (an optimal planner). We had to explicitly ground our problem encodings for these planners, which sometimes made the problem easier. For example, the goal position of each block in the Striped Tower problem became explicitly known. Although SGPlan5 allowed quantified goals, their use make the planner very unscalable – with a quantified goal formula stating the alternating color condition, it ran for 1500 seconds without solving a Striped Tower problem instance with 10 pairs of blocks.

We present a summary of the results in Fig. 18. The results clearly illustrate the utility of learning generalized plans. We used vanilla versions of the planners for this comparison. The sensing aspect of the unit delivery problem could not be accurately translated without quantified goals, and was thus not included in the comparison. All but the larger runs for SGPlan were carried out on a 1.6GHz Pentium Dual Core machine with 1.5GB RAM. SGPlan with 30 or more pairs of objects had to be run on a 2GHz machine with 2GB RAM.

## 7.3 Evaluation

In this section we evaluate the generalized plans found by ARANDA-Learn, and described in the previous section (Table 1). The plan instantiation and applicability costs can effectively be reduced to $O(1)$, using the natural approach of maintaining lists of objects of each role. Plan optimality for Transport 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. However, our plan has a better makespan (and its loss due to generalization is 0). As discussed in Section 2.1, depending on the situation, the optimality of a generalized plan can be defined to measure its performance over a variety of cost measures, including makespan.

### 7.3.1 Other Planners

A head to head comparison with results from other planners was not possible. Of the two most relevant approaches, KPLANNER could not handle our examples, and DISTILL's implementation was not available. We provide a summary of the features of these planners in Table 2.

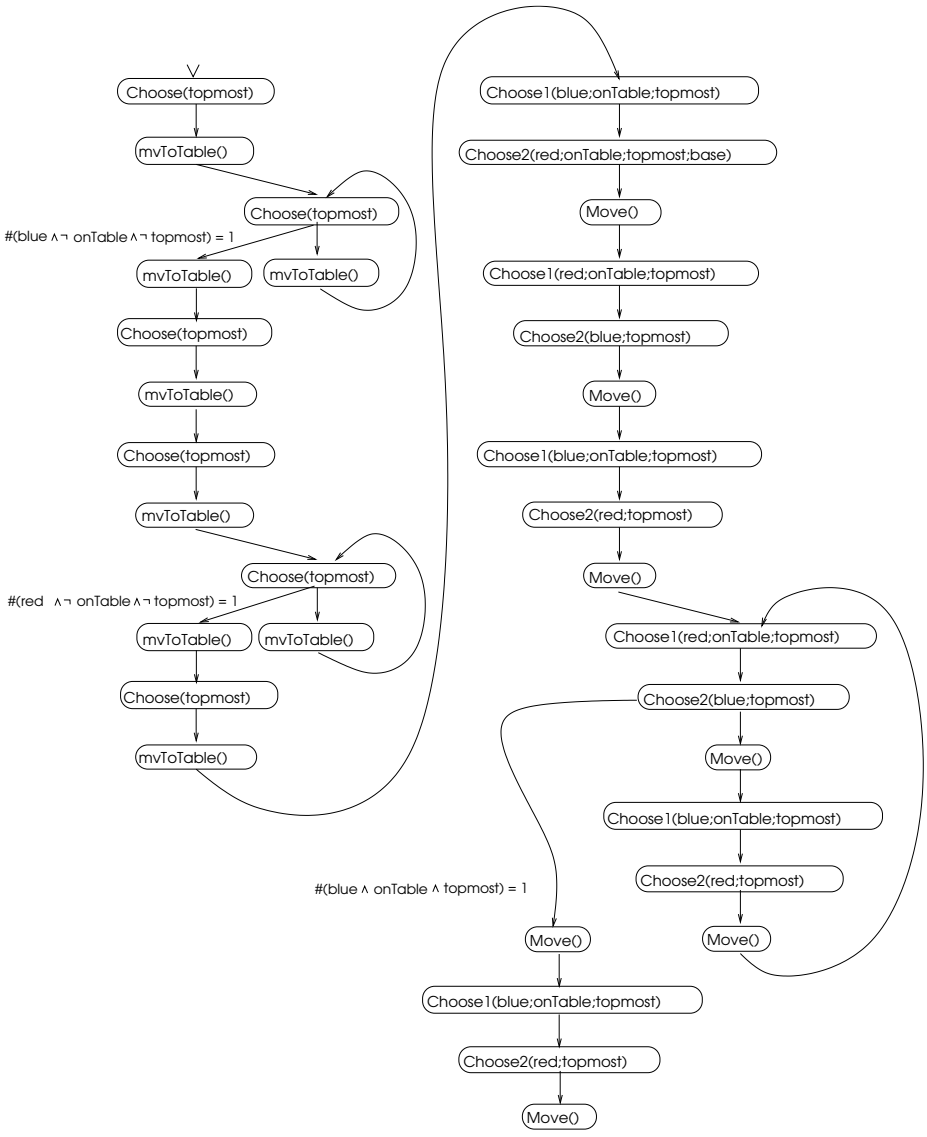


Figure 16: Generalized Plan for Striped Block Tower

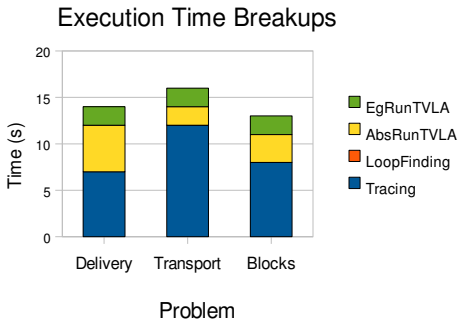


Figure 17: Break-ups of ARANDA-Learn's plan generalization times

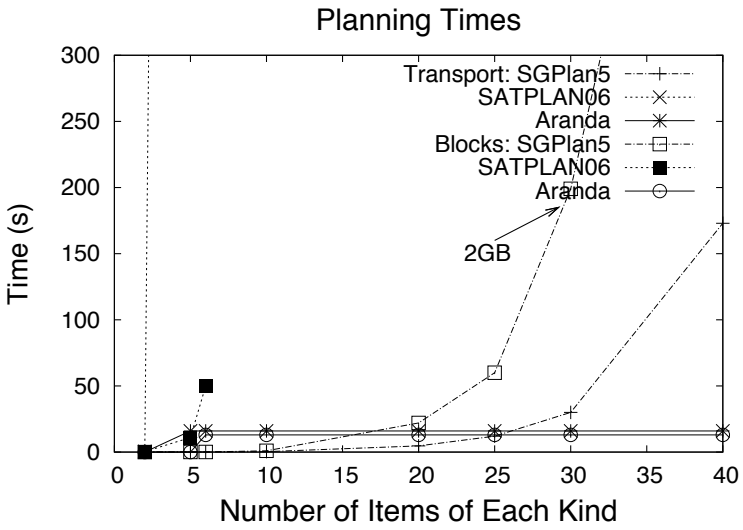


Figure 18: Comparison of plan generalization and planning times

# 8 Conclusion and Future Work

In this paper we formalize the notion of generalized planning and present new algorithms for generalizing example plans and searching for generalized plans from scratch. We identify a class of domains where our methods for computing plan applicability are proven to work. Our experimental results illustrate the potential and applicability of these techniques.

Our work establishes a new paradigm for solving generalized planning problems and presents several directions for future work. Searching in abstract state spaces presents new challenges and requires a different class of heuristics. We currently only allow paths with simple loops in our generalized plans. Efficiently representing preconditions of nested loops is a natural extension, along with the extension of our methods to wider classes of domains. 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).This paper works under the restriction of constant change in our actions. We are extending methods for finding loop preconditions under linear change, as seen in unary domains. More general notions of roles, using description logic can also help in increasing precision and the ability for focus classifiability.

| Evaluation Metric | Blocks | Transport | Delivery |
|---|---|---|---|
| Domain Coverage | $p \geq 4$ | $p \geq 2$ | $n_i \geq 2$ |
| Instantiation Cost | $O(p)$ | $O(p)$ | $O(n_i)$ |
| Applicability Test | $O(p)$ | $O(p)$ | $O(n_i)$ |
| Optimality | 1 | $\frac{9p}{11p} = 0.81$ | 1 |
| Loss of Optimality | 0 | 0 | 0 |

Table 1: Evaluation of some generalized plans. All $O(p)$ and $O(n_i)$ costs can be reduced to $O(1)$ in implementation (see Sec. 7.3). $p$=No. of item pairs, $n_i$=No. of items

| Planner Property | ARANDA | DISTILL | KPLANNER |
|---|---|---|---|
| Plan Synthesis | Yes | No | Yes |
| Without Preprocessing | Yes | No | Yes |
| Precondition Evaluation | Yes[a] | No | No |
| Loop Variables | Unbounded | ?[b] | 1 |

Table 2: Summary of abilities of some planners

[a] extended-LL domains

[b] LoopDISTILL has been demonstrated only on examples with a single loop variable, but extensions may be possible.

## Acknowledgments

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

## References

Bonet, B., Geffner, H., 2000. Planning with incomplete information as heuristic search in belief space. In: Proc. 6th International Conf. on Artificial Intelligence Planning and Scheduling.

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.

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.

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

Feng, Z., Hansen, E. A., 2002. Symbolic Heuristic Search for factored Markov Decision Processes. In: Proc. of The Eighteenth National Conference on Artificial intelligence.

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.

Fikes, R., Nilsson, N., 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Tech. rep., AI Center, SRI International, sRI Project 8259.

Hammond, K. J., 1996. Chef: A Model of Case Based Planning. In: Proc. of the Thirteenth National Conference on Artificial Intelligence.

Henzinger, T. A., Jhala, R., Majumdar, R., Sutre, G., 2002. Lazy Abstraction. In: Symposium on Principles of Programming Languages.

Hoey, J., St-Aubin, R., Hu, A., Boutilier, C., 1999. SPUDD: Stochastic planning using decision diagrams. In: Fifteenth Conference on Uncertainty in Articial Intelligence.

Hoffmann, J., Brafman, R. I., 2005. Contingent Planning via Heuristic Forward Search with Implicit Belief States. In: Proc. The International Conference on Automated Planning and Scheduling.

Hoffmann, J., Sabharwal, A., Domshlak, C., 2006. Friends or Foes? An AI Planning Perspective on Abstraction and Search. In: Proc. The International Conference on Automated Planning and Scheduling.

IPC-5, 2006. Fifth International Planning Competition. `http://zeus.ing.unibs.it/ipc-5/`. URL `http://zeus.ing.unibs.it/ipc-5/`

Knoblock, C. A., 1991. Search Reduction in Hierarchical Problem Solving. In: Proc. of the Ninth National Conference on Artificial Intelligence.

Levesque, H. J., 2005. Planning with Loops. In: Proc. of the International Joint Conference on Artificial Intelligence.

Loginov, A., Reps, T., Sagiv, M., 2006. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In: Proc. of Static Analysis Symposium.

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, volume 2937 of LNCS.

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.

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

Srivastava, S., Immerman, N., Zilberstein, S., 2008. Learning Generalized Plans Using Abstract Counting. In: Proc. of the Twenty-Third AAAI Conference on AI.

Winner, E., Veloso, M., 2003. DISTILL: learning domain-specific planners by example. In: Proc. of the International Conference on Machine Learning.

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.

# A Canonical Abstraction

Canonical abstraction (Sagiv et al., 2002) groups states together by only counting the number of elements of a role up to two. If a role contains more than one element, they are combined into a *summary element*.

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

| No. | Constraint |
|---|---|
| 1. | $on(x, y)$ is functional. |
| 2. | inverse of $on(x, y)$ is functional. |
| 3. | a block is either $onTable$ or on another block. |
| 4. | $on*$ is the transitive closure of $on$. |

Table 3: Integrity constraints used in the blocks world (see Fig. 19)

**Definition 11** *(Embedding)* Let $S_1$ and $S_2$ be two structures and $f : |S_1| \to |S_2|$ be a surjective function. $f$ is an *embedding* from $S_1$ to $S_2$ ($S_1 \sqsubseteq^f S_2$) iff for all relation symbols $p$ of arity $k$ and elements, $u_1, \ldots, u_k \in |S_1|$, $[\![p(u_1, \ldots, u_k)]\!]^{S_1} \prec [\![p(f(u_1), \ldots, f(u_k))]\!]^{S_2}$.

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

**Definition 12** *(Canonical Abstraction)* The embedding of $S$ into its *canonical abstraction* wrt the set $A$ of abstraction predicates is the map:

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

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

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

# B Focus and Coerce

The focus operation on a structure $S$ with a formula $\varphi(\bar{x})$ attempts to produce the possible structures embeddable in $S$ with definite truth values for all possible instantiations of the variables $\bar{x}$. Fig. 19 shows the focus operation on a tower of blocks in the blocks world. The abstraction predicates are *topmost* and $onTable$. Edges on the right of the block nodes represent the *on* relation; edges on the left represent its transitive closure. The structure to be focused represents all towers of height at least 3. The focus operation produces four possible structures. Integrity constraints used for the focus operation are shown in Table 3. The coerce operation refines the upper two structures using constraints (1) to make the summary element on which $obj_1$ lies unique. Constraint (3) is used to remove the third focused structure; constraint (4) eliminates the fourth focused structure.

In general, infinitely many focused structures can be generated as a result of the focus operation. For instance, focusing on the formula $\exists u(on(u, x))$ on the initial structure in Fig. 19 attempts to draw out all elements from the middle summary element which are beneath another. Since the abstract structure represents towers of all heights greater than 3, definite truth values for every instantiation of the focus formula can be obtained only by listing out structures for every possible height.

The focus formulas used in our methods are constrainted to be uniquely satsifiable and do not suffer from this problem.

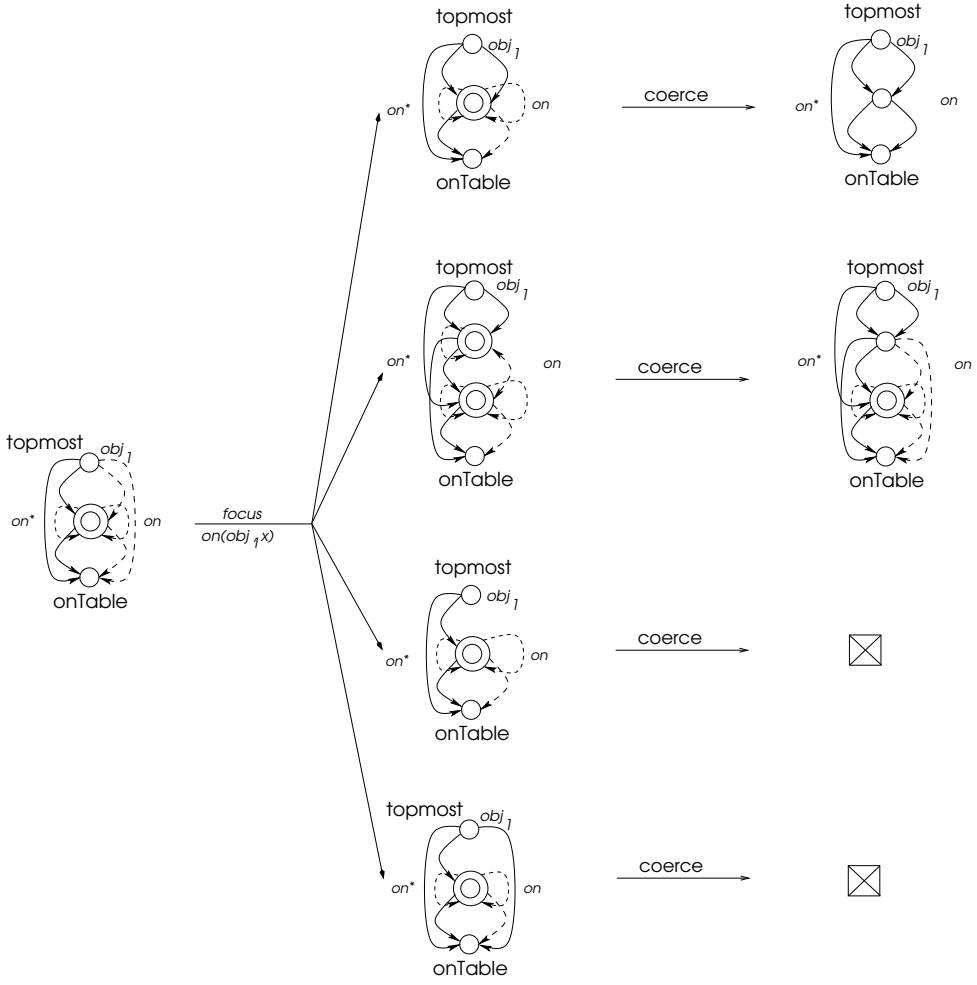


Figure 19: Focus operation on a structure in the blocks world with the formula $on(obj_1, x)$.