9-2010

Foundations And Applications Of Generalized Planning

Siddharth Srivastava
University of Massachusetts - Amherst

FOUNDATIONS AND APPLICATIONS OF GENERALIZED PLANNING

A Dissertation Presented

by

SIDDHARTH SRIVASTAVA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2010

Computer Science

FOUNDATIONS AND APPLICATIONS OF GENERALIZED PLANNING

A Dissertation Presented

by

SIDDHARTH SRIVASTAVA

Approved as to style and content by:

Neil Immerman, Co-chair

Shlomo Zilberstein, Co-chair

George Avrunin, Member

J. Eliot B. Moss, Member

Hector Geffner, Member

Andrew G. Barto, Department Chair
Computer Science

To my parents, without whose encouragement this may never have started.

ACKNOWLEDGMENTS

This thesis would not have been possible without the support and guidance of Prof. Neil Immerman, who encouraged me to explore many, disparate directions of research during my initial years as a graduate student. Much of this thesis is built around the interplay between the fields I explored during this period. My interaction with Neil taught me the value of clear formalizations, without which many of the ideas presented in this thesis would have been incomplete.

I am also deeply grateful to Prof. Shlomo Zilberstein for his enthusiasm and support for this direction of work long before he agreed to help in directing it. Shlomo's patience, encouragement and insightful advice were key in taking this entire direction of research from its inception to the results. I am also thankful to him for initiating me to the process of obtaining research funding. Shlomo's foresight and diligence in all aspects of his research and advising sets a model I hope to be able to follow in the future.

I am also thankful to my wife Sanchayeeta, who has been a constant source of optimism during my graduate studies. Without her perspective and support this work would not have reached fruition. My parents, my sister and my grandparents played the role of effortless motivators through their own strength and focus on learning.

I would also like to thank the countless reviewers who contributed by holding me to higher standards and my friends at UMass whose company allowed me to start every day with rejuvenated enthusiasm. My friend and former-colleague Pradeep's unstoppable excitement about research, education, and Isaac Asimov played no small part in getting me out of the corporate world and into research on AI.

I am also grateful to the National Science Foundation, whose support for generalized planning under grant IIS-0915071 played perhaps the most important role in enabling the development of ideas presented in this thesis.

Finally, I would like to thank my committee members, Prof. Eliot Moss, Prof. George Avrunin and Prof. Hector Geffner for helpful and insightful comments on all components of the thesis.

ABSTRACT

FOUNDATIONS AND APPLICATIONS OF GENERALIZED PLANNING

SEPTEMBER 2010

SIDDHARTH SRIVASTAVA

INTEGRATED M.Sc., INDIAN INSTITUTE OF TECHNOLOGY KANPUR

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Neil Immerman and Professor Shlomo Zilberstein

Research in the field of Automated Planning is largely focused on the problem of constructing plans or sequences of actions for going from a specific initial state to a goal state. The complexity of this task makes it desirable to find “generalized” plans which can solve multiple problem instances from a class of similar problems. Most approaches for constructing such plans work under two common constraints: (a) problem instances typically do not vary in terms of the number of objects, unless theorem proving is used as a mechanism for applying actions, and, (b) generalized plan representations avoid incorporating loops of actions because of the absence of methods for efficiently evaluating their effects and their utility. Approaches proposed recently address some aspects of these limitations, but these issues are representative of deeper problems in knowledge representation and model checking, and are crucial to the problem of generalized planning. Moreover, the *generalized planning problem* itself has never been defined in a manner which could unify the wide range of representations and approaches developed for it.

This thesis is a study of the fundamental problems behind these issues. We begin with a comprehensive formulation of the generalized planning problem and an identification of the most significant challenges involved in solving it. We use an abstract representation from recent work in model checking to efficiently represent situations with unknown quantities of objects and compute the possible effects of actions on such situations. We study the problem of evaluating loops of actions for termination and utility by grounding it in a powerful model of computation called abacus programs. Although evaluating loops of actions in this manner is undecidable in general, we obtain a suite of algorithms for doing so in a restricted class of abacus programs, and consequently, in the class of plans which can be translated to such abacus programs.

In the final sections of this thesis, these components are utilized for developing methods for solving the generalized planning problem by generalizing sample plans and merging them together; by using classical planners to automate this process and thereby solve a given problem from scratch; and also by conducting a direct search in the space of abstract states.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Automated planning is among the most fundamental problems in the field of Artificial Intelligence. The study of automated planning originated with notions of general problem solving; in general, it can be described as the activity of computing a sequence of actions which will lead to a desired situation. Since planning is a hard problem (PSPACE-complete even if the problem is specified using propositional calculus with grounded actions (Bylander, 1994)), various approaches have been proposed to construct "generalized" plans which can be reused and applied on multiple problem instances.

Consider a simplified firefighting problem as an example: an agent needs to determine if any room in a building with multiple, single-hallway floors is on fire. It is known that if a room is on fire, smoke can be detected from anywhere in that floor, and its heat can be detected just outside the door. The agent has a smoke detector and a heat detector. The general solution is straightforward: the agent must first use its smoke sensor to search the floors for fire, and if smoke is detected, use its heat sensors to find the room on fire. However, as stated, this problem is beyond even the representational capabilities of state-of-the-art planners: existing planners would require a specification of the *exact* number of floors and number of rooms per floor; in addition, most formalizations don't allow expression of constraints such as "a room on fire implies smoke in its hallway".

As a result of these limitations, existing approaches must handle each instance of the firefighting problem with different numbers of floors and rooms independently.

Further, even if correct plans have been produced for such searches under test conditions for a few small buildings using reactive control or state-of-the-art planners it is not possible to construct reliable plans for larger buildings. For the fire fighting agent, reliability is an important factor: generalizations are bound to be incomplete and planning time is limited–the agent should be able to determine possible gaps in its generalization efficiently, and request assistance if it is at a building for which its plan is likely to fail.

1.1 State of the Art

Planning in its current form originates from research on developing a robot which could interact with its environment to solve problems like rearranging objects (Fikes and Nilsson, 1971). Although the research problems involved with robot manipulation have since been largely separated from planning, modern planning domain specification languages are still based on the STRIPS representation developed for this project (Ghallab et al., 1998; Fox and Long, 2003; Gerevini and Long, 2005).

The problem of developing “generalized" plans which would apply to classes of similar problem instances was identified almost immediately after the development of the STRIPS framework (Fikes, Hart, and Nilsson, 1972). The objective was to construct a structure for efficient retrieval and storage of useful plan segments. Many approaches have since been proposed for the construction of such structures, both automatically and by utilizing hand-coded domain-specific control knowledge (Baier, Fritz, and McIlraith, 2007).

Early approaches to this problem such as triangle tables (Fikes, Hart, and Nilsson, 1972) and case based planning (CBP) (Hammond, 1986; Spalzzi, 2001) constructed plan structures that would be widely applicable, but incurred significant computational costs for appropriately storing plan segments and subsequently for selecting and modifying the usable ones when presented with a new problem. In explanation based learning (EBL) (Dejong and Mooney, 1986), a proof or an explanation of a solution was generalized to be applicable to different problem instances. However, EBL requires a hand coded domain-theory 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 predicates which captured recursive concepts, BAGGER2 could identify their application in proofs of plan instances, and generalize these proofs to produce plans with recursive or looping structures. However, this approach did not address the problem of the correctness of its solutions, which could result in non-terminating computations.

Perhaps the most studied aspect of this problem is captured by the framework of *contingent planning* (Peot and Smith, 1992; Bonet and Geffner, 2000; Hoffmann and Brafman, 2005; Bryce, Kambhampati, and Smith, 2006), where the agent does not have precise information about its state, and therefore *needs* to construct a plan for handling multiple scenarios. In essence, this is the fundamental problem in generalized planning, but contingent planners work in the special case where complete information about a particular problem to be solved is learned gradually during plan execution itself, through sensing actions. Contingent planners also suffer from one of the most common and significant limitations of approaches for generalized planning: their solution plans are tree-structured, with distinct plan branches corresponding to every object-property which may vary across problem instances. With this restriction, solution *representations* become exponential in the number of variable object-properties, making the problem solving process inherently intractable. With a few exceptions (e.g., conditonal nonlinear planning (Peot and Smith, 1992)), contingent planners also do not attempt to merge different solution branches, which could help alleviate the plan representation problem.

The inclusion of loops in plans is necessary for constructing powerful generalizations, and recent approaches have considered richer plan representations that allow cyclic flow of control. In strong cyclic planning (Cimatti et al., 2003), the objective is to produce plans with loops in domains where actions may have to be repeated due to the possibility of failed outcomes, or where temporal goals require repetition of some action sequences. However, in this framework linear plans are always preferred: cyclic plans are produced only if no acyclic plan can solve the given problem. Loops in these plans are therefore not used as structures for generalization, and they are not analyzed for making progress towards a goal.

KPLANNER (Levesque, 2005) produces plans with loops with the particular objective of increasing the range of problems that they solve. It proceeds by iteratively finding plans for problem instances of increasing values of a unique *planning parameter* and attempts to find loops by extracting recurring patterns. This system provides only a limited form of preconditions for the computed plans: they are guaranteed to work in a user-supplied interval of values of the planning parameter.

Another recent approach, DISTILL (Winner and Veloso, 2003, 2007), attempts to learn domain specific planners (dsPlanners) through examples. Such dsPlanners can be used to generate plans for different problem instances in a domain. 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, which consists of branches and simple loops. This approach however is limited to plan learning, and does not address the problem of correctness or applicability of its generalizations.

These approaches show a growing trend towards the development of plans which utilize loops for compactness and greater applicability in terms of problem sizes. However, there is also a simultaneous trend towards the weakening of fundamental notions such as plan correctness, or even more generally, knowledge of the potential effects of a computed plan. Although the inclusion of loops in plans makes it harder (and even impossible in the most general case) to evaluate these properties, plans without any information about correctness or expected results on application have very low utility.

The combined factors of computing a widely applicable, efficient generalized plan with a compact representation and predictable effects seem to produce an unsolvable problem. While this is true in the most general case, actual limitations on the solvability of this problem are not known.

1.2 Objectives of This Thesis

The focus of this thesis is on the following longstanding, albeit informally defined generalized planning problem:

Given a "class" of problem instances of interest, construct a "generalized plan" for "efficiently" solving them.

All the approaches discussed above have this objective. However, this problem, and consequently, the nature of its solutions, have never formulated or studied comprehensively. This makes it difficult to analyze different approaches, identify what they achieve, and to build on those components. Our first objective is to *develop a well-defined notion of generalized plans and identify the most significant problems in computing them.*

As discussed above, efficiently representing and working with unknown and unbounded quantities of objects is a key component of generalized planning problems. However, representing unknown quantities of objects has been recognized as a challenge not only for planning, but for all of AI. First-order representations solve a part of this problem by using predicates and quantifiers to express facts about world states in a compact manner. However, first-order theorem proving turns out to be very inefficient as a mechanism for planning. We therefore need to *develop a representation which utilizes the expressiveness of first-order logic, while allowing a heuristic, directed-search based approach for planning.*

Although the benefits of including loops in plans have long been evident, planning with loops remains a notoriously intractable problem. Remarking on this subject, Levesque (2005) notes that "even short iterative programs can be quite difficult to reason about". He concludes that "faced with an intractable reasoning problem, we can look for compromises. ... [and] forego the strong guarantees of correctness". KPLANNER is not alone in making these compromises: most approaches for finding generalized plans work with very low objectives regarding plan correctness. The underlying problem faced by all approaches in this direction is that plans with loops and even very primitive actions can simulate Turing machines (see, for instance Fact 1 on page 48), and thus have an undecidable halting problem. This result makes it impossible, in general, to perform fundamental tasks such as evaluating the net effect of a loop of actions included in a solution plan. The inclusion of unpredictable segments of operations which could traverse the entire state space can undermine the validity of any approach. However, despite the significance of this problem, there are no approaches for addressing it. On the other hand, the model checking community has produced numerous advances in approximately analyzing and even proving termination guarantees of limited classes of programs (Henzinger et al., 2002; Cook, Podelski, and Rybalchenko, 2006; Podelski and Rybalchenko, 2004). A direct application of these approaches to planning is difficult because action specifications in planning are typically richer and more expressive than program statements. However, for developing a sound approach for finding generalized plans we require at the least, *a study of the extent to which reliable generalized plans can be found and development of efficient methods for computing the effects of loops of actions in these classes*.

One of the most popular approaches for finding generalized plans is to generalize existing, working sample plans. In fact, to our knowledge all the approaches which attempt to find plans for handling multiple problem instances[1], beginning with triangle tables, BAGGER2, CBP, EBP, KPLANNER[2] and DISTILL are based on this idea. Of these, only KPLANNER, BAGGER2, and DISTILL attempt to create generalizations with looping structure; only KPLANNER explicitly addresses the problem of correctness. Our objective is to *create plan generalizations with loops and well defined notions of correctness*.

Creating generalizations which include loops has created a new problem in planning: since actions in a loop apply on different states in every iteration, it is not easy to determine *when*, and *if* a newly obtained plan segment will be applicable to one of those intermediate states. At the same time, a single example plan is often insufficient to explore all the different possibilities and multiple plans will generally be required to solve a given class of problem instances. The problem of merging multiple example plans to create a generalized plan with loops has not been addressed so far by any known approach. We therefore aim to *develop an approach for constructing a generalized plan with loops by extracting useful segments from multiple sample plans*.

Invariably, generalized plans constructed from sample solutions undergo an intermediate phase where they solve some, but not all the possible problem instances of interest. Although the problem of determining when these plans could fail is in itself a challenge, another aspect of generalized planning is to be able to extend partial, incomplete plans towards covering more problem instances. In fact, a solution to this problem will also allow the development of hybrid approaches like KPLANNER in a more general setting. KPLANNER's unique planning parameter allows it to generate new example plans aimed at solving new instances from the desired class. In order to do this more generally, we need to *identify potentially unsolved problem instances and extend the intermediate generalized plan using the generalizations of their solutions. This should*

[1]The objectives of strong cyclic planning differ from this.

[2]KPLANNER employs a directed hybrid search by incrementally finding solutions and generalizing them

allow the development of a hybrid generalized planning system by starting the process on an empty generalized plan.

1.3 Overview and Contributions of This Thesis

Formalizing the Generalized Planning Problem and Evaluation Criteria This thesis begins with a simple formulation of generalized plans (Section 2.1) which captures diverse efforts in this direction, ranging from triangle tables to KPLANNER. This formulation captures many interesting aspects of generalized planning, including the fact that generalized plans can also incur computational costs in producing real solutions: triangle tables required a search for the appropriate macro operations, CBP required extensive computation for plan retrieval and adaptation, KPLANNER and almost any approach with loops requires an instantiation into a linear sequence of actions. Naturally, each approach incurs a different cost, with loop unrollings being computationally inexpensive compared to the search required while using triangle tables. On the other hand, KPLANNER solutions are for specific classes of problems varying over a unique planning parameter, while triangle tables can solve any solvable problem in the domain. In fact, this formulation of generalized plans leads us to conclude that the quality of a generalized plan depends on various factors, which also define the key challenges in computing generalized plans. We present and study these factors in Chapter 2.

Representation for Planning with Unbounded Quantities Our representations for states and actions are motivated by first-order approaches for planning, such as situation calculus (Levesque, Pirri, and Reiter, 1998), as well as the TVLA system for model checking (Sagiv, Reps, and Wilhelm, 2002). TVLA uses 3-valued logical structures for compactly expressing the set of program states possible at each step in a given program. These 3-valued, abstract structures can capture unbounded sets of states with unbounded universes. We use this mechanism to compactly represent the class of problem instances to be solved, and use a specific form of TVLA's action update mechanism to apply actions directly on abstract states while minimizing loss in precision.

State abstraction is a crucial component in our approach to generalized planning: we use abstract states to recognize when certain properties which once held recur, and thus to identify situations where a previously applied sequence of actions can be repeated — or in other words, when this sequence of actions can be placed in a loop. We also use abstract states in the manner of TVLA, to represent the states possible at any point in a generalized plan's execution. In addition, the capacity of abstract states to represent sets of concrete states allows us to use them as *belief* states for planning in the presence of partial observability. The mechanism for state abstraction, as well as our representations for states, actions, and generalized plans are presented in Chapter 3.

Complexity Characterization and Efficient Algorithms for Computing Preconditions and Effects of Loops of Actions We begin our study of loops of actions with a fundamental paradigm of computation called *abacus programs*. Abacus programs have the same computational power as Turing machines, use only increment and decrement operations, and have a convenient representation as graphs. Further, plans with loops in many domains can be directly translated into abacus programs. Analyzing abacus programs with loops of simple actions thus gives us valuable leverage in understanding the effects with loops of richer, planning domain actions. Chapter 4 presents a number of results on the conditions on graph structure under which loops in abacus programs can be analyzed efficiently, along with algorithms for doing so.

In Chapter 5, we describe the connection between plans and abacus programs and present algorithms for translating plans into abacus programs without changing their graph structure. In the remainder of the thesis, these approaches are used extensively both during, and after the generation of generalized plans to determine that (a) any loop being created makes measurable progress and will terminate, (b) *when*, or after how many steps each loop will terminate, and (c) the set of problem instances that a generalized plan with loops will solve.

Approach for Plan Generalization Our approach for generalizing a sample plan is to apply the plan on an abstract state representing a super-set of the problem instance that it is known to solve (we call this process “tracing" the plan in the abstract state space). This provides a sequence of abstract states resulting after each action in the plan; repeated states in this sequence indicate that a combination of properties recurred, and thus the intermediate sequence of actions may be placed in a loop. However, the resulting loop may be a *static* loop, amounting to a null effect. We use the techniques developed in preceding chapters to avoid such loops and only create those which make measurable progress, and are guaranteed to terminate. This is presented in the first part of Chapter 6.

Approach for Extending Plans with Loops In order to be able to utilize additional sample plans to cover potentially unsolved problem instances, we store the set of possible problem instances at each step of the generalized plan. When a new plan is presented, we trace the plan and use its intermediate abstract states to determine when the subsequent plan segment will be applicable. If an abstract state has already been solved, the immediately subsequent steps need not be added to the generalized plan; on the other hand, if one of its abstract states captures a previously unexplored part of the state space, a new branch can be created for handling it using the given plan's actions. The second part of Chapter 6 describes this process.

Approaches for Plan Synthesis In Chapter 7, we present two approaches for synthesis of generalized plans from scratch. The first approach conducts a search in the abstract states space for paths *with progressive loops* leading to a goal state. Although this is the first approach for finding generalized plans by search, a practical implementation of this approach requires extensive development of heuristics for guiding the search process. An alternative to this approach is to use the fairly advanced directed-search capabilities of modern *classical* planners for generating generalized plans. The second part of Chapter 7 describes an approach for identifying potentially unsolved abstract states from partial generalized plans, creating a specific, concrete instance of these states, and invoking a classical planner on these instances to obtain new concrete plans which can be merged with the generalized plan along the lines of the techniques developed in Chapter 6.

CHAPTER 2

THE GENERALIZED PLANNING PROBLEM

Informally, the problem of generalized planning is to find plans that can solve a set of problem instances. In one form or another, this problem has been studied since the earliest work on STRIPS planning (Fikes, Hart, and Nilsson, 1972) and the fundamental motivations behind it stem from classical planning itself. Consider the simple planning problem of unstacking a tower of blocks. Given a problem instance with 3 blocks, with block b_3 on block b_2, and b_2 on b_1, the solution plan would be: $\mathit{moveToTable}(b_3), \mathit{moveToTable}(b_2)$. The problem of classical planning is to find such solution plans for specific problem instances like the three block tower described above. As we increase the number of blocks in this problem, the complexity of solving it grows exponentially, although the solutions address common subproblems and are remarkably similar to each other. Approaches for finding generalized plans attempt to extract, and subsequently utilize such common solutions and problem structures.

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

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

While Unstack contains the basic idea of how to solve any unstacking problem, it cannot be used directly to solve a particular instance of the problem. For example, let I be an

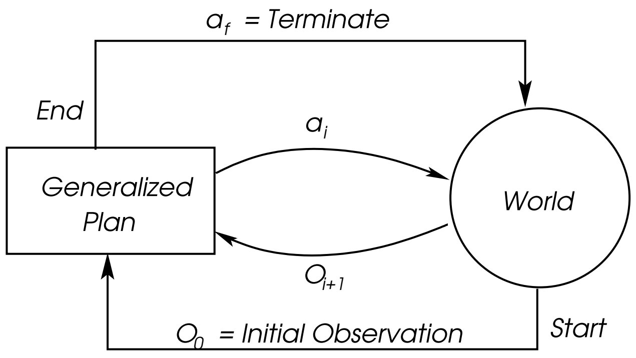


Figure 2.1: Execution model for generalized plans

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

Fig. 2.1 extends this approach to a generic model for executing a generalized plan. In this figure, the "world" represents the system on which the plan has to be applied, and a *problem instance* is a completely specified state of this system. Throughout this thesis, we will use the following assumptions about the world on which plans will be applied:

1. Any uncertainty in action outcomes stems from lack of information (or *partial observability*) about the current state of the world.

2. This uncertainty, if present, is absolute: the probabilities of possible real states are unknown.

3. All actions generate an observation. In settings with complete observability, this observation is the resulting state; in settings with partial observability the observation is the observable portion of the resulting state.

The first assumption implies that applying an action on a state of the world must result in a unique resultant state. However, in many situations the current state of the world may not be known precisely. Actions applied on such partially known states may result in one of a set of possible observational outcomes, depending on the true state that the world was in.

Plan execution starts with the initial observation O_0 which consists of all the known information about the initial state of the world. Subsequently, the execution of a generalized plan amounts to applying a sequence of actions on the given world model. Plan execution terminates with a special termination action (a_f) which does not generate any observation. We consider the computational process of generating each successive action as the generalized plan's *method for instantiation*. This method may utilize the entire history of observations while generating the next action. Consequently, what we require from generalized plans is a special form of a *policy* as used in Markov decision processes.

More specifically, the process of instantiation can be understood to be evaluating a *policy with termination actions* which maps sequences of observations to actions. Let O be the set of observations possible in a domain, and A the set of domain actions. Formally, a policy P with termination actions is of the form $P : O^* \rightarrow A \cup \{a_f\}$ with the restriction that for any $\bar{O}_1 \in O^*$, if we have $P(\bar{O}_1) = a_f$, then $P(\bar{O}_1\bar{O}_2) = a_f$ for all $\bar{O}_2 \in O^*$.

In deterministic situations, the world model can be simulated. Consequently, in such settings generalized plans can be instantiated completely for any initial state by simulating plan execution.

2.1 Architecture of Generalized Plans

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

In general, the *data-structure* component of a generalized plan can be used to store specific algorithms for the class of problem instances of interest (such as a formal representation of the algorithmic plan string of the Unstack plan shown above), or more general domain-control-knowledge (Baier et al., 2009). A generalized plan need not provide the guarantee that all its instantiations will be finite. Plan execution or even a complete offline instantiation of the plan may therefore never terminate. On the other hand, the fact that a plan's instantiation method terminates need not imply that it will always achieve the goal. Proving that a generalized plan is "correct" in the sense of reaching a goal state starting from a given problem instance therefore subsumes proofs of termination as well as goal-reachability.

This architecture of generalized plans unifies various approaches to "efficiently" producing "good" plans for classes of problems. Approaches for macro tabulation such as Triangle Tables (Fikes, Hart, and Nilsson, 1972), or plan compilation such as case-based planning (CBP (Spalzzi, 2001)) can also be understood as developing the generalized plan's knowledge data-structure in order to utilize instantiation methods more efficient than classical planners. More recent approaches like KPLANNER (Levesque, 2005) and loopDISTILL (Winner and Veloso, 2007) aim to extend the applicability of generalized plans to unbounded classes of problems by including loops of actions in the generalized plan's knowledge data-structure.

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

2.2 Evaluation Criteria for Generalized Plans

Approaches for classical planning typically have a single objective: to generate the shortest possible plan for solving a given problem instance. Generalized plans on the other hand need to address several different axes of utility, of which the need for solving multiple problem instances is only the most obvious. As discussed above, this requirement is in fact satisfied even by classical planners — which are clearly not ideal generalized plans: generalized plans need to have efficient instantiation processes. On the other hand, if we only wanted a low cost of instantiation, a classical *plan* could be treated as a generalized plan — classical plans are fully instantiated and will incur no further computational cost for instantiation. However, a classical plan suffers from the limitation of being able to solve only a single problem instance of interest.

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

1. Complexity of checking applicability
2. Complexity of plan instantiation
3. Domain coverage
4. Complexity of computing the generalized plan
5. Quality of the instantiated plan

Complexity of Checking Applicability An applicability test for a generalized plan is a procedure which takes as its input a problem instance and returns True or False as its output, reflecting whether or not the generalized plan can solve the given problem instance. The complexity of checking applicability is the computational complexity of this procedure. A generalized plan can be designed to proceed in one of two ways when given an input problem instance: (1) conduct a pre-designed applicability test to determine if an instantiation will be possible, and if so, proceed to find it, or, (2) directly attempt an instantiation. The problem with the second approach is that instantiation can be an expensive and wasteful operation if the generalized plan cannot actually solve the given problem instance. As mentioned above, instantiations may even enter potentially non-terminating computation, undermining the utility of a generalized plan. On the other hand, a generalized plan could provide an efficient applicability test which can determine applicability in time linear in the size of the given problem instance. While developing such tests is impossible in the most general case due to the undecidability of the halting problem for general programs, for many generalized plans in the existing planning domains it is actually possible. Although the first approach is desirable, it is often very difficult to construct an applicability test; the ideal situation would be to have a linear-time or better applicability test.

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

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

Quality of the Instantiation The quality of instantiation of a generalized plan determines its usability on a problem relative to any available alternative solutions. Ideally, the sequence of actions produced by a generalized plan for a given problem should be optimal according to a measure like the number of actions or their cost. Determining how well or poorly a generalized plan's instantiations perform is not always easy to determine. In this thesis, we will assume that alternative solutions do not exist, and consequently, any instantiation which can solve a given problem instance will be desirable.

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

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

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

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

Complexity of Computing A Generalized Plan The complexity of constructing a generalized plan depends on the computational complexity of representing its knowledge data-structure. A contingent plan (Peot and Smith, 1992; Bonet and Geffner, 2000) can be used as the knowledge data-structure of a generalized plan. Such a generalized plan would have a clear applicability test (by definition, it would solve all instances of the initial belief state used while computing the contingent plan) and a low cost of instantiation. However tree-structured representations used for expressing contingent plans can grow exponentially with every unknown predicate tuple, making such plans inherently more difficult to find. Plan representation thus becomes an important factor when considering the complexity of deriving a generalized plan itself. Approaches like DISTILL, KPLANNER, and BAGGER2 (Shavlik, 1990) mitigate this cost by constructing plans with loops that can instantiate into larger concrete plans. While adding loops can significantly reduce the size of the data-structure used in a generalized plan and increase its domain coverage, it can in general have adverse effects on plan applicability tests and make such plans unreliable. This is because plans with loops and branches approach the expressive power of programs – determining when they will work, or even *terminate* is thus undecidable in general for such plans.

The five factors discussed above determine the quality of a generalized plan. While various approaches have addressed different subsets of these factors, there were none that address all of them. In the next section, we present an analysis of some well-established approaches along these five factors.

2.3 Analysis of Existing Approaches

2.3.1 Approaches Focusing on Domain Coverage

These approaches focus on increasing the domain coverage. Consequently, the generalized plans they produce are intended to work for very broad classes of problem instances. However, they incur high costs of instantiation which may even exceed the cost of planning directly, without the additional knowledge components.

2.3.1.1 Triangle Tables

Almost as soon as the problem of planning took its modern form in the STRIPS representation, it became clear that reusing plans or operator sequences would reduce significant amounts of repeated effort while solving similar problem instances. Triangle tables (Fikes, Hart, and Nilsson, 1972) were designed as lookup tables for operator sequences or *macro operators*, together with their overall effects and preconditions. These tables were used to monitor the execution of plans as well as provide short-cuts to the re-planning process in case of an unexpected result.

This compilation of plan segments forms one of the earliest approaches for generalized planning: the motivation was to obtain a large domain coverage with a cost of instantiation lower than that of planning from scratch. In the worst case however, inclusion of macro operators increases the search space of possible operators. Therefore, although the coverage of triangle tables would definitely be greater than that of the individual plans, they do not guarantee a lower cost of instantiation.

2.3.1.2 Case Based Planning

Triangle tables were difficult to maintain while avoiding multiple, subsuming versions of operator sequences. An alternate approach, case based planning (CBP), was developed as a model for "planning from experience" (Hammond, 1986; Spalzzi, 2001). Like triangle tables, the objective of CBP was to reduce the cost of instantiation while obtaining a wide domain coverage. CBP approaches directly addressed the problem of redundant operator sequences by developing specialized representations for storing plans in a database. This facilitated faster retrieval of plans relevant to the given problem. Approaches like CHEF (Hammond, 1986) used a model of "the causality of the domain" to be able to repair these retrieved plans to be applicable to the given problem.

In CBP approaches, subroutines for plan retrieval, adaptation, and efficient storage have to be executed on every problem instance. This increases the cost of instantiation; CBP approaches can also suffer from a lack of an efficient test of applicability: while triangle tables only increased the set of available operators and thus had the same applicability as the underlying action model, the applicability of a CBP database depends on the scope of its plan-memory and adaptation routines (although a classical planner invocation could always be incorporated with a CBP system to make it universally applicable).

2.3.2 Approaches with Low Costs of Instantiation

Approaches discussed in this section incur fewer overheads during plan instantiation. This is typically accompanied with trade-offs in the domain coverage of the produced generalized plans and absence of clear applicability tests.

2.3.2.1 Explanation Based Planning

Explanation based planning borrows ideas from explanation based learning (EBL), where the proof of a given solution is generalized to solve other, conceptually similar problem instances. Methods for EBL rely on complete domain theories to be able to generate these proofs. The BAGGER2 system (Shavlik, 1990) could generate plans with recursive structures for handling problems with unbounded numbers of objects. By using recursive structures, BAGGER2 significantly reduced the number of possible actions or rules to be attempted during plan instantiation.

However, in order to do so, BAGGER2 used an a-priori hand-coded domain theory which included the recurrence relationships for the recursive concepts that could be encountered in a solution. Although the use of a domain theory allowed BAGGER2 to make well-justified generalizations, it could not address the problem of designing an applicability test: BAGGER2 could generate (but not identify) non-terminating plans.

2.3.2.2 Contingent Planning

The problem of contingent planning addresses a version of the generalized planning problem where the agent does not have precise information about its environment and therefore needs to be prepared to solve any of the possible instances it might encounter. The agent's actions in contingent planning problems may allow it to gain new information about its state. A contingent plan, therefore, may consist of such *sensing* actions; the remainder of the plan after these actions can depend upon the results of these actions. At any stage, the set of possible states that the agent may be in constitutes its *belief state*. The contingent planning problem can thus be formulated as a search problem in the space of the agent's belief states (Bonet and Geffner, 2000), with the goal belief states consisting only of real goal states. Contingent plans can be completely instantiated only during plan execution, because parts of the plan may depend on the results of sensing actions.

By definition, a contingent plan must solve every initial state represented by the initial belief state. The applicability test for contingent plans thus reduces to testing whether the current instance is captured by the initial belief state, which can typically be done in time linear in the size of the initial state. Contingent plans typically consist of fully instantiated operations, and can be easily instantiated into linear sequences of operations once the action observations become available. This makes contingent plans desirable from the point of view of applicability tests, domain coverage, and instantiation costs.

However, contingent plans are typically represented using tree structured representations (Hoffmann and Brafman, 2005) that grow exponentially with the number of observations to be performed. This makes finding contingent plans computationally intractable. Contingent planning thus addresses most of the requirements of generalized planning – with the exception of the complexity of computing a generalized plan. Effectively, this restricts the applicability of contingent planners to problems that have small solutions.

2.3.2.3 Strong Cyclic Planning

In strong cyclic planning (Cimatti et al., 2003), plans are represented as execution control tables mapping problem states to actions. These tables may incorporate cyclic flows of control if from every state in the cycle, there exists a path of actions leading to a goal state. With this structure, under the assumption that during execution every possible action outcome in the plan has a non-zero probability, the plan will eventually terminate in the goal state.

Strong cyclic planning allows us to represent compact contingent plans and is particularly applicable in problems with temporally extended goals, or situations where actions may fail and the only way to reach the goal is to repeat them until they succeed (for instance, when the goal is to stack a block on another, but the *move* action may drop a block back on to the table). A strong cyclic planner produces a plan with loops only if no acyclic plan can solve the given problem.

Although strong cyclic planning alleviates the representational constraints introduced by contingent planning in some situations, loops in strong cyclic plans are not utilized for generalization. Further, applicability tests for strong cyclic plans are weaker: strong cyclic plans are guaranteed to terminate, but they may perform an unbounded number of operations before doing so. This is unavoidable in some problems (such as when there is a possibility of the move action dropping a block) but strengthening these guarantees for other classes of problems would require further analysis of the effects of loops included in a strong cyclic plan.

2.4 Discussion

The development of approaches for generalized planning started with compilation of plan segments for future reuse and adaptation. The focus of research then shifted to approaches with efficient instantiations, almost all of which used representations with loops of actions. However, all of the approaches developed so far address only subsets of the interdependent problems of developing applicability tests, achieving broad domain coverage with low costs of instantiation and representation.

Modern approaches such as KPLANNER, DISTILL and controller synthesis by Bonet et al. (Bonet, Palacios, and Geffner, 2009) address these problems with the significant exception of applicability tests. However, the problems involved in constructing an applicability test are the very problems involved in constructing a plan with loops: creating an arbitrary loop of actions in itself is neither challenging, nor even useful. The critical advantage in including a loop of actions is when this loop forms a compact representation of a more general, useful process of computation; and in order to determine *when* a loop being considered for addition by any approach would be useful, we need to determine *if*, and *when* it will have a useful effect. Approaches like KPLAN-

NER, DISTILL, and controller synthesis by Bonet et al. use empirical observation of loop effects in known or simulated executions to recognize a loop as well as to justify its future utility. In the following chapters, we will see that in many situations, we can do better, and efficiently. While our approach is similar to existing approaches in using executions to identify potential loops, we conduct analytical computations of potential loops' effects before adding them. Consequently, in these domains we obtain plans that have low costs of instantiation, broad domain coverage, use loops for compact representation and also provide efficient tests of applicability with a representation of the possible effects on plan execution.

CHAPTER 3

STATE, ACTION AND PLAN REPRESENTATION

We begin this chapter by describing the standard, logic-based framework that we use to describe planning. This is followed by the definition of our representation of generalized plans and their instantiations in Section 3.2. Section 3.3 describes the state abstraction mechanism we use for generalized planning and Section 3.4 describes how the action formulation is extended to deal with abstract states.

3.1 Actions and Concrete States

States are represented as logical structures. Actions are expressed using update formulas that define the new truth values of tuples for each predicate.

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

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

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

- $at(x,y)$: object x is at location y.

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

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

Each action a consists of a precondition $pre(a)$ and update formulas, $up(p,a)$, defining the new value of each predicate p after a has been applied. An action can be applied on a state only if its preconditions hold. For example, the following is the definition of the action *Move*:

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

We use the notation τ_a to denote the set of all the update formulas for an action, and $\tau_a(s)$ to denote the result of applying those formulas on a structure s. We will represent the update formula for the predicate p – such as the above update formula for the predicate at – in the following form:

$$p' \equiv [\neg p \ \land \ \Delta^+_{p,a}] \quad \lor \quad [p \ \land \ \neg\Delta^-_{p,a}] \tag{3.1}$$

Here $\Delta^+_{p,a}$ denotes the conditions under which predicate p is changed to true on action a, and $\Delta^-_{p,a}$ denotes the conditions under which it is changed to false. In our implementation, constants are represented as unary predicates that are constrained to be unique. They can thus be updated in a manner similar to predicates, using Eq. 3.1.

In addition to defining the vocabulary and actions of a planning problem, we typically include an *integrity constraint* that specifies the set of valid states. For example, the integrity constraint, $\mathcal{K}_d$ for our unit delivery is the universal closure of the conjunction of the following formulas:

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

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

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

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

Transitive closure is used in our framework to express connectivity properties such as the transitive closure of *on* ("above") in the blocks world (see the Striped Block Tower, Green Block and Hall-A problems, Section 6.1.3).

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

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

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

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

3.2 Generalized Plan Representation

Intuitively, a generalized plan is a full fledged algorithm. We represent the data structure component of generalized plans using a graph representation. Formally,

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

This gives us a convenient representation of generalized plans which is also similar to standard representations for finite state transducers. For ease in explanation of some of our algorithms for constructing generalized plans however, we will use the equivalent, dual notation where conditions label nodes and actions label edges. In the rest of this thesis, all references to generalized plans refer to graph-based generalized plans.

This representation of actions and plans is similar to situation calculus (Levesque, Pirri, and Reiter, 1998) and Golog programs (Levesque et al., 1997). However, a significant difference between our framework and Golog programs is that we automatically

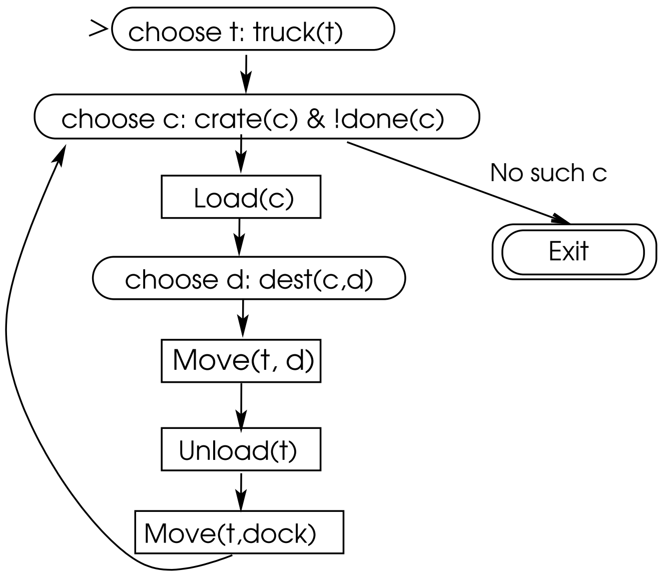


Figure 3.1: A generalized plan for delivery. The start node is labeled *choose t: truck(t)*.

generate edge labels (in the form of summarized, abstract structures) representing the set of concrete states that can provably be solved by the generalized plan starting with the subsequent node's action. Further, while Golog programs are typically hand-coded, albeit sometimes in a partially specified manner, our objective is to find generalized plans automatically together with the class of problem instances that they can solve.

Fig. 3.1 shows a generalized plan for the delivery problem. A generalized plan can include *choice* actions for choosing objects to be used as arguments for future actions. These actions select an object which satisfies a given formula in first order logic, and assign it to a constant used in action update formulas. Intuitively, if multiple objects satisfy the formula used for selection, we require that the generalized plan should work with any of those qualifying objects. Choice actions are discussed in detail in Section 3.4.3; they are constructed automatically in our approach for generalized planning (Section 6.1).

In general, compound node labels consisting of multiple actions and choice actions can be used for ease of expression. For simplicity, we allow only a single action per node and require that all of an action node's operands be instantiated (using choice actions) before executing that node.

3.2.1 Instantiation of Graph-Based Generalized Plans

A generalized plan's *control configuration* is given by a tuple $\langle pc, S, i\rangle$ where $pc \in V$ is the current control node, S, the problem state for which an action has to be produced; and i, an instantiation mapping the arguments of $\ell(pc)$ to elements of the state S. As mentioned above, the instantiation i is constructed using choice actions (Section 3.4.3). A control configuration determines the next action to be executed as the action $\ell(pc)$ with the arguments represented by i. Successive instantiated actions are produced by taking as input, the state resulting from an execution of the previous instantiated action, and following the edge in the generalized plan whose conditions are satisfied by this state, starting with the initial node s. After executing the action at a node $u \in V$, the next possible control nodes are those neighbors v of u for which the condition $\ell(\langle u, v\rangle)$, and the preconditions of action $\ell(v)$ are both satisfied by the current state S with the current instantiation i. We assume the existence of default edges leading to a terminal (trap) state labeled with a termination action, which are taken when suitable next nodes cannot be found in the generalized plan or when an action node is reached without an instantiation for all of its action's arguments.

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

In general, it is undecidable to determine the preconditions of a generalized plan because of the undecidability of the halting problem and the fact that a generalized plan can be used to represent an arbitrary program. However, in practice we finesse this problem by considering only finite domains. In particular, we call a generalized planning problem “finitary” if for every instance $i \in \mathscr{I}$, 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 capture most real-world situations and have a decidable halting problem. In particular, the language consisting of instances that a generalized plan solves in a finitary domain is decidable. This is because in these domains we can maintain a list of visited states (which has to be finite), and identify non-terminating behavior if a state is revisited. We formalize this notion with the following observation:

Observation 1 *(Decidability in finitary domains)* The halting problem and the set of problem instances solved by any generalized plan in a finitary domain is decidable.

3.3 State Abstraction Using 3-Valued Logic

In this section we describe the system of state abstraction we use for compactly representing unbounded sets of concrete structures. Action application on abstracted states will be discussed in detail in the next section.

The TVLA static analysis system (Sagiv, Reps, and Wilhelm, 2002) uses three-valued logic to represent sets of structures of unbounded size, using a single, bounded-size abstract structure. We adopt this representation, using abstract, three-valued structures to compactly express sets of planning problems (concrete structures) of unbounded size.

In a 3-valued structure, each tuple may be present in a relation with definite logical values 1 (present), 0 (not present), or indefinite value $\frac{1}{2}$ (perhaps present). The added flexibility provided by the indefinite value makes all the difference. We will use the notation $|S|$ to refer to the universe of the logical structure S.

Definition 4. (3-valued Structure) *A* 3-valued structure*, also called an* abstract structure*, S over vocabulary $\mathcal{V} = \langle p_1^{a_1}, \ldots, p_r^{a_r}; c_1, \ldots, c_t \rangle$ consists of a non-empty universe $|S|$, and for every predicate symbol $p_i^{a_i}$ and tuple $(u_1, \ldots, u_{a_i}) \in |S|^{a_i}$, a truth value*

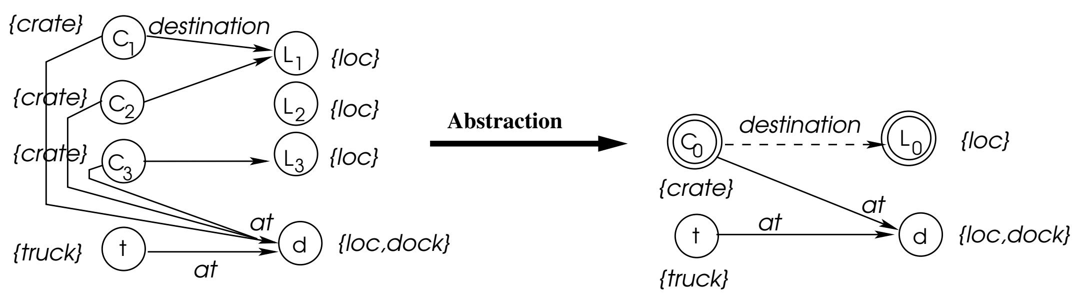


Figure 3.2: Abstraction in the delivery domain

$[[p(u_1,\ldots,u_k)]]_S \in \{0,1,\frac{1}{2}\}$, *and as usual, for every constant symbol* c_j *an element of the universe,* $[[c_j]]_S \in |S|$.

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

Example 1. *Fig. 3.2 shows a diagram of a concrete structure, C, representing a unit delivery problem consisting of three crates, one truck, and three locations. A three-valued structure, S, is shown on the right. The double circles represent summary locations. The solid arrows represent truth values of "1" and the dotted arrows represent truth values of "$\frac{1}{2}$".*

Intuitively the abstract structure S in Fig. 3.2 represents the concrete structure, C, as well as all other unit delivery problems that have exactly one truck, the truck is at the dock and empty, and there is at least one location not equal to the dock.

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

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

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

For domain schema $\mathscr{D} = \langle \mathscr{V}, \mathscr{A}, \mathscr{K} \rangle$, we use the notation,

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

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

In a domain schema, a subset of the unary predicates, A, is identified as the set of *abstraction predicates*. In all examples used in this thesis, the set of abstraction predicates is the set of all (observable) unary predicates. Furthermore, the constants are interpreted as non-summary elements.

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

$$\mathrm{role}(a) \quad = \quad \left\{ p_i \in A \;\middle|\; [[p_i(a)]]_S = 1 \right\} \cup \left\{ c_j \;\middle|\; [[c_j]]_S = a \right\}.$$

For example, in Fig. 3.2 elements c_1, c_2, c_3 have role $\{crate\}$, L_1, L_2, L_3 have role $\{loc\}$, t has role $\{truck\}$, and d has role $\{loc, dock\}$.

The abstraction predicates are used to keep certain properties definite as we abstract others. Each concrete structure C is represented by its *canonical abstraction*: the most precise abstract structure in which all elements of the same role are merged:

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

1. $f(u) = e_{\text{role}(u)}$
2. $[[r(e_1, \ldots, e_a)]]_S = \sup_{\preceq}\{[[r(u_1, \ldots, u_a)]]_S \mid f(u_i) = e_i, i = 1 \ldots, a\}$, *for all relation symbols r^a.*

Thus the truth value of $r(e_1, \ldots e_n)$ in S is the definite value 0 or 1, if C agrees on that value of $r(u_1, \ldots, u_n)$ for all elements of C of the appropriate roles. Otherwise, the value in S is $\frac{1}{2}$. For example, in Fig. 3.2, $S = \mathit{canon}(C)$. In general, suppose that C is a concrete structure and $S = \mathit{canon}(C)$. Then by the above definition, e_r is a summary element of S i.e., $[[e_r = e_r]]_S = \frac{1}{2}$, iff C has more than one element of role r. Furthermore, regardless of how large C is, $|S|$ has no more than 2^a elements where a is the total number of constant symbols and abstraction predicates. Increasing the number of abstraction predicates makes canonical abstractions more precise at the cost of increasing their size.

3.4 Action Application on Abstract states

We now present the methodology for applying action updates on abstract states. We begin by describing TVLA's focus and coerce operations, which make abstract structures more precise prior to action application; we describe how these operations are used in our system for generalized planning in Section 3.4.2, followed by a description of choice actions in Section 3.4.3. Finally, we present a brief discussion of how this framework relates to, and can be used for, modeling belief states and non-deterministic sensing actions of contingent planning.

Action application on an abstract structure succeeds only if its precondition evaluates to 1. Therefore, action updates will fail on abstract structures which represent states that don't satisfy the preconditions in additions to those that do. Further, when applied to an abstract structure with imprecise truth values, update formulas for actions might evaluate to $\frac{1}{2}$. Propagation of the $\frac{1}{2}$ truth value in this way can quickly result in very imprecise structures with no useful information. It is therefore desirable to improve the precision of abstract structures in terms of the key predicates that an action and its preconditions depend on. This is done in TVLA through the *focus* and *coerce* operations.

3.4.1 Focus and Coerce

The idea behind focus and coerce operations is to generate distinct structures for the different *definite* truth values possible for a given set of properties on a given abstract structure. The role of focus in this methodology is to perform a case analysis, producing a structure for every possible truth value of the atoms in the given properties; coerce completes the process by *refining* structures (by making imprecise truth values 0 or 1) as far as possible to satisfy the integrity constraints and rejecting any structure produced by focus which cannot be refined to be consistent with the integrity constraints (Sagiv, Reps, and Wilhelm, 2002). This mechanism is used to increase precision in abstract structures just prior to action application.

Given an abstract structure S and a formula ϕ on which we need precision, a focus operation is defined as one that produces a set of possibly abstract structures, $\mathit{Focus}(S,\phi) = \{S_1, S_2, \ldots S_k\}$, which capture exactly $\gamma(S)$, and in each of which ϕ evaluates to a definite truth value for any possible instantiation of its free variables. In general, the set $\mathit{Focus}(S,\phi)$ may be infinite. Consequently, there is no general algorithm for focus. TVLA's limited focus algorithm on a formula with one free variable on a structure which has only one role ($Role_i$) is illustrated in the top row of Fig. 3.3: if $\phi()$ evaluates to $\frac{1}{2}$ on a summary element, e, then this can be captured by three different abstract structures corresponding to cases where: either all of e satisfies ϕ, or part of it does and part of it doesn't, or none of it does. In general, additional elements created during this process inherit the truth values of other predicates from the original summary element. Note that ϕ evaluates to a definite truth value for all elements in each of these three structures. The focus algorithm on a binary predicate, at most one of whose arguments is a summary element, is identical. In fact, this algorithm works in any situation where at most one of a predicate's free variables is instantiated with a summary element (the focus formulas used in this thesis satisfy this requirement). Otherwise, this algorithm does not terminate. The focus operation wrt a set of formulas works by successive focusing wrt each formula in turn.

This process of splitting summary elements could produce structures that violate the integrity constraints. Such structures are later removed by TVLA's coerce operation. Using the integrity constraints, coerce either refines structures by making their relations' truth values precise, or discards them when a refinement reconciling a focused structure with the integrity constraints is not possible. Further descriptions of these operations can be found at (Sagiv, Reps, and Wilhelm, 2002).

3.4.2 Action Specific Focus Formulas

Recall that the predicate update formulas for an action operator take the form shown in Equation 3.1. For unary predicate updates, expressions for Δ_i^+ and Δ_i^- are *monadic* (i.e. have only one free variable, corresponding to the free variable on the LHS, apart from action arguments whose values will be constants when an action is applied). When applied on a structure with precise truth values for abstraction predicates, an update of the form of Eq. 3.1 can result in imprecise truth values for these predicates only if the formulas $\Delta^\pm$ evaluate to imprecise truth values. Consequently, in order to keep the abstraction predicates precise, we focus on $\Delta^\pm$ expressions prior to action application.

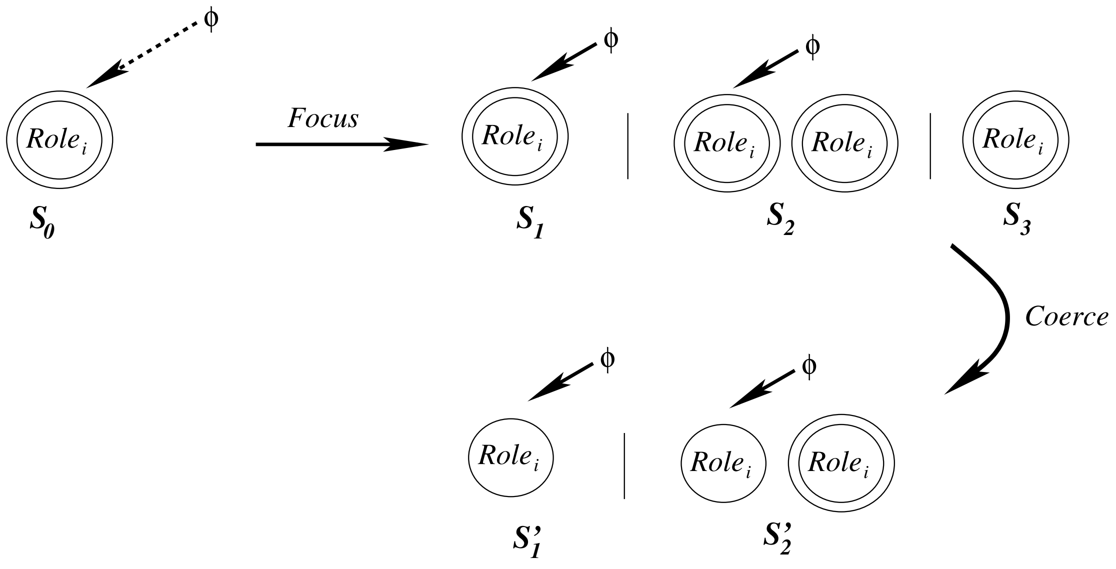


Figure 3.3: Effect of focus with respect to ϕ.

We choose the set of focus formulas to be used prior to an action update as exactly the predicate terms occurring in $\Delta^{\pm}$ formulas for the abstraction predicate updates and the predicate terms used in action preconditions. The fact that these formulas are monadic ensures that the focus algorithm with these formulas terminates.

We use F_a to denote this set of focus formulas for an action a. We illustrate this choice of focus formulas using the following example from the blocks world, since non-choice actions in the unit delivery problem do not need focus formulas.

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

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

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,* $on(\mathrm{obj}_1, x) \wedge (x \neq \mathrm{obj}_2)$ *is included in the focus formula for* Move()*. Effectively, this formula is* $on(\mathrm{obj}_1, x)$ *because* $x \neq \mathrm{obj}_2$ *will evaluate to a definite truth value for every instantiation of* x*. This is because the constant* obj_2 *will be assigned to a singleton element by a choice action.*

3.4.3 Isolating Action Arguments

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

Consider Fig. 3.3. If integrity constraints restricted ϕ to be unique and satisfiable, then structure S_3 in Fig. 3.3 would be discarded by coerce. Further, the summary elements for which $\phi()$ holds in S_1 and S_2 would be replaced by singletons (the lower row in Fig. 3.3). The two structures S_1' and S_2' 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. In other words, this combination of focus and coerce yields the two possible situations when an individual element is selected from a summary element: that individual may, or may not, be the last remaining individual represented by the summary element. Elements chosen from their summary elements in this manner can be used as action arguments.

Choice actions of the form "choose c: $\xi(c)$" can therefore be implemented by applying the following steps on a given structure. ("*chosen*" is a new predicate, with the integrity constraint of uniqueness)

1. Set the *chosen* predicate: $chosen(x) := \xi(x) \wedge \frac{1}{2}$
2. Focus w.r.t $chosen(x)$: This creates multiple structures with different possible choices of elements satisfying ξ.
3. Set the argument: for every resulting structure, set constant c to the element satisfying *chosen*.

Example 3. *Consider the sequence of operations in Fig. 3.4 in a simplified version of the delivery domain (we ignore the trucks and current positions of crates).* $chosen(x)$ *is initialized to* $\frac{1}{2}$ *for all objects with the role* $crate$ *in this figure. The first focus operation illustrates the drawing out of an action argument from its summary element, in this case, of role* {crate}. *A constant c is set to the drawn out crate. The second focus operation focuses on* destination(c, x), *effectively creating possible cases for the destination of crate c. Integrity constraints are used to assert that (a)* $chosen(x)$ *must hold for a unique element, and (b) every crate has a unique destination, so that coerce discards structures where c has none, or non-unique destinations.*

Note that in this example, different outcomes of focus operations can be easily differentiated on the basis of the number of elements of some role. After the first focus operation, 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 the conditions under which an action branch leading to a goal will be taken (Chapter 5).*

Action Application on Abstract Structures: Summary The overall approach for applying actions on abstract structures is shown in Fig. 3.5. The abstract structure is first focused w.r.t action-specific focus formulas. The resulting focused structures are

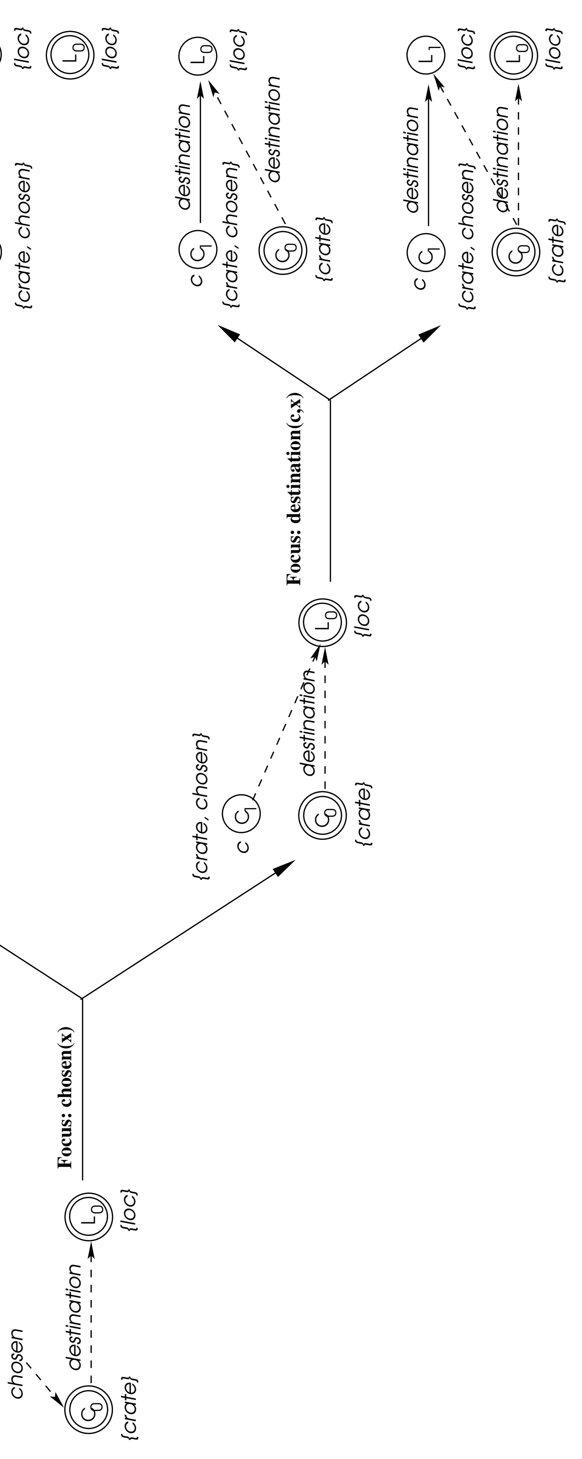


Figure 3.4: A sequence of focus operations in the delivery domain.

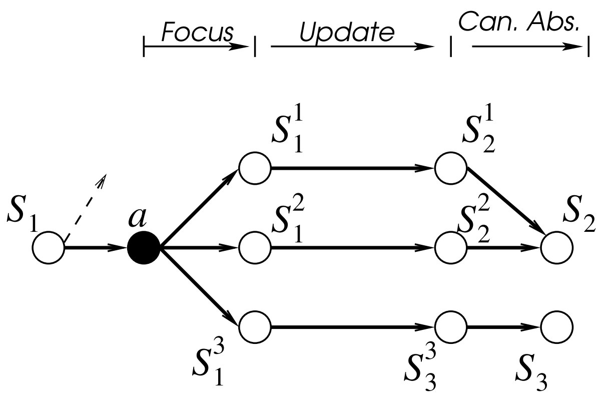


Figure 3.5: Action update mechanism

then tested against the preconditions, and action updates are applied to those that qualify. We then remove any constants representing action arguments, and canonically abstract the resulting structures, yielding the abstract result structures. The operation of canonical abstraction is referred to as "blurring" in TVLA, and is denoted with a b in the transitions defined below.

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

Definition 8. (Action Transition) *Let a be an action and S_1 a three-valued structure with constants representing each of a's arguments. $S_1 \xrightarrow{a} S_2$ holds iff S_1 and S_2 are three-valued structures and there exists a focused structure $S_1^1 \in f_{F_a}(S_1)$ s.t. $S_2 = \text{canon}(\tau_a(S_1^1))$. The transition $S_1 \xrightarrow{a} S_2$ can be decomposed into a set of transition sequences for each result of the focus operation: $\{(S_1 \xrightarrow{f_{F_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2) | S_1^i \in f_{F_a}(S_1) \wedge S_2^i = \tau_a(S_1^i) \wedge S_2 = \text{canon}(S_2^i)\}$.*

The local safety theorem (Sagiv, Reps, and Wilhelm, 2002) shows that this action mechanism is sound: the results of action application on an abstract state S_1 represent all possible results of action application on $\gamma(S_1)$. Consequently, any property true of all the abstract result states must be true of all the truly possible results of action application on $\gamma(S_1)$. In the following statement, extend the notation of τ_a to represent its element-wise application on a set of states.

Theorem 1. (Local Safety Theorem) *(Sagiv, Reps, and Wilhelm, 2002) Let $S_1, \ldots, S_n$ be the results of applying action a on S_0. Then $\tau_a(\gamma(S_0)) \subseteq \cup_{i=1\ldots n}\gamma(S_i)$.*

3.5 Belief States and Sensing Actions

The abstraction methodology described in the previous sections translates the generalized planning problem into a contingent planning problem with partially observable states. More precisely, this abstraction results in a state space with uncertainty about object quantities and properties, such that information about object quantities available to the agent during plan execution is sufficient only to determine if there are multiple, none, or exactly one element of each role.

These abstract states represent sets of possible concrete states in a manner similar to the modeling approach used in contingent planning, where belief states (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005) represent sets of possible real world states which are indistinguishable due to lack of information. Current belief state representations, however, do not capture uncertainty in object quantities. Contingent planners use "sensing" actions to determine properties of belief states. A sensing action leads to multiple possible belief states, corresponding to the different values of the property being sensed.

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

Example 4. *A partially observable version of the delivery domain can be constructed by adding uncertainty about the number of crates and locations and the* destination *relation. The canonically abstracted structure on the right in Fig. 3.2 can be used to represent the* *belief state of such a formalization. We can define a sensing action,* findDest(c,d)*, for determining a crate's destination using the focus formula* dest(c,l) *and update formulas setting a new constant* d *to the crate's destination. This formulation allows us to solve the sensing version of the delivery problem, as discussed in Section 6.2.4.*

3.6 Discussion

3.6.1 State and Action Representations

Many plan and knowledge representation systems have been proposed for planning. Early approaches, prior to the development of the STRIPS framework (Fikes and Nilsson, 1971) relied upon first-order logic both as a representation and as the inference mechanism for determining the effect of an action on a state. These approaches suffered from the *frame* problem due to the need for axioms specifying all facts that were *not* changed by an action. The STRIPS approach solved this problem by using an extra-logical mechanism for action updates. A state was represented as the set of propositions true in that state; actions consisted of add and delete lists, consisting of propositions to be added or removed from that state. In doing so however, STRIPS sacrifices existential quantification, because of which arguments of actions in a STRIPS specification must include every object whose properties may be changed. An action for moving block x to block y, when both x and y are topmost, would require as arguments x, y, as well as the block on which x rests–so that the topmost predicate can be set for this block during the action update.

Although situation calculus (Levesque, Pirri, and Reiter, 1998) follows a pre-STRIPS approach, it avoids the frame problem by using *successor state axioms*. These axioms specify the conditions under which predicates hold, or cease to hold in a manner similar to Eq. 3.1 on page 27:

$$pre(a,s) \rightarrow \Big[\big(p(\bar{x}, do(a,s)) \leftrightarrow \gamma_F^{+}(\bar{x},a,s)\big) \vee \big(p(\bar{x},s) \wedge \neg\gamma_F^{-}(\bar{x},a,s)\big)\Big] \qquad (3.2)$$

In this equation capturing the effect of action a on predicate p when applied in situation s, γ_F^+ represents the condition under which p becomes true and γ_F^-, the condition under which it becomes false. $pre(a,s)$ holds if the preconditions of a are satisfied in s.

Our approach is similar to situation calculus in utilizing a first-order predicate update equation, but also similar to the STRIPS approach in using a query evaluation on a state represented as a first-order structure to apply the action update. Eq. 3.1 presents a convenient template for predicate re-definitions which we use throughout. However, it can be replaced by a more general formula without changing the mechanism for applying actions on concrete states.

Our approach also differs from existing action representation approaches in using explicit *choice* actions to select action arguments. Making argument choices explicit allows us to generalize these choices as well as to efficiently account for their cost during plan instantiation.

Finally, integrity constraints allow us to specify constraints such as: "if a room is on fire, its hallway must have smoke" for the firefighting problem.

3.6.2 Generalized Plan Representations

As discussed in Section 3.2, prior approaches have addressed the problem of representing generalized, algorithmic plans. Representations such as robot programs (Levesque, 2005) and dsPlanners (Winner and Veloso, 2003) have been used for the construction of generalized plans; representations such as Golog and LPP (Baier et al., 2009) have been used to design generalized plans by hand, for encoding preferred planning operations. Graph based generalized plans allow us to easily represent control flows while making their structure explicit. In Chapter 4, we will use this to categorize a class of generalized plans for which preconditions can be found accurately and efficiently. Subsequently, while constructing generalized plans, we will use their edge labels to represent abstract structures capturing the set of states possible at any point in the generalized plan. These labels turn out to be very useful, and serve multiple purposes in addition to determining the flow of control during execution: in Section 6.2, we will use these labels to efficiently merge segments of different example plans which handle the similar problem instances; in Chapter 7, these labels are used to extend partial generalized plans to handle new problem instances.

3.6.3 State Abstraction

We use abstraction for *state aggregation*, which has been extensively studied for efficiently representing universal plans (Cimatti, Roveri, and Traverso, 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, canonical abstraction can construct aggregations of states from different problem instances with different numbers of objects.

Hoffmann, Sabharwal, and Domshlak (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). Our objective is to use abstraction for the different problem of simultaneously planning in infinitely many state spaces.

CHAPTER 4

ANALYZING ABACUS PROGRAMS WITH LOOPS

In order to be able to construct generalized plans with loops, we first study how to determine if a potential loop of actions will make progress towards the goal, or, at the least, terminate after a bounded number of iterations. With such methods in hand, we will construct generalized plans by evaluating the benefit of creating possible loops of actions.

We start by studying a powerful model of computation called abacus programs. We will solve the problem of determining loop termination and progress for some classes of abacus programs; in the next chapter, we will show how plans with loops in many problem domains can be directly translated into such abacus programs. Finally, we will use these techniques in developing algorithms for finding generalized plans in Chapters 6 and 7.

4.1 Abacus Programs

Abacus programs (Lambek, 1961) are finite automata whose states are labeled with actions that increment or decrement a fixed set of registers. Formally,

Definition 9. (Abacus Programs) *An abacus program* $\langle \mathscr{R}, \mathscr{S}, s_0, s_h, \ell \rangle$ *consists of a finite set of registers* $\mathscr{R}$*, a finite set of states* $\mathscr{S}$ *with special initial and halting states* $s_0, s_h \in \mathscr{S}$ *and a labeling function* $\ell : \mathscr{S} \setminus \{s_h\} \mapsto \text{Act}$. *The set of actions,* Act, *consists of actions of the form:*

- $Inc(r,s)$*: increment* $r \in \mathscr{R}$*; goto* $s \in \mathscr{S}$*, and*

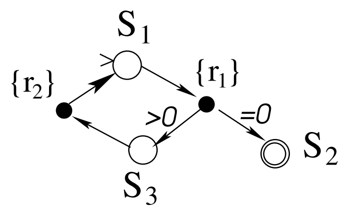


Figure 4.1: An abacus machine for the program: `while` $(r_1 > 0)$ $\{ r_1--; r_2++\}$

- *Dec*(r, s_1, s_2): *if* $r = 0$ *goto* $s_1 \in \mathscr{S}$ *else decrement* r *and goto* $s_2 \in \mathscr{S}$

We represent abacus programs as bipartite graphs with edges from states to actions and from actions to states. In order to distinguish abacus program states from states in planning, we will refer to a state in the graph of an abacus program as a “node". The two edges out of a decrement action are labeled $= 0$ and > 0 respectively (see Fig. 4.1).

Given an initial valuation of its registers, the execution of an abacus program starts at s_0. At every step, an action is executed, the corresponding register is updated, and a new node is reached. An abacus program *terminates* iff its execution reaches the halt node. The set of final register values in this case is called the *output* of the abacus program.

Abacus programs are equivalent to Minsky Machines (Minsky, 1967), which are as powerful as Turing machines and thus have an undecidable halting problem:

Fact 1. *The problem of determining the set of initial register values for which an abacus program will reach the halt node is undecidable.*

Nevertheless, for some abacus programs halting *is* decidable, depending on the complexity of the loops. A *simple loop* is a cycle. A *simple-loop* abacus program is one all of whose non-trivial strongly connected components are simple loops. In the next section we show that for any simple-loop abacus program, we can efficiently characterize the exact set of register values that lead not just to termination, but to any desired “goal” node defined by a given set of register values (Theorem 2).

4.2 Applicability Conditions for Simple Loops

Let $S_1, a_1, \ldots, S_n, a_n, S_1$ be a simple loop (see Fig. 4.2). We denote register values at nodes using vectors. For example, $\bar{R}^0 = \langle R_1^0, R_2^0, \ldots, R_m^0 \rangle$ denotes the initial values of registers $R_1, \ldots, R_m$ at node S_1. Let $a(i)$ denote the index of the register changed by action a_i. Since these are abacus actions, if there is a branch at a_i, it will be determined by whether or not the value of $R_{a(i)}$ is greater than or equal to 0 at the *previous* node.

We use subscripts on vectors to project the corresponding registers, so that the initial count of action a_i's register can be represented as $\bar{R}^0_{a(i)}$. Let Δ^i denote the vector of changes in register values $R_1, \ldots, R_m$ for action a_i corresponding to its branch along the loop. Let $\Delta^{1..i} = \Delta^1 + \Delta^2 + \cdots + \Delta^i$ denote the register-change vector due to a sequence of abacus actions $a_1, \ldots, a_i$. Given a linear segment of an abacus program, we can easily compute the preconditions for reaching a particular register value and node combination:

Proposition 1. *Suppose* $S_1 \xrightarrow{a_1} S_2 \xrightarrow{a_2} \cdots S_n$ *is a linear segment of an abacus program where* S_i *are nodes,* a_i *are actions and* $\bar{F}$ *is a vector of register values. A set of necessary and sufficient linear constraints on the initial register values* $\bar{R}^0$ *at* S_1 *can be computed under which* S_n *will be reached with register values* $\bar{F}$*.*

Proof. (Sketch) We know $\bar{F} = \bar{R}^0 + \Delta^{1..n}$. We only need to collect the conditions necessary to take all the correct action branches, keeping us on this path. This can be done by computing the register values at each node S_i in terms of $\bar{R}^0$, and using this expression to state the required inequality for following the required branch of the next action. □

Proposition 2. *Suppose we are given a simple loop,* $S_1, a_1, \ldots, S_n, a_n, S_1$*, of an abacus program. Then in* $O(n)$ *time we can compute a set of linear constraints,* $C(\bar{R}^0, \bar{F}, l)$*, that are satisfied by initial and final register tuples,* $\bar{R}^0, \bar{F}$*, and natural number,* l*, iff starting*

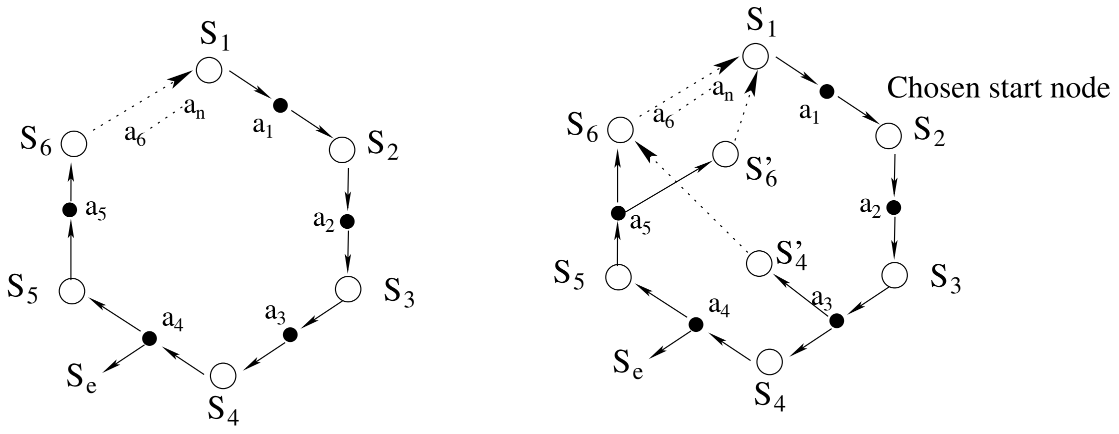


Figure 4.2: A simple loop with (right) and without (left) shortcuts

an execution at S_1 with register values $\bar{R}^0$ will result in l iterations of the loop, after which we will be in S_1 with register values $\bar{F}$.

Proof. Consider the action a_4 in the left loop in Fig. 4.2. Suppose that the condition that causes us to stay in the loop after action a_4 is that $R_{a(4)} > 0$. Then the loop branch is taken during the first iteration starting with fluent-vector $\bar{R}^0$ if $(\bar{R}^0 + \Delta^{1..3})_{a(4)} > 0$. This branch will be taken in l subsequent loop iterations iff $(\bar{R}^0 + k \cdot \Delta^{1..n} + \Delta^{1..3})_{a(4)} > 0$, and similar inequalities hold for every branching action, for *all* $k \in \{0, \ldots, l-1\}$. More precisely, for one full execution of the loop starting with $\bar{R}^0$ we require, for all $i \in \{1, \ldots, n\}$:

$$(\bar{R}^0 + \Delta^{1..i-1})_{a(i)} \circ 0$$

where $\circ$ is one of $\{>, =\}$ depending on the branch that lies in the loop; (this set of inequalities can be simplified by removing constraints that are subsumed by others). Since the only variable term in this set of inequalities is $\bar{R}^0$, we represent them as $\mathsf{LoopIneq}(\bar{R}^0)$. Let $\bar{R}^l = \bar{R}^0 + l \times \Delta^{1..n}$, the register vector after l complete iterations. Thus, for executing the loop completely l times, the required conditions are $\mathsf{LoopIneq}(\bar{R}^0) \wedge \mathsf{LoopIneq}(\bar{R}^{l-1})$. These two sets of conditions ensure that the conditions for execution of intermediate loop iterations hold, because the changes in register values due to actions are constant, and the expression for $\bar{R}^{l-1}$ is linear in them. Note that these conditions are necessary and sufficient since there is no other way of executing a complete iteration of the loop except by undergoing all the register changes and satisfying all the branch conditions.

Hence, the necessary and sufficient conditions for achieving the given register-value after l complete iterations are:

$$C(\bar{R}^0, \bar{F}, l) \equiv \mathsf{LoopIneq}(\bar{R}^0) \wedge \mathsf{LoopIneq}(\bar{R}^{l-1}) \wedge (\bar{F} = \bar{R}^l).$$

Each loop inequality is constant size because it concerns a single register. The total length of all the inequalities is $O(n)$ and as described above they can be computed in a total of $O(n)$ time. $\square$

Note that an exit during the first iteration amounts to a linear segment of actions and is handled by Prop. 1. Further, the vector $\bar{F}$ can include symbolic expressions. Initial values R^0 can be computed using $R^l = F$; these expressions for R^0 can be used as target values for subsequent applications of Prop. 2. Therefore, when used in combination with Prop. 1, the method outlined above produces the necessary and sufficient conditions for reaching any node and register value in an abacus program:

Theorem 2. *Let Π_A be a simple-loop abacus program. Let S be any node in the program, and $\bar{F}$ a vector of register values. We can then compute a disjunction of linear constraints on the initial register values that is a necessary and sufficient condition for reaching S with the register values $\bar{F}$.*

Proof. Since Π_A is acyclic except for simple loops, it can be decomposed into a set of segments starting at the common start-node, but consisting only of linear paths and simple loops (this may require duplication of nodes following a node where different branches of the plan merge). By Prop. 1 and 2, necessary and sufficient conditions for each of these segments can be computed. The disjunctive union of these conditions gives the overall necessary and sufficient condition. □

4.3 Nested Loops Due to Shortcuts

Due to the undecidability of the halting problem for abacus programs, it is impossible to find preconditions of abacus programs with arbitrarily nested loops. The previous section demonstrates, however, that structurally restricted classes of abacus programs admit efficient applicability tests. Characterizing the precise boundary between decidability and undecidability of abacus programs in terms of their structural complexity is an important open problem.

In this section, we show that methods developed in the previous section *can* be extended to a class of nested loops caused due to non-deterministic actions. Non-deterministic actions are common in planning but do not exist in the original definition of abacus domains. In the representation of Definition 9, we define a non-deterministic action in an abacus program $NSet(r, s_1, s_2)$ as follows:

- $NSet(r, s_1, s_2)$: set r to 0 and goto $s_1 \in S$ **or** set r to 1 and goto $s_2 \in S$.

We assume that the register r is new, or unused by deterministic actions. A non-deterministic action thus has two outgoing edges in the graph representation, corresponding to the two possible values it can assign to a register value. Either of these branches may be taken during execution. Although the original formulation of abacus programs is sufficient to capture any computation, the inclusion of non-deterministic actions allows us to conveniently treat a powerful class of nested loops (encountered in partially observable planning) as a set of independent simple loops.

Definition 10. (Complex Loops) *A* complex *loop in a graph is a non-trivial strongly connected component that is not a simple loop.*

Definition 11. (Shortcuts) *A* shortcut *in a simple loop is a linear segment of nodes starting with a branch caused due to a non-deterministic action in the loop and ending at any subsequent node in the loop, but not after a* designated *start node. The start node must precede all of the loop's shortcuts (e.g., node S_2 in Fig.4.2).*

For ease in exposition, we assume that no branch originating at a node on the shortcut but not on the simple loop leads to a node on the simple loop. However, methods for computing applicability conditions in the following sections can be easily applied to such shortcuts with branches: the fundamental idea is to treat each complete loop execution possible around the start node as a simple loop, and this extends naturally to situations with branches on shortcuts.

The definition above also constrains shortcuts to originate from a branch of a non-deterministic action; shortcuts beginning with branches of deterministic, register-decrementing actions are considered in Sections 4.3.3.1 and 4.3.3.2.

In terms of graph structure, simple loops with shortcuts categorize the class of graphs with *cycle rank* (Eggan, 1963) one. This class of graphs captures many common control flows, including those with doubly nested loops and nested `for` loops such as:

```
for i=1 to n do {for j=1 to k do {xyz}}.
```

Actions which create shortcuts in such loops can be easily transformed into non-deterministic actions followed by actions with the original conditions.

4.3.1 Applicability Conditions for Monotone Shortcuts

We now consider a special class of simple loops with shortcuts, where the shortcuts are *monotone*:

Definition 12. (Monotone Shortcuts) *The shortcuts of a simple loop are* monotone *if the sign (positive or negative) of the net change, if any, in a register's value is the same due to every simple loop created by the shortcuts.*

For ease of exposition we require that the start nodes of all shortcuts in a simple loop occur either at the common start node, or before the end node of any other shortcut, making shortcuts non-composable (i.e., only one shortcut can be taken in every iteration). Non-composability allows us to easily count the simple loops caused due to shortcuts independently. For instance, we can view the loop with shortcuts in Fig. 4.2 as consisting of 3 different simple loops. Which loop is taken during execution will depend on the results of non-deterministic actions a_3 and a_5. Additionally, we will only consider the case where non-deterministic actions occur on the outer, simple loop. Composable shortcuts and branches caused due to non-deterministic actions on shortcuts can be handled similarly by considering all possible completions of the loop independently, as simple loops. However, this may result in exponentially many simple loops in the worst case.

Suppose an abacus program Π is a simple loop with m monotone shortcuts and a chosen start node S_{start}. We consider the case of l complete iterations of Π counted at its start node, with $k_1, \ldots, k_m$ representing the number of times shortcuts $1, \ldots, m$ are taken, respectively. The final, partial iteration and the loop exit can be along any of the shortcuts, or the outer simple loop, and can be handled as a linear program segment. Let k_0 be the number of times the underlying simple loop is executed without taking any shortcuts. Then,

$$k_0 + k_1 + \ldots k_m = l \tag{4.1}$$

Determining Final Register Values We denote the loop created by taking the i-th shortcut as $loop_i$, and the original simple loop taken when none of the shortcuts are taken as $loop_0$. The final register values after the $l = \sum_{i=0}^{m} k_i$ complete iterations can be obtained by adding the changes due to each simple loop, with Δ^{loop_i} denoting the change vector due to $loop_i$:

$$\bar{F} = \bar{R}^0 + \sum_{i=0}^{m} k_i \Delta^{loop_i} \tag{4.2}$$

Cumulative Branch Conditions For computing sufficient conditions on the achievable register values after $k_0, \ldots, k_m$ complete iterations of the given loops, the approach is to treat each loop as a simple loop and determine its preconditions. Note that every required condition for a loop's complete iteration stems from a comparison of a register's value with zero. We therefore want to determine the lowest possible value of each register during the $k_0, k_1, \ldots k_m$ iterations of loops $0, \ldots, m$, and constrain that value to be greater than zero. For every register R_j, we first identify the index of simple loop which can cause the greatest negative change in a single, partial iteration starting at S_{start}, as $min(j)$, and the value of this change as $\delta_{min(j)}$. For readability we will use $\widehat{\jmath}$ to denote $min(j)$.

Let R^+ and R^- be the sets of registers undergoing *net* positive and negative changes respectively, by any loop. For $R_j \in R^+$, the lowest possible value is $R_j^0 + \delta_{\widehat{\jmath}}$. The required constraint on R_j is simply $R_j^0 + \delta_{\widehat{\jmath}} \geq 0$ ("$\geq$" because "$>$" must hold *before* the decrement), since the value of R_j can only increase after the first iteration. For $R_j \in R^-$, the lowest possible value is $R_j^0 + \sum_{i \neq \widehat{\jmath}} k_i \Delta_j^{loop_i} + (k_{\widehat{\jmath}} - 1)\Delta_j^{loop_{\widehat{\jmath}}} + \delta_{\widehat{\jmath}}$, achieved when $loop_{\widehat{\jmath}}$ is executed at the end, after all the iterations of the other loops. This leads to the following inequalities:

$$\forall R_j \in R^- \{R_j^0 + \sum_{i=0}^{m} k_i \Delta_j^{loop_i} + \delta_{\widehat{\jmath}} - \Delta_j^{loop_{\widehat{\jmath}}} \geq 0\} \qquad \text{(Sufficient(1))}$$

$$\forall R_j \in R^+ \{R_j^0 + \delta_{\widehat{\jmath}} \geq 0\} \qquad \text{(Sufficient(2))}$$

Together with Eqs. (4.1 & 4.2), these inequalities provide sufficient conditions binding reachable register values with the number of loop iterations and the initial register values. However, the process for deriving them assumed that for every j, $loop_{\widehat{\jmath}}$ will be executed at least once. We can make these constraints more accurate by using a disjunctive formulation for selecting the loop causing the greatest negative change among those that are executed at least once. For register R_j, let $0_j, \ldots, m_j$ be the ordering of loops in decreasing order of negative change values caused by an initial segment of the loop starting at S_{start}. We use $k_{i<x} = 0$ as an abbreviation for $\forall i < x$: $k_i = 0$. We can then write a disjunction of constraints corresponding to the first loop in $loop_{0_j}, \ldots loop_{m_j}$ which has non-zero iterations:

$$\forall R_j \in R^- \bigvee_{x=0_j,\ldots,m_j} \{k_{i<x} = 0; k_x \neq 0;$$

$$R_j^0 + \sum_{i \geq x} k_i \Delta_j^{loop_i} + \delta_x - \Delta_j^{loop_x} \geq 0\} \tag{4.3}$$

$$\forall R_j \in R^+ \bigvee_{x=0_j,\ldots,m_j} \{k_{i<x} = 0; k_x \neq 0; R_j^0 + \delta_x \geq 0\} \tag{4.4}$$

Constraints 4.3 & 4.4 are derived from the unnumbered constraints above by replacing $\hat{\jmath}$ with x, which iterates over loops in the order $0_j..m_j$, specific to register R_j; δ_x represents the greatest negative change in loop x for role j.

If the partial negative change for every register in every loop is at most the net negative change of that loop, conditions 4.3 & 4.4 reduce to the conditions labeled Sufficient (1 & 2) above. This is because δ_x becomes equal to $\Delta_j^{loop_x}$ in Eq. 4.3; all of the disjuncts in Eq. 4.3 then reduce to $\forall R_j \in R^- \{R_j^0 + \sum_{i=0}^{m} k_i \Delta_j^{loop_i} \geq 0\}$. Further, Eq. 4.4 and Sufficient (2) become vacuously true because the partial negative change cannot be more than the net negative change (zero) for registers in R^-.

Accuracy of the Computed Conditions Note that these conditions do not deal with equality conditions that may have to be satisfied for staying in a loop. Equality conditions are very constraining, and may constrain the execution of a loop corresponding to a shortcut to occur exactly once, when the equality condition holds. However, conditions (4.1-4.4) can be extended to include equality conditions for the first and last iteration of each loop. This will make (4.1-4.4) sufficient conditions for situations where equality branches are required to stay in the loop (in our experience this is rare in planning domains). However, adding these constraints may also make (4.1-4.4) unsatisfiable if the same register is used in two different equality constraints corresponding to two different loops caused by shortcuts.

In order to discuss when conditions (4.1-4.4) are accurate and not over-constraining, we first define order independence:

Definition 13. (Order Independence) *A simple loop with shortcuts is order independent if for every initial valuation of the registers at S_{start}, the set of register-values possible at S_{start} after any number of iterations does not depend on the order in which the shortcuts are taken.*

An equality constraint in a loop is considered *spurious* if no loop created by the shortcuts changes the register on which equality is required. During the execution of the loop, the truth of such conditions will not change. Consequently, such equality conditions do not introduce order dependence. In practice, these conditions can be translated into conditions on register values just prior to entering the loop.

A simple loop with shortcuts will have to be order dependent if one of the following holds: (1) the lowest value achievable by a register during its execution depends on the order in which shortcuts are taken. In this case, possible lowest values will impose different constraints for each ordering; or, (2) a *non-spurious* equality condition has to be satisfied to stay in a loop. In the latter case, the non-deterministic branch leading to the shortcut that has the equality condition will have to be taken at the precise iteration when equality is satisfied. In fact, the disjunction of these two conditions is necessary and sufficient for a loop to be order dependent.

Proposition 3. *A simple loop with shortcuts is order-dependent iff either (1) the lowest value achievable by a register during its execution depends on the order in which shortcuts are taken or (2) a non-spurious equality condition has to be satisfied to continue a loop iteration.*

Proof. Sufficiency of the condition was discussed above. If the loop is order dependent, then there is a register value that is reachable only via a “good" subset of the possible orderings of shortcuts. Consider an ordering with the same number of iterations of these shortcuts, not belonging to this subset. During the execution of this sequence, there must be a first step after which a loop iteration that could be completed in the good subset, cannot be completed in the chosen ordering. This has to be either because an inequality > 0 is not satisfied before a decrement, which implies (1) holds, or because $R_j = 0$ is required to continue the iteration; this must have been possible in the good loop orderings, but $R_j > 0$ must hold here, which implies case (2) holds. □

A naive approach of even expressing the necessary conditions for an order dependent loop can be exponential in the number of shortcuts, even while considering just a single iteration of each loop. Deriving better representations for such conditions is an important direction for future work.

Example 5. *Consider loops l_1, l_2 created by shortcuts in a larger loop. l_1 increases R_1 by 5 and R_2 by 1. l_2 first decreases R_1 by 4 and then increases it by 5. l_1, l_2 are monotone shortcuts but their combination is order dependent: at S_{start} with $R_1 = 1$, l_2 cannot be executed completely before executing l_1. Expressing precise preconditions for reachable register values thus requires a specification of the order in which the shortcuts have to be taken.*

We can now present two results capturing the accuracy of the conditions (4.1-4.4).

Proposition 4. *If Π is an order independent simple loop with monotone shortcuts, then Eqs. (4.1-4.4) provide necessary and sufficient conditions on the initial and achievable register values.*

Proof. By construction, the inequalities ensure that none of the register values drops to zero, so that if a register value satisfies the inequalities, then it will be reachable.

This proves that the conditions are sufficient. Suppose that a register value $\bar{F}$ is reachable from $\bar{R}^0$, after $k_0, \ldots k_m$ iterations of $loop_0, \ldots, loop_m$ respectively. Eq. (2) cannot be violated, because the changes caused due to the loops are fixed; Eq. (1) will be satisfied trivially. If $\bar{R}^0, k_0, \ldots, k_m$ don't satisfy Eqs. (4.3-4.4), the lowest value achieved during the loop iterations will fall below zero because the loop is order independent. Therefore, (4.1-4.4) must be satisfied. □

Proposition 5. *If* Π *is a simple loop with monotone shortcuts, then Eqs. (4.1-4.4), together with constraints required for equality branches during the first and last iterations of the shortcuts containing them give sufficient conditions on the possible final register values in terms of their initial values.*

Proof. By construction, conditions (4.1-4.4) and the equality constraints ensure that every branch required to complete k_i iterations of loop i will be satisfied. □

This leads to the main result of this section, which is analogous to Theorem 2 for simple loops:

Theorem 3. *Let* Π *be an abacus program, all of whose strongly connected components are simple loops with monotone shortcuts. Let S be any node in the program, and $\bar{F}$ a vector of register values. We can then compute a disjunction of linear constraints on the initial register values for reaching S with the register values $\bar{F}$. If all simple loops with shortcuts in* Π *are order independent, the obtained precondition is necessary and sufficient.*

Proof. Similar to the proof by decomposition for Theorem 2, using Propositions 4 and 5. □

Semantics of the Computed Conditions In the result and the conditions constructed above, the k_i variables, which count the number of times a non-deterministic action effect occurs, appear to be measuring an inherently unpredictable property (non-determinism) and seem to mitigate the utility of the computed preconditions. However, as we will see in the next section, non-deterministic abacus actions may stand for sensing actions; while we may not be able to predict the outcome of each sensing action, it may still be possible to know how many times a certain outcome is possible, which is all that we need for the conditions above. In addition, if the k_i are used as parameters, the sufficient conditions above capture their tolerable values under which a desired register value may be achieved.

While our methods seem to be imprecise (sufficient but not necessary) for loops with order dependent shortcuts, the non-determinism behind such shortcuts significantly undermines the value of complete necessary and sufficient conditions expressed in terms of the possible loop orderings: these orderings cannot be predicted in advance as they depend on the order in which different outcomes of non-deterministic actions will be taken. In other words, in an adversarial formulation of the problem where nature "controls" the non-deterministic action outcomes, we actually *need* the worst-case conditions provided by Eqs. (4.1-4.4).

4.3.2 Relaxing Monotonocity

Although the introduction of non-deterministic actions makes it easier to express plans with sensing actions, it significantly adds to the power of abacus programs consisting of simple loops with shortcuts. Specifically, we show below that reachability in an abacus program consisting of a simple loop with non-monotone shortcuts is at least as hard as the problem of reachability in a vector addition system (Hopcroft and Pansiot, 1979). Vector addition systems are not as powerful as Turing machines, but still have a hard reachability problem. Although it has been proved that reachability in vector addition systems is decidable, known algorithms require non-primitive-recursive space (Kosaraju, 1982). We use the formalization of vector addition systems as presented by Hopcroft and Pansiot (1979):

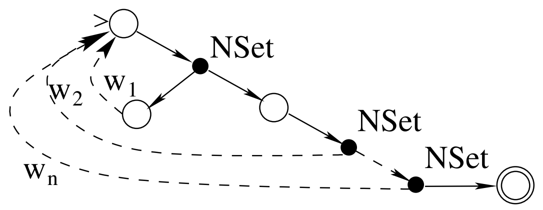


Figure 4.3: Reduction of a vector addition system to a non-deterministic abacus program.

Definition 14. (Vector addition systems) *An n-dimensional vector addition system (VAS) is a pair (x, W) where $x \in \mathbb{N}^n$ is called the start point and W is a finite subset of $\mathbb{Z}^n$. The* reachability set *of the VAS (x, W) is the set of all z, $z = x + v_1 + \cdots + v_j$ where each $v_i \in W$ and all the components of all intermediate sums, $x + v_1 + \cdots + v_i$, $1 \le i \le j$ are non-negative.*

Proposition 6. *Determining the set of register values reachable at the start node of an abacus program consisting of a simple loop with shortcuts is at least as hard as the problem of determining the reachable register values in a vector addition system.*

Proof. Suppose (x, W) is an n-dimensional VAS. We construct an abacus program with n registers, one for each dimension in the given VAS. We can then construct a simple loop with shortcuts so that each of the simple loops created corresponds to a unique $w \in W$. The shortcut corresponding to w_i would consist of sequences of incrementing/decrementing actions for each dimension in w_i (see Fig. 4.3). For decrementing actions, the zero-branches lead to an exit from the loop to a trap state. In the resulting abacus program, a given configuration of register values is reachable at the start state iff there exists an ordering of the simple loops created by shortcuts which leads to it. In other words, every reachable register-value configuration corresponds to a sum of the w_i's with none of the intermediate values being negative. □

4.3.3 Simple Loops with Shortcuts due to Deterministic Actions

In this section we discuss simple loops with shortcuts without any non-deterministic actions.

Algorithm 1: Reachability for deterministic, monotone shortcuts

```
Input: Deterministic abacus program in the form of a simple loop with
       monotone shortcuts, an initial register configuration $\bar{R}^0$
Output: Sequence of (loop id, #iterations) tuples.
1  $\bar{R} \leftarrow \bar{R}^0$
2  Iterations ← empty list
3  LoopList ← simple loops created by shortcuts
4  while LoopList $\neq \emptyset$ do
5      if no $l \in$ LoopList satisfies $\mathsf{LoopIneq}_l(\bar{R})$ then
6          Return Iterations
       end
7      $l \leftarrow$ id of loop for which $\mathsf{LoopIneq}_l(\bar{R})$ holds
8      Remove $l$ from LoopList
9      $l_{max} \leftarrow$ FindMaxIterations($\bar{R}, l$)
10     Iterations.append(($l, l_{max}$))
11     $\bar{R} \leftarrow \bar{R} + l_{max}\Delta^l$
   end
12 Return Iterations
```

4.3.3.1 Monotone Shortcuts

Let Π_A be an abacus program in the form of a simple loop with shortcuts originating at deterministic (decrementing) actions. Taking an initial register valuation as input, Alg. 1 computes a sequence of tuples representing the order in which the simple loops created by shortcuts will be taken, and the number of times each such loop will be executed in this ordering. Such a sequence is sufficient to calculate all the reachable register values during an execution of the given program.

Alg. 1 relies on the following observations:

1. Because all the shortcuts are monotone, if a loop is executed for a certain number of iterations and then exited, the flow of control will never return to that loop.

2. For any given configuration of register values at the start node, at most one of the simple loops created by shortcuts may be completely executable. This is because if multiple simple loops can be executed starting from a given register value configuration, then at some action node in the program, it should be possible for the

control to flow along more than one outgoing edge. However, this is impossible because every action which has multiple outcomes (a decrementing action) has exactly two branches, whose conditions are always mutually *inconsistent*.

The overall algorithm works by identifying the unique loop l whose $\mathsf{LoopIneq}$ is satisfied by the value $\bar{R}$ (initialized to $\bar{R^0}$)[steps 5-8], calculating the number of iterations which will be executed for that loop until $\mathsf{LoopIneq}$ gets violated [step 9], updating the register values to reflect the effect of those iterations [step 11] and identifying the next loop to be executed [the while loop, step 4].

The subroutine FindMaxIterations uses the inequalities in $\mathsf{LoopIneq}$ (see prop. 2) to construct the vector equation $(\bar{R} + l_{max}\Delta^l + \Delta^{1..i-1})_{a(i)} \circ 0$ for every action in loop l. This system of equations consists of an inequality of the following form for every i corresponding to a decrementing action in the loop:

$$l_{max} < (\bar{R}_{a(i)} + \Delta^{1..i-1}_{a(i)})/\Delta^l_{a(i)}$$

Since $\bar{R}$ is always known during the computation, the floor of the minimum of the RHS of these equations for all i yield the largest possible value of l_{max}. Equality constraints either drop out (if the net change in their register's value due to the loop l is zero and they are satisfied during the first iteration), or set $l_{max} = 1$ (if the net change in their register's value is not zero, but it is satisfied during the first iteration). Note that if there is any loop which does not decrease any register's value, it will never terminate. This will be reflected in our computation by an l_{max} value of ∞.

Let b be the maximum number of branches in a loop created due to the shortcut, and L the total number of simple loops generated due to the shortcuts. The most expensive operation in this algorithm is step 5, where $\bar{R}$ is tested on every loop's inequality (these loop inequalities only need to be constructed once). Step 5 is executed in $O(Lb)$ time and step 9 in $O(b)$ time. The entire loop may be executed at most L times, resulting in a total execution time of $O(L^2b)$. On the other hand, if such a program is

	Deterministic	Non-deterministic
Monotone Shortcuts	Alg. 1	Eq.(1-4) (order-indep.)
Non-monotone Shortcuts	unknown	VAS$\preceq$

Table 4.1: Known results on reachability for abacus programs of cycle rank one.

directly applied on a problem instance and the program terminates, then the execution time for the program will be of the order of the largest register value.

4.3.3.2 Non-monotone Shortcuts and Linear Hybrid Automata

Currently, the complexity and decidability of the problem of reachability for abacus programs consisting of simple loops with non-monotone, deterministic shortcuts is unknown. In general, reachability problems for abacus programs can be easily represented as reachability problems for linear hybrid automata (LHA) (Alur, Henzinger, and Ho, 1996). While a hybrid system is a model of computation which combines discrete state transitions with continuous flows of real-valued variables within each state, in linear hybrid automata the flows of these variables are constrained by linear expressions. Numerous implementations of approximate and partially decidable algorithms have been developed for model checking linear hybrid systems. Deterministic abacus programs with simple loops with shortcuts can be easily represented as a particularly simple class of linear hybrid systems, with register changes occurring as “jump" transitions between discrete states. We have obtained promising results using LHA analysis tools on these translated representations. A detailed analysis of their applicability, as well as the impact of monotonicity and our restrictions based on graph structure on LHA verification algorithms is left for future work.

Table 4.1 summarizes known results about determining the set of states reachable from a given initial state for abacus programs with simple loops with shortcuts.

4.4 Discussion

Summary The study of abacus programs with loops will allow us to efficiently analyze plans for handling unbounded numbers of elements in the next chapter. Although it is immediate that the problem of determining reachable states through a given abacus program cannot have a general solution, results in this chapter prove that for restricted classes of abacus programs reachability can be determined very efficiently. Analysis of this problem using the representation of abacus programs has an added benefit: we can use these results to control the growth of generalized plans along structures that are decidable; further, these results will allow us to determine the utility and safety of adding a possible loop to a generalized plan.

Directions for Future Work The results in this chapter initiate the study of reachability along classes of graphs. Directions for future work include a better understanding of the boundary between decidability and undecidability in terms of the structure of abacus programs, and also the impact of constraints such as monotonicity and cycle-rank one on related approaches for model checking of programs. Along these lines, it would also be informative to study if existing methods for model checking and analysis of programs as well as hybrid systems can be guaranteed to work on any class of abacus programs which relaxes these constraints.

Related Work Although various approaches have studied the utility and generation of plans with loops, very few provide any guarantees of termination or progress for their solutions. Approaches for strong cyclic planning (Cimatti et al., 2003) attempt to generate plans with loops for achieving temporally extended goals and for handling actions which may fail. Loops in strong cyclic plans are assumed to be *static*, with the same likelihood of a loop exit in every iteration. The structure of these plans is such that it is always *possible*–in the sense of graph connectivity–to exit all loops and reach the goal; termination is therefore guaranteed if this can be assumed to occur eventually. Among more recent work, KPLANNER (Levesque, 2005) attempts to find plans with loops that generalize a single numeric planning parameter. It guarantees that the obtained solutions will work in a user-specified interval of values of this parameter. DISTILL (Winner and Veloso, 2007) identifies loops from example traces but does not address the problem of preconditions or termination of its learned plans. Bonet, Palacios, and Geffner (2009) derive plans for problems with fixed sizes, but the controller representation that they use can be seen to work across many problem instances. They also do not address the problem of determining the problem instances on which their plans will work, or terminate. To the best of our knowledge, complexity of restricted classes of abacus programs in terms of graph structure has not been studied before. However, abacus programs have been utilized in studies of planning frameworks before, notably in the analysis of planning with numeric variables (Helmert, 2002).

Finding preconditions of linear segments of plans has been well studied in the planning literature. Triangle tables (Fikes, Hart, and Nilsson, 1972) can be viewed as a compilation of plan segments and their applicability conditions. However, there has been no concerted effort to find preconditions of plans with loops. Static analysis of programs deals with similar problems of finding program preconditions. These methods typically work with the weaker notion of *partial correctness*, where a program is guaranteed to provide correct results *if* it terminates. Methods like Terminator (Cook, Podelski, and Rybalchenko, 2006) specifically attempt to prove termination of loops, but do not provide precise preconditions or the number of iterations required for termination.

CHAPTER 5

TRANSFORMING PLANS INTO ABACUS PROGRAMS

Methods developed in the previous chapter can be used to find preconditions of plans with loops which satisfy the appropriate graph-structural requirements, as long as planning actions can be characterized in terms of simple numeric changes. One approach for achieving such actions is to consider their effects as changes in the number of elements satisfying different properties. However, determining the appropriate set of properties whose counts will be useful in characterizing actions is not always obvious.

In this chapter we show how canonical abstraction can be used to automatically translate action transitions in a class of planning domains into abacus actions by treating *role-counts* (the number of elements of each role in a structure) as register values for abacus actions. These methods will be used in the following chapters for construction of generalized plans with loops, termination guarantees, and preconditions.

5.1 Overview

We begin by illustrating the idea behind finding preconditions for success of action sequences on a special class of domains that use only unary predicates. These ideas are then generalized to abstract domains with binary relations that satisfy some key requirements (FC^3 domains, Definition 18). Section 5.3 presents a set of necessary conditions under which canonical abstraction produces FC^3 domains; the complexity of our algorithms is discussed in Section 5.3.1. Domains satisfying these necessary conditions are called *extended-LL domains*. Section 5.4 discusses an easily recognizable subclass of extended-LL domains; the transport example discussed below will turn out to be a member of this subclass. Section 5.5 formalizes the close relationships between generalized plans in extended-LL domains and abacus programs, allowing us to use results from the previous chapter to find plan preconditions in these domains. Section 5.6 demonstrates the power of the resulting methodology by applying it on ten different plans with loops which have been studied in the literature, albeit so far without approaches for reliably computing their effects.

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

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

Fig. 5.1 on the next page shows a sequence of action operations on an abstract initial structure S_1. For the purpose of this example, assume that the goal is to have exactly one object at $L1$, as in structure S_6. Note that this sequence of actions creates a loop; the only branch is caused by the first choice action. Unlike loops over a sequence of concrete states, this loop makes progress towards the goal. In the following development, we will measure this progress using *role-counts*:

Definition 15. (Role-count) *The* role-count *of a role R in a concrete structure C is the number of elements in C that satisfy R.*

Note that precise role-counts can only be computed for concrete structures. Summary elements in abstract structures obscure their role-counts; in the case of the action transitions in Fig. 5.1 however, it is possible to compute the precise *changes* in role-counts due to each action on its preceding abstract structure. It can also be proved

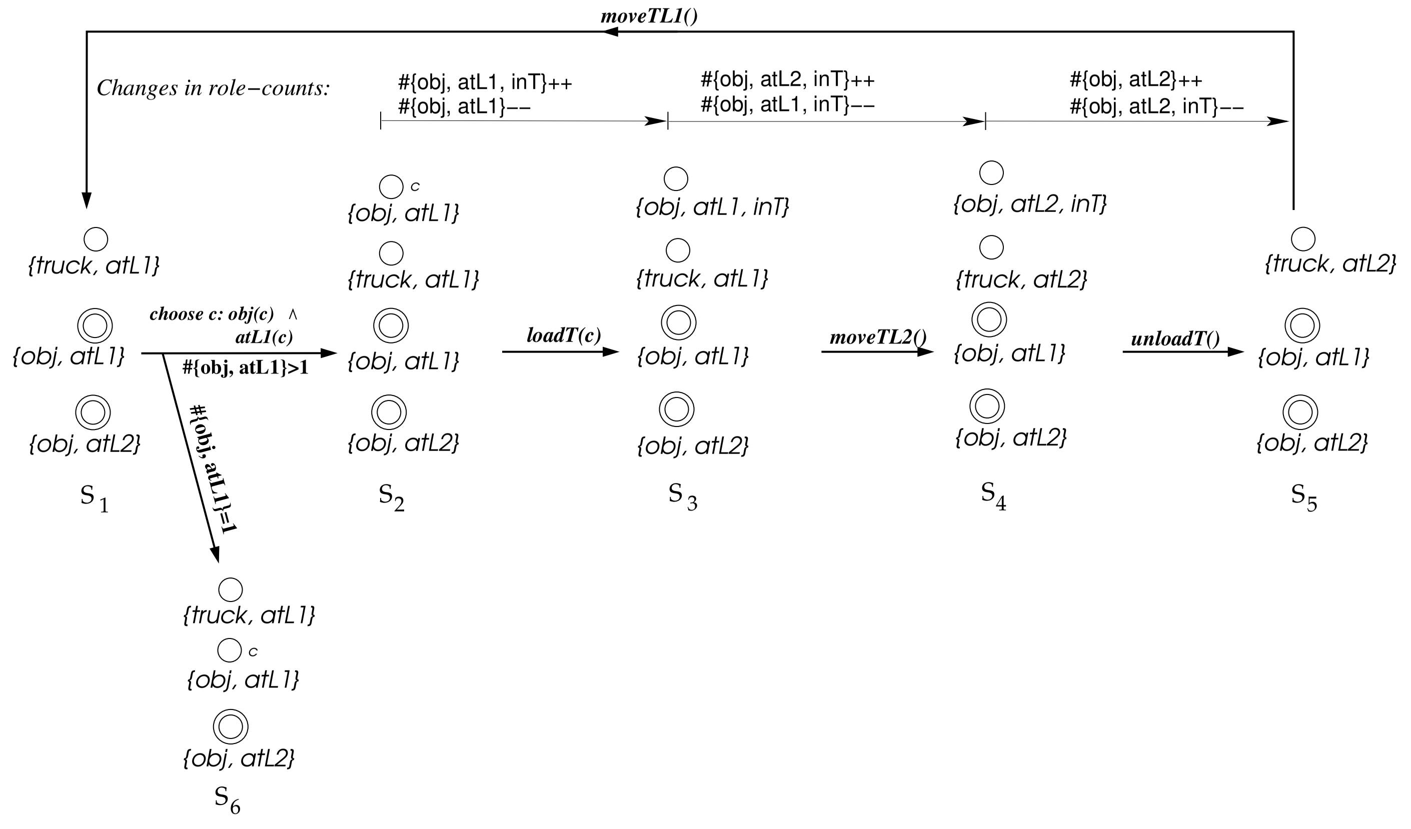


Figure 5.1: A sequence of actions in a unary representation of transport domain. Role-count changes are shown only for roles involving *object*, abbreviated as *obj*.

that every concrete structure represented by the abstract structures in Fig. 5.1 will undergo the same changes, as annotated near the top of the figure (this is not true in general for action application on abstract states). Further, the condition determining whether or not the branch exiting the loop is taken can be determined, and depends on a role-count.

Suppose n_1^0 denotes the role-count of $\{object, atL1\}$ for a concrete structure embeddable in S_1. The role-count change annotations near the top of Fig. 5.1 indicate that n_1^0 will drop by one in every iteration of the loop. Therefore, we can determine that the branch exiting the loop will be taken after exactly $n_1^0 - 1$ iterations. This means that

1. The goal is provably reachable from *any* of the infinitely many structures represented by S_1.

2. Given a structure $s \in S_1$ the number of steps required to reach the goal following the given loop can be easily determined.

In any domain representation constructed using just unary predicates if action arguments are drawn out prior to action application (sec. 3.4.3), it is possible to carry out this method of analysis to determine facts like (1) and (2) above for a generalized plan with any number of simple loops. In the remainder of this section we provide the details for a generalization of this technique to a broader class of domains.

5.2 FC^3 Conditions

The most important properties of the simplified transport domain that made it possible for us to determine preconditions over sequences and loops of actions are:

1. When an action has multiple abstract structures as outcomes, role-counts in the initial structure determine which branch will be taken.

2. Given a pair (S, a) where the arguments for a are instantiated by constants in the abstract structure S, the changes in role counts of every concrete structure

represented by S due to a are the same! This enables us to represent precisely the changes in role-counts caused by an action on an abstract structure.

Note that combining 2 with 1 above brings us close to the abacus program formulation; a precise relationship will be presented in the form of Lemma 2 and Theorem 7.

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

Focus-classifiability will allow us to categorize branches caused due to the focus operation in terms of simple inequalities, as in the case of the first action in Fig. 5.1.

Definition 16. (Focus Classifiability w.r.t. $\mathcal{R}$) *A focus operation f_F on a structure S satisfies* focus classifiability w.r.t. $\mathcal{R}$ *if for every $S_i \in f_F(S)$ it is possible to compute a constraint $l_j \in \mathcal{E}_{\mathcal{R}}$ such that for every $C \in \gamma(S)$, $C \in \gamma(S_i)$ iff $C \models l_j$.*

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

Definition 17. (Constant Change) *An action transition $S_1 \xrightarrow{f_{F_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2$ shows* constant change w.r.t. a set of roles $\mathcal{R}$ *iff there exists a constant δ_j for each $R_j \in \mathcal{R}$ such that whenever $C_1 \in \gamma(S_1^i), C_2 \in \gamma(S_2^i)$ and $C_1 \xrightarrow{a} C_2$, we have $\#_{R_j}(C_2) = \#_{R_j}(C_1) + \delta_j$.*

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

Definition 18. (FC^3 Domains) *Let $\mathscr{S}$ be a set of abstract states closed under transitions for actions from a set $\mathscr{A}$ (i.e., if $S_i \in \mathscr{S}$ and $S_i \xrightarrow{a_1} \cdots \xrightarrow{a_k} S_f$ with $a_1, \ldots, a_k \in \mathscr{A}$, then $S_f \in \mathscr{S}$). $\mathscr{S}$ is an FC^3 domain*[1] *w.r.t. $\mathscr{E}_{\mathscr{R}}$ and $\mathscr{A}$ iff for every $S_1 \in \mathscr{S}$ and transition $S_1 \xrightarrow{f_{F_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2$ where $a \in \mathscr{A}$, the focus operation satisfies focus classifiability, and the transition itself shows constant change w.r.t. $\mathscr{R}$.*

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

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

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

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

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

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

Theorem 4. (Preconditions for a linear sequence of structures and actions) *Suppose we have a sequence of actions $a_1, a_2, \ldots, a_n$ such that $S_1 \xrightarrow{f_{F_{a_1}}} S_1^i \xrightarrow{up_{a_1}} S_2^i \xrightarrow{b} S_2 \cdots \xrightarrow{b} S_n \xrightarrow{f_{F_{a_n}}} S_n^i \xrightarrow{up_{a_n}} S_{n+1}^i \xrightarrow{b} S_{n+1}$, in an FC^3 domain. Then we can find a constraint $l_{initial}$ on S_1 such that a member $C \in \gamma(S_1)$ reaches $S_{n+1}|_{l_{final}}$ along this path of transitions iff $C \in \gamma(S_1|_{l_{initial}})$.*

Theorem 4 mirrors Prop. 1 on page 49 for linear segments of abacus programs. FC^3 domains however, differ from abacus programs along one key aspect: action effects can be classified in terms of comparison with *any* constant in FC^3 domains as opposed to abacus domains where comparisons are only conducted with a fixed constant (zero). In spite of this, the approach developed for finding preconditions of simple loops of abacus programs can be applied to any FC^3 domain. In the case of simple loops with shortcuts however, the approach we developed for abacus programs requires that all comparisons be made with a unique constant.

The next section serves two purposes. First, it identifies some sufficient conditions under which canonical abstraction produces FC^3 domains. Second, these conditions enforce the restriction that different action outcomes be categorized in terms of comparisons between role-counts and a single constant (one).

5.3 Extended-LL Domains: Sufficient Conditions for Obtaining FC^3 Domains

In the previous section we developed a set of requirements on abstract state spaces under which we will be able to find preconditions of plans with simple loops. We now provide a set of sufficient conditions on abstract states and the syntax of action operations under which the FC^3 conditions are satisfied. In domains satisfying these conditions, constraints determining focus branches and role-count change vectors due to actions can be computed in time linear in the number of elements in the initial abstract structure.

We call a formula ϕ with a single free variable *role-specific* if it can only hold for objects of a certain specific role in a given structure. In other words, ϕ is role-specific in S iff there exists a role R such that for all $C \in \gamma(S)$ we have $C \models \forall x(\phi(x) \to R(x))$, where we use $R(x)$ as an abbreviation for the conjunction of predicates in R together with literals denoting negations of abstraction predicates not in R. The following proposition gives sufficient conditions for focus-classifiability. We call a formula "uniquely satisfiable" if it must hold for exactly one element.

Proposition 7. (Sufficient conditions for focus-classifiability) *If ψ is uniquely satisfiable in all $C \in \gamma(S)$ and role-specific in S then the focus operation f_ψ on S satisfies focus classifiability w.r.t $\mathscr{E}_{\mathscr{R}}$. Further, in this case the constraints used for classifying the focus branches are inequalities between role-counts and 1.*

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 satisfying the role while not satisfying the focus formula: either there is only one element of the role, and it satisfies the focus formula, or there are more than one elements of that role and one of them satisfies the focus formula (see Fig. 3.3). The branch is thus classifiable on the basis of the number of elements in the role ($= 1$ or > 1). $\square$

Note that if a focus formula is unsatisfiable in all $C \in \gamma(S)$, then the focus operation will have only one outcome, and branch classification will not be required. We can immediately extend Prop. 7 to a set of role-specific and uniquely satisfiable formulas as long as any pair of these formulas either always, or never coincide:

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

The condition of unique satisfiability on the focus formulas for actions (the $\Delta^{\pm}$ expressions) used in Prop. 7 and Corollary 1 also gives us actions with constant change:

Proposition 8. (Sufficient conditions for constant change) *Let a be an action whose predicate update formulas take the form shown in Eq. 3.1. Action a shows constant change if for every abstraction predicate p_i, all the expressions Δ_i^+, Δ_i^- are at most uniquely satisfiable.*

Proof. Suppose $S_1 \xrightarrow{f_{F_a}} S_1^i \xrightarrow{\tau_a} S_2^i \xrightarrow{b} S_2$; $C_1 \in \gamma(S_1^i)$ and $C_1 \xrightarrow{\tau_a} C_2 \in \gamma(S_2^i)$.

For constant change we need to show that $\#_{R_i}(C_2) = \#_{R_i}(C_1) + \delta$ where δ is a constant. Recall that a role is a set of abstraction predicates. Furthermore, because the set of focus formulas f_{F_a} consists of pairs of formulas Δ_i^+ and Δ_i^- for every abstraction predicate, and these formulas are at most uniquely satisfiable, each abstraction predicate changes on at most 2 elements. The focused structure S_1^i shows exactly which elements undergo change, and the roles that they leave or will enter.

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

Since the required conditions in Prop. 8 are subsumed by those in Corollary 1, Corollary 1 provides sufficient conditions under which a focus operation on an abstract structure satisfies the FC^3 conditions of focus classifiability and constant change.

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

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

Definition 19. (Extended-LL domains) *An extended-LL domain is a domain schema $\mathscr{D}$ with a start structure S_{start} such that all its actions' focus formulas F_{a_i} are role-specific,* *exclusive when not equivalent, and uniquely satisfiable in every structure reachable from a start structure* S_{start}.

More formally, if $S_{\text{start}} \rightarrow^* S$ *and* $\Delta_i^{\pm}$ *are the action specific focus formulas, then* $\forall i, j, \forall e, e' \in \{+, -\}$ *we have* Δ_i^e *role-specific and either* $\Delta_i^e \equiv \Delta_j^{e'}$ *or* $\Delta_i^e \implies \neg\Delta_j^{e'}$ *in* S.

Note that if actions can be decomposed so that each action operator has only one focus formula, the restriction of "exclusive when not equivalent" in Definition 19 becomes vacuously true.

Intuitively, extended-LL domain-schemas are those where the information captured by roles is sufficient to determine whether or not an object of any role will undergo change due to an action. Examples of such domains are linked lists, blocks-world scenarios, assembly domains where different objects can be constructed from constituent objects of different roles, and transport domains.

In general, domains can be proved to be extended-LL domains by inductively proving the properties in Definition 19 for all the structures reachable from a given start structure. In practice, this can be proved more easily. In the delivery domain for instance, the only focus operations correspond to choice operations (which satisfy the extended-LL conditions: the choice formulas are defined to be unique and role-specific) and the focus operation for crate destinations, which are also role-specific to the role *loc* and constrained to be unique.

Theorem 5. (Sufficient conditions for FC^3 domains: extended-LL domains) *The space of all reachable states in an extended-LL domain constitutes an* FC^3 *domain.*

5.3.1 Complexity of Finding Preconditions in Extended-LL Domains

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

Algorithm 2: ComputeSingleStepChanges

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

Conditions classifying branches from a structure S can also be computed efficiently in extended-LL domains: we know all action branches take place as a result of the focus operation. The role(s) responsible for the branch will have different numbers of elements in the focused structures prior to action update. Using a straightforward comparison of role counts, the responsible role and its counts (> 1 or $= 1$) for different branches can be found in $O(s)$ operations where s is the number of roles in S.

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

5.4 Classical Unary Domains

We can now see the motivating example shown in Fig. 5.1 as a special case of extended-LL domains where all the predicates in the vocabulary are unary. We define *classical unary domains* as domain-schemas with only unary predicates whose action updates can be represented using finite, but possibly conditional *add* and *delete* lists of properties.

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

$$up(p(x), a) \equiv \neg p(x) \wedge [\vee_{i=1..n}\{x = arg_i\} \wedge \Delta^+(x)] \tag{5.1}$$

$$\vee \quad p(x) \wedge \neg[\vee_{i=1..n}\{x = arg_i\} \wedge \Delta^-(x)] \tag{5.2}$$

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

Under canonical abstraction, such domains lose almost no information. Since we always draw-out action arguments prior to action application in the abstract state space, the update carried out by Eqs. (5.1) and (5.2) always shows constant change in the abstract state space (the reasons are similar to those in Prop. 8). Action updates in classical unary domains require no focus operations - every formula evaluates to definite truth values since all the unary predicates are abstraction predicates. The only branches are caused due to the operations for drawing out action arguments. These operations use focus formulas constrained to be role-specific and uniquely satisfiable, and thus satisfy focus-classifiability (Prop. 7). This leads to the following theorem:

Theorem 6. (Classical unary domains are extended-LL domains) *All classical unary domains are extended-LL domains, and consequently, result in FC^3 domains under canonical abstraction.*

Many interesting problems can be translated into classical unary domains by creating an instance for every relation of arity k with all possible values for the other $k-1$ arguments: in the simplified transport domain introduced earlier in this section we constructed unary relations atL_i corresponding to the different possible locations. This process is similar to propositionalizing the relations, where the resulting vocabulary would have had a constant, (instead of a unary relation) for every relation and tuple of arity k (instead of tuples of arity $k-1$).

The limitation of this approach is that it does not allow generalization in the numbers of arguments which have been converted into relation instances (e.g. the locations in the transport example). Extended-LL domains are thus a strict generalization of classical unary domains, allowing us to represent problem domains like the delivery problem and blocks-world problems as described in Section 5.6 on page 83 and Section 6.1.3 on page 96.

5.5 Extended-LL Domains and Abacus Programs

We can now complete the development of methods for finding preconditions of plans with loops. Intuitively, generalized plans with extended-LL domain actions correspond to abacus programs with the same topological structure, permitting a direct application of the methods developed in the previous chapter.

5.5.1 Translation From Plans to Abacus Programs

We begin with a lemma establishing the relationship between extended-LL actions and abacus programs.

Lemma 2. *Let $S_1 \xrightarrow{a_1} S_2$ be an action operation in an extended-LL domain, where S_2 represents only one of the possible results of a focus operation conducted as a part of a_1.*

For all $s_1 \in \gamma(S_1)$, the effect of this operation on role-counts in s_1 can be represented as a linear abacus program Π_a whose registers represent role-counts in s_1. In other words, there is an abacus program Π_a such that for all $s_1 \in \gamma(S_1)$, $a_1(s_1) \in \gamma(S_2)$ iff the final node of Π_a is reachable starting with the initial role-counts (register values) of s_1.

Proof. The desired program Π_a will have a register for every role with an element in S_1 or S_2. The changes in role-counts due to any action a on an abstract structure S_0 in

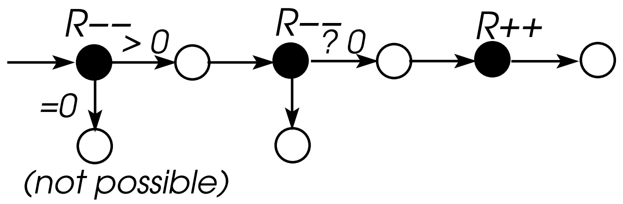


Figure 5.2: Linear abacus program segment for a decrementing action in an extended-LL domain.

an extended-LL domain can be listed as a set of increments: $\{(R_1^+, c_1), \ldots (R_k^+, c_k)\}$ and a set of roles decremented by 1 $\{R_1^-, \ldots R_m^-\}$.

Starting with an initial state-node for Π_a labeled S_0, we first add sequences of c_i abacus action nodes for each role R_i that needs to be incremented, with new intermediate state-nodes. Let the final state-node obtained after this operation be ns_I.

We then need to add abacus operations simulating role decrements. First, for each decrement to be conducted we identify the branch ($R_i^- > 1$ or $R_i^- = 1$) that was taken in the given action operation. This is possible because of the extended-LL domain assumption. Starting at ns_I, we add sequences of abacus operations corresponding to each decrement operation. Since abacus actions can only conduct comparisons with zero, each of these sequences consists of the following 3 actions as shown in the figure below: one decrement operation for the role, an extra decrement to conduct a comparison with zero instead of with 1, and finally, an increment operation to reverse the extra decrement (see Fig. 5.2).

Here, the comparison operation after the second decrement is the operation identified in the given extended-LL action. Chaining such role-decrementing sequences of operations one after the other, starting at ns_I gives us a linear abacus program; we set the unique halting state of this program to be the last state in the sequence. By construction, the role-counts of a state $s_1 \in \gamma(S_1)$ will lead to the halting state in Π_a iff the branch condition for $S_1 \xrightarrow{a_1} S_2$ are satisfied. □

A plan with extended-LL domain actions can therefore be converted into an abacus program without changing its structural complexity (its loop structure). This is formalized in the following result, where we assume, without loss of generality, that the initial node of a generalized plan represents a null action.

Theorem 7. *Let* Π *be a graph-based generalized plan whose edges are labeled with abstract structures and every segment in the graph of the form* $\xrightarrow{S_i} a_k \xrightarrow{S_j}$ *represents the transition* $S_i \xrightarrow{a_k} S_j$ *in an extended-LL domain with sensing actions. There exists an abacus program* Π_a *with the same loop structure as* Π*, such that any node* n *in* Π *is reachable from an initial state* s_i *iff a corresponding node* n_a *is reachable in* Π_a *starting with the initial register valuation corresponding to* s_i*'s role-counts.*

Proof. Consider the dual representation of Π_D where structures label nodes and actions label edges. The overall idea is to construct Π_a by converting each transition from Π_D into a linear abacus program segment as in Lemma 2.

In case a structure node has multiple outgoing edges, there are two possible cases:

1. There are only two outgoing edges, labeled with the same action, and these branches are role-count classifiable. In this case, we use the same initial decrementing segment (the first decrementing operation in Fig. 5.2) while converting the two transitions into an abacus program with a branch.

2. The outgoing transitions are not role-count classifiable and stem from a sensing action. In this case, we use the NSet action to obtain a non-deterministic abacus program.

The halting state of each such transition's abacus program segment becomes the initial state for subsequent transitions; the halt state of the abacus program can be treated as any node with out-degree zero.

Equivalence of reachability follows by an inductive proof of equivalence of path existence between a pair of nodes in Π and the corresponding nodes in Π_a, using Lemma 2 as the base case. □

Note: The second construction in this proof, which uses the NSet actions can also be used to translate non-deterministic generalized plans with nodes having overlapping outgoing-edge conditions into non-deterministic abacus programs.

In the rest of this thesis, we will only consider generalized plans which satisfy the requirements of Theorem 7. In this way, the union of structures labelling the outgoing edges from the initial, null-action node of a generalized plan represents the set of possible initial states on which the plan can be executed. Theorem 7 allows us to use the methods for computing preconditions of abacus programs in the previous chapter for finding preconditions of generalized plans in extended-LL domains.

5.5.2 Translation From Abacus Programs to Plans

A similar structure preserving translation can be used to translate abacus actions into sequences of extended-LL domain actions which use roles as registers. Note that in extended-LL domains, an action which increases a role-count also necessarily decrements some other role-count. Therefore, in order to simulate abacus actions that increment a register without a corresponding decrement, we add an extra role R_∞ from which all the action simulating increments can delete objects. Comparisons $R_\infty > 0$ are assumed to always succeed. Thus, we have:

Lemma 3. *Linear segments of abacus programs can be simulated by linear segments of programs in extended-LL domains and vice versa.*

Corollary 2. *Plans with extended-LL domain actions can simulate abacus programs without increasing the loop complexity and vice versa.*

Note however, that plans in extended-LL domains tend to be more compact since a single action can update many role-counts, with increments larger than 1.

Theorem 8. *Plans with extended-LL domain actions are Turing complete.*

Extended-LL domains thus represent a powerful class of planning domains. Their action operations, however, are fundamentally simple and can be analyzed along the lines developed in the previous sections.

The next section shows a range of problems which can be represented in extended-LL domains, and whose actions can be treated as abacus actions. As a result, preconditions and termination guarantees of a wide range of plans with loops in these domains can be computed very efficiently. We also demonstrate our approach on plans with complex loops created by non-deterministic sensing actions.

5.6 Example Plans and Preconditions

We implemented the algorithm for constructing preconditions for simple loops and order independent nested loops due to shortcuts, and applied it to various plans with loops that have been discussed in the literature.

In all of these plans, the changes in every possible execution of the simple loop with shortcuts on every role-count (or register) are all in the same direction. Using the notation of Eqs. (4.1-4.4), δ_x, or the partial negative change for a register in $loop_x$ is never more than $\Delta_j^{loop_x}$ for all loops. Therefore, Eqs. 4.1, 4.2, and the conditions labeled Sufficient (1) and (2) on page 55 are necessary and sufficient (see the discussion following Eqs. (4.3 & 4.4)). The precondition computation algorithm constructs these equations by symbolically evaluating each constraint using the loop change vectors Δ. These vectors in turn are calculated using methods discussed in Section 5.3.1.

Existing approaches solve different subsets of these problems, but almost uniformly without computing plan preconditions or termination guarantees. For nested loops, our implementation takes a node in a strongly connected component as an input and computes an appropriate start node. It then decomposes the component into independent simple loops and computes the preconditions.

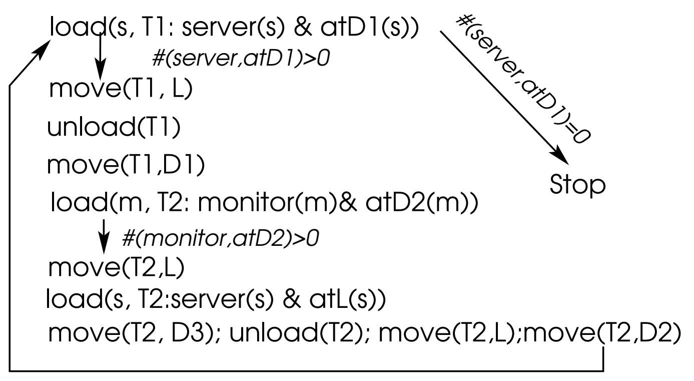

load(s, T1: server(s) & atD1(s))
#(server,atD1)>0
move(T1, L)
unload(T1)
move(T1,D1)
load(m, T2: monitor(m)& atD2(m))
#(monitor,atD2)>0
move(T2,L)
load(s, T2:server(s) & atL(s))
move(T2, D3); unload(T2); move(T2,L);move(T2,D2)
#(server,atD1)=0
Stop

Figure 5.3: Solution plan for the transport problem

The evaluation in this section focuses on our approach for computing preconditions of plans in abacus program representations; we present detailed representations of these problems and approaches for computing these generalized plans in the following chapters. Table 5.1 shows timing results for computing preconditions for 10 different plans.

Plan Representation Figs. 5.3, 5.4, and 5.5 show solution plans for some of the test problems. The default flow of control continues line by line (semi-colons are used as line-breaks). Edges are shown when an action may have multiple outcomes and are labeled with the conditions that must hold *prior* to action application for that edge to be taken (as with abacus programs). Only the edges required by the plan are drawn; the preconditions must ensure that these edges are always taken. For clarity, in some cases we label only one of the outcomes of an action, and the others are assumed to have the complement of that label. Actions are written as "ActionName(args:argument-formula(args))". Any object satisfying an action's argument formula may be chosen for executing the plan. The desired halt states are indicated with the action "Stop".

Transport In the transport problem (Srivastava, Immerman, and Zilberstein, 2008) two trucks have to deliver sets of packages through a "Y"-shaped map (shown in

Fig. 6.5 on page 100 together with the representational details for computing a generalized plan for this problem). Locations D1, D2 and D3 are present at the three terminal points of the Y; location L is at the intersection of its prongs. Initially, an unknown number of servers and monitors are present at D1 and D2 respectively; trucks T1 (capacity 1) and T2 (capacity 2) are also at D1 and D2 respectively. The goal is to deliver all objects to D3, but only in pairs with one of each kind.

The problem is modeled using the predicates *{server, monitor, atD$_i$, inT$_i$, atL, T1, T2}*. As discussed in the previous section, role-counts in this representation can be treated as register values and actions as abacus actions on these roles. The plan shown in Fig. 5.3 first moves a server from D1 to L using T1. T2 picks up a monitor at D2, moves to L, picks up the server left by T1 and transports both to D3. The first action, *load*, uses as its arguments an object s (satisfying $server(s) \land atD1(s)$), and the constant T1 representing the truck T1. It decrements the count of the role *{server, atD1}* and consequently has two outcomes depending on its value. Note that the second load action in the plan also has two outcomes, but only the one used in the plan is shown. In order to reach the Stop state with the goal condition, we require that final values of s_1 =#*{server, atD1}* and m_2 =#*{monitor, atD2}* be zero. Let s_3=#*{server, atD3}* and m_3=#*{monitor, atD3}*. The changes caused due to one iteration of the loop are $+1$ for m_3, s_3 and -1 for s_1, m_1. Using the method developed in Prop 2, the necessary and sufficient condition for reaching the goal after l iterations of the loop is that there should be equal numbers of objects of both types initially: $m_2^0 = l = s_1^0$.

Transport Conditional We made the transport problem conditional by introducing two non-deterministic aspects: objects left at L may get lost, and servers may be heavy, in which case the simple load action drops them and the forkLift action has to be used. Fig. 5.4 shows a plan solving this version of the problem. Extra branches have been added to the skeletal plan seen in Fig. 5.4 for handling non-deterministic action outcomes. The modified plan uses forkLift actions when needed; if a server is not

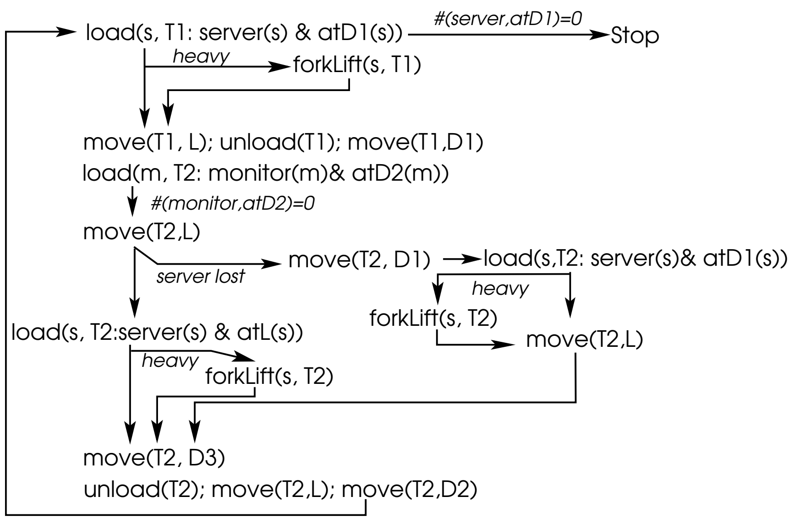

load(s, T1: server(s) & atD1(s))
#(server,atD1)=0
Stop
heavy
forkLift(s, T1)
move(T1, L); unload(T1); move(T1,D1)
load(m, T2: monitor(m)& atD2(m))
#(monitor,atD2)=0
move(T2,L)
server lost
move(T2, D1)
load(s,T2: server(s)& atD1(s))
heavy
forkLift(s, T2)
move(T2,L)
load(s, T2:server(s) & atL(s))
heavy
forkLift(s, T2)
move(T2, D3)
unload(T2); move(T2,L); move(T2,D2)

Figure 5.4: Solution plan for the conditional version of transport

found when T2 reaches L, the plan proceeds by moving T2 to D1, loading a server, and then proceeding to D3. Note that the shortcut for the “server lost" has a sub-branch, corresponding to the server being heavy. The plan can be decomposed into 8 simple loops. Of these, 4, which use the “server lost" branch use one extra server (loops $0, 5, 6$ and 7 in the inequality below). Let role-counts s_2, m_2, s_3, m_3 be as in the previous problem. Then, the obtained applicability conditions are:

$s_3^f = m_3^f = \sum_{i=0}^{7} k_i$

$m_2^f = m_2^0 - \sum_{i=0}^{7} k_i = 0$

$s_1^f = s_1^0 - \sum_{i=0}^{7} k_i - k_0 - k_5 - k_6 - k_7 = 0$

These conditions show that every possible loop decrements the role-counts s and m; however, in order to have all objects at D3 the conditions now require extra servers to be kept at D1, amounting to the number of times a server was lost.

Recycling In this problem a recycling agent must inspect a set of bins, and from each bin, collect paper and glass objects in their respective containers. The plan includes nested loops due to shortcuts (Fig. 5.5), with the start node at *PickObj*. *senseType* is a

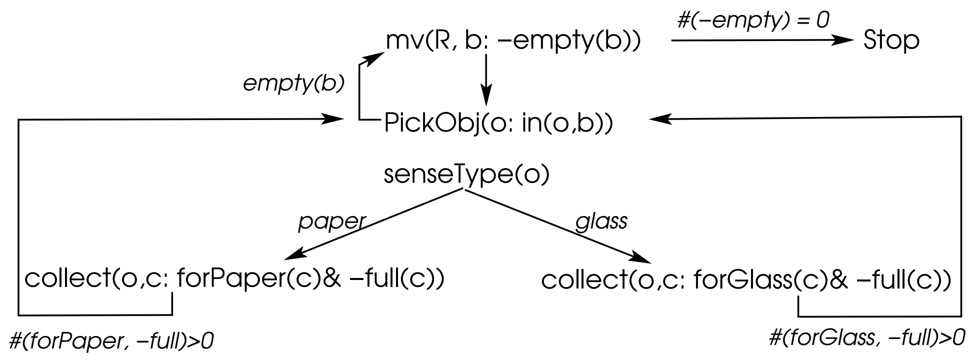


Figure 5.5: Solution plan for the recycling problem

sensing action, and the collect actions decrement the available capacity of each container, represented as the role-count of *{forX, ¬full}* where X is paper or glass. Let *e, fg, fp, p, g* denote the role-counts of non-empty bins, glass container capacity, paper container capacity, paper objects and glass objects respectively. Let l_1 denote the number of iterations of the topmost loop, l_2 of the paper loop and l_3 of the glass loop. The applicability conditions are:

$$e^f = e^0 - l_1 = 0, \quad fp^f = fp^0 - l_2 \geq 0, p^f = p^0 + l_2, \quad fg^f = fg^0 - l_3 \geq 0, \quad g^f = g^0 + l_3.$$

Note that the non-negativity constraints guarantee termination of all the loops.

Accumulator The accumulator problem (Levesque, 2005) consists of two accumulators and two actions: *incr_acc(i)* increments register i by one and *test_acc*(), tests if the given accumulator's value matches an input k. Given the goal $acc(2) = 2k-1$ where k is the input, KPLANNER computes the following plan: *incr_acc(1);* ***repeat*** *{incr_acc(1); incr_acc(2); incr_acc(2)}**until** test_acc(1); incr_acc(2)*. Although the plan is correct for all $k \geq 1$, KPLANNER can only determine that it will work for a user-provided range of values. This problem can be modeled directly using registers for accumulators and asserting the goal condition on the final values after l iterations of the loop (even though there are no decrement operations). We get

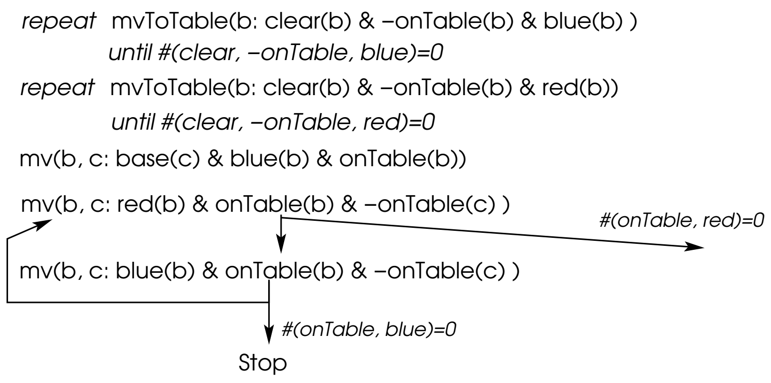
Problem	Time (s)	Problem	Time(s)
Accumulator	0.01	Prize-A(7)	0.02
Corner-A	0.00	Recycling	0.02
Diagonal	0.01	Striped Tower	0.02
Hall-A	0.01	Transport	0.01
Prize-A(5)	0.01	Transport (conditional)	0.06

Table 5.1: Timing results for computing preconditions

$$acc(1) = l + 1; \; acc(2) = 2l + 1 = 2k - 1.$$

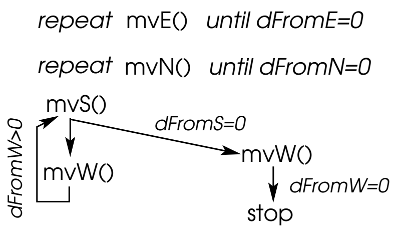
This implies that $l = k - 1 \geq 0$ iterations are required to reach the goal.

Further Test Problems and Discussion We tested these algorithms with many other plans with loops. Table 5.1 shows a summary of the timing results. The runs were conducted on a 2.5GHz AMD dual core system. Problems Hall-A, Prize-A(5) and Prize-A(7) (Bonet, Palacios, and Geffner, 2009) concern grid world navigation tasks. In Hall-A the agent must traverse a quadrilateral arrangement of corridors of rooms; the prize problems require a complete grid traversal of $5 \times n$ and $7 \times n$ grids, respectively. Note that at least one of the dimensions in the representation of each of these problems is taken to be *unknown* and *unbounded*. Our implementation computed correct preconditions for plans with simple loops for solving these problems. In Hall-A, for instance, it correctly determined that the numbers of rooms in each corridor can be arbitrary and independent of the other corridors. The Diagonal problem is a more general version of the Corner problem (Bonet, Palacios, and Geffner, 2009) where the agent must start at an unknown position in a rectangular grid, reach the north-east corner and then reach the southwest corner by repeatedly moving one step west and one step south. In this case, our method correctly determines that the grid must be square for the plan to succeed. In Striped Tower (Srivastava, Immerman, and Zilberstein, 2008), our approach correctly determines that an equal number of blocks of each color is needed in order to

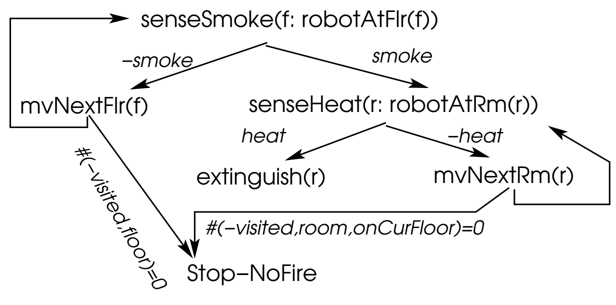


repeat mvToTable(b: clear(b) & −onTable(b) & blue(b))
until #(clear, −onTable, blue)=0
repeat mvToTable(b: clear(b) & −onTable(b) & red(b))
until #(clear, −onTable, red)=0
mv(b, c: base(c) & blue(b) & onTable(b))
mv(b, c: red(b) & onTable(b) & −onTable(c))
#(onTable, red)=0
mv(b, c: blue(b) & onTable(b) & −onTable(c))
#(onTable, blue)=0
Stop

(a) Alternating



(b) Diagonal



(c) Fire Fighting

```
repeat  mvE()
        until dFromE=0
repeat  mvN()
        until dFromN=0
repeat  mvW()
        until dFromW=0
repeat  mvS()
        until dFromS=0
```

(d) Hall-A

Figure 5.6: Generalized Plans for Test Problems

create a tower of blocks of alternating colors. In all the problems, termination of loops is guaranteed by non-negativity constraints such as those presented above.

5.7 Discussion

This chapter completes our development of methods for analyzing plans with loops by translating them into abacus programs. Results show that the algorithms developed are very efficient for extended-LL domains. The primary function of extended-LL domains is to capture representations where action branches on abstract states can be classified in terms of role-counts. The methods discussed in this and the previous chapter can however be applied more generally. In particular, action branches that cannot be classified in terms of role-counts can be considered to be non-deterministic from the point of view of precondition computation. The computed preconditions for the resulting plans may not be accurate (sufficient but not necessary), but may still provide useful information such as a guarantee of termination or an upper bound on the number of iterations until termination.

Identifying broader classes of applicability of these methods is a natural direction for future work. In particular, these methods can be applied in any situation where changes in certain quantities are sufficient to determine when a loop of actions will terminate. The idea of using role counts can be extended to the numbers of elements satisfying more general properties. These properties could be constructed using methods from description logic. Another direction for future work is to combine these methods with approaches for symbolic computation of preconditions of action sequences (Sanner and Boutilier, 2009).

CHAPTER 6

GENERALIZING SAMPLE PLANS

This chapter describes algorithms for generalizing sample plans using state abstraction. We first discuss an approach for generalizing a single concrete plan by adding choice actions and identifying loops in Section 6.1. In most generalized planning problems however, a single plan cannot visit and solve all the possible problem situations, even in the abstracted state space. However, merging plans with loops becomes difficult because each iteration of a loop may result in different states, with different paths to the goal. Section 6.2 presents an approach for addressing this problem using abstract representations of intermediate states in a loop of actions.

6.1 Generalizing a Single Plan

We present our approach for computing a generalized plan from a plan that works for a single problem instance in Algorithm 3. A preliminary version of this algorithm was described in (Srivastava, Immerman, and Zilberstein, 2008). The input to Alg. 3 is a concrete example plan $\pi = (a_1, a_2, \ldots, a_n)$ for a concrete state C_0. Let S_0 be an abstract structure embedding C_0. In order to be able to find preconditions, S_0 should be such that the space of structures reachable from it constitutes an extended-LL domain. In our experience, the canonical abstraction of C_0 suffices; if the space of reachable structures is not extended-LL, loops can still be found, but the procedure for finding preconditions may encounter a branch which is not focus classifiable, and therefore yield only necessary, but not sufficient preconditions.

Algorithm 3: ARANDA-Learn

```
  Input: $\pi = (a_1, \ldots, a_n)$: plan for $C_0$
  Output: Generalized plan $\Pi$
1 SASequence $\leftarrow$ Trace($C_0, \pi$)
2 loopSet$\leftarrow$ formLoops(SASequence)
3 $\Pi \leftarrow$ createGraph(SASequence, loopSet)
4 $S_f \leftarrow$ last structure in SASequence
5 $l_\Pi \leftarrow$ findPrecon($S_0, \Pi, \varphi_g$)
6 return $\Pi_r, l_\Pi$
```

The idea behind Alg. 3 is to apply a given concrete plan in the abstract state space, starting with an abstract start state (line 1). Because of abstraction, recurring properties become easily identifiable as repeating abstract states. Procedure *formLoops* uses these recurring identical structures to identify potential loops (line 2). *formLoops* returns a data structure representing all the loop positions and lengths; this is converted in a straightforward manner to a graph representation with nodes and edges by the subroutine *createGraph* (line 3).

If there is a constraint on the final abstract structure under which the goal formula is satisfied, then this is back propagated into a constraint on the initial structure in Π using methods described in Chapters 4 and 5. This is implemented in the *findPrecon* subroutine (line 5).

The methodology for *findPrecon* was discussed in Chapters 4 and 5. We now provide a description of the subroutines *Trace* (listed on pg. 93) and *formLoops* (listed on pg. 94).

6.1.1 Tracing

Procedure *Trace* takes as input, a concrete plan π and a concrete structure C_0 and returns a trace, or a sequence of abstract structures and actions (SASequence). In order to do so it first generalizes the choice actions in π (line 2). The generalized choice action selecting action a_i's k^{th} argument is specified using a formula capturing exactly the role of the element o_k chosen by the original choice action, in the preceding con-

```
Procedure Trace($C_0, \pi$)
1 $S_0 \leftarrow canon(C_0)$
2 $(a_1, \ldots, a_{n_c}) \leftarrow$ GeneralizeChoiceActions($\pi$)
3 for $i$ in $[1, \ldots, n_c]$ do
4     $C_i \leftarrow a_i(C_{i-1})$
5     $AbsStrucSet \leftarrow a_i(S_{i-1})$
6     for $S$ in $AbsStrucSet$ do
          if $C_i \sqsubseteq S$ then
7             $S_i \leftarrow S$
          end
      end
  end
8 return SASequence $\leftarrow (S_0, a_1), (S_1, a_2), \ldots (S_{n_c-1}, a_{n_c-1}), S_{n_c}$
```

crete state, C_{i-1}. The choice action is constructed as discussed in Section 3.4.3. The resulting sequence of actions is successively applied on concrete and abstract states, starting with C_0 and its canonical abstraction, S_0 (lines 3,4,5). After each action's application, the set of abstract structures obtained is traversed while searching for the one that embeds the corresponding concrete result (line 7). Since action updates on abstract structures capture all possible results, and the results of the focus operation are mutually inconsistent, exactly one such abstract structure will be found. This abstract structure becomes the next abstract structure in the trace, and the one on which the next action operator will be applied. Once all actions have been applied and all the abstract structures capturing the observed concrete results at each step have been obtained, a sequence of (abstract state, action) tuples is returned.

6.1.2 Identifying Loops

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

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

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

ment of SASequence that forms the loop; the loopExit index denotes the last element of SASequence which can be rolled into an iteration of this loop.

formLoops makes a single pass over the input sequence of abstract-state and instantiated action pairs while maintaining a look-up table, *Last*, for the last index where a particular (state, action) pair occurred. If the k^{th} element of SASequence matches its j^{th} element ($j < k$), then the index j is taken as the beginning of a loop (loopStart) and index k as its end (loopEnd). Such a repeated pair $(S_{j-1}, a_j) = (S_{k-1}, a_k)$ indicates that some properties that held in the concrete state after application of a_{j-1} were true again after application of a_{k-1} as witnessed by the fact that $S_{k-1} = S_{j-1}$, and further, that in the example plan, the same action $a_j = a_k$ was applied at this stage. This is our fundamental cue for identifying an unrolled loop – as long as an identical abstract state can be reached again, the same actions can be applied. The subroutine *safeLoop* returns True iff the loop makes a net non-zero change, determined using methods from chapters 4 and 5.

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

Example 6. *Consider the transport problem discussed in Section 5.1. Fig. 6.1 shows a concrete plan execution on this problem. By adding a choice action before the first* load *operation, and tracing out the plan on the canonical abstraction of the initial structure*

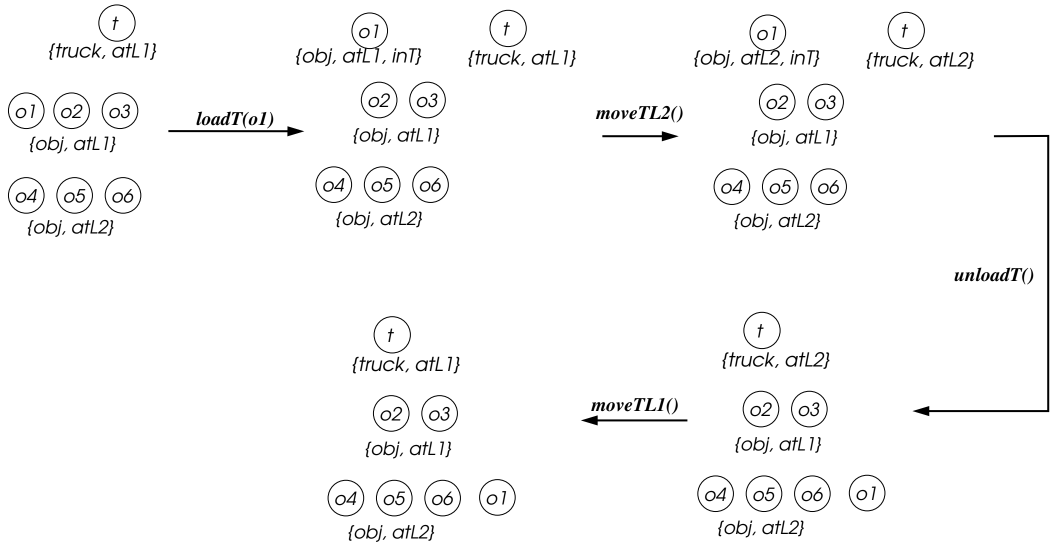


Figure 6.1: An example plan in the transport domain.

we get exactly the path shown in Fig. 5.1. The included loop can be identified using formLoops*, as described above.*

6.1.3 Implementation and Results

We implemented a prototype for ARANDA-Learn in Python, using TVLA as an engine for computing action results. We ran the prototype on some 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. Table 6.1 (pg. 109) shows a comprehensive summary of the preconditions for the generalized plans found for these problems, and the start structures on which they apply. The l_i variables in this table correspond to the number of iterations of the i^{th} loop in the generalized plan, where the numbering begins from the terminal node. Timing results for different phases of plan generalization, and for precondition evaluation are shown in Table 6.2 (pg. 110).

Each of the generalized plans described in this section can also be found directly by the plan synthesis approach (ARANDA-Synth) discussed in Chapter 7. In fact, ARANDA-

Synth can find complete generalized plans for these problems by finding the extra branches required to cover the smaller problem instances missed by the generalizations presented below.

Note on Problem Domains The first three problems described below are representative of many similar problems in the transport and blocks world domains. The Hall-A, Corner-A, Prize-A and GreenBlock problems are from work by Bonet, Palacios, and Geffner (2009), who designed these problems in a partially observable framework where observations are automatically triggered when the real states generating them are reached. This formulation thus does not require sensing actions. Although this is a very different formulation from ours, the problems remain meaningful and interesting in our setting as well. In most cases, the abstraction we used to represent the multiple possible initial states corresponded with the belief states used by Bonet et al. to reflect partial observability. In general, their solutions are much more compact than ours - this is expected, as we restrict our implementation to find only simple loops.

Delivery

We implemented the non-deterministic version of the delivery problem with a sensing action *findDest* for one truck. The action and vocabulary for this problem were defined in Section 3.1 on page 26 and the non-deterministic, sensing aspects were described in Section 3.5 on page 43. Because of the restriction to a single truck, the *Move* and *Load* actions requires only one argument representing the destination and the crate to be loaded respectively; the *Unload* action does not require arguments, and the predicate *in* becomes unary and holds for the object currently in the truck. The input example plan delivered five objects to two different locations. The abstract start structure is shown in Fig. 6.2. ARANDA-Learn found the generalized plan shown in Fig. 6.3. Since the delivery domain is an extended-LL domain, we can use the methods described in Chapter 5 to compute the preconditions for this plan as $\#(item) \geq 3$. In

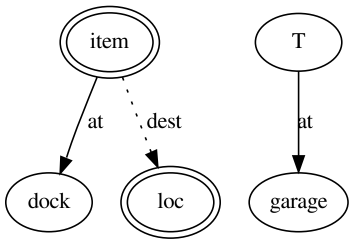


Figure 6.2: Abstract start structure for the Delivery problem

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., $\#(crate) = l + 3$, where $l \geq 0$ is the number of loop iterations).

Trucks

Vocabulary: $\{Monitor, Server, T1, T2, atL1, atL2, inT1, inT2\}$

Actions: $\{LoadT_i(x), UnloadT_i(), GoToL_jT_i()\}$

This is a problem from the transport domain. We have two source locations L1 and L2, which have a variable number of monitors and servers respectively (Fig. 6.5). There are two trucks, $T1$ at L1 and $T2$ at L2 with capacities 1 and 2 respectively. The generalized planning problem is to deliver *all* – regardless of the actual numbers – items to L4, but only in pairs with one item of each kind.

We represented this domain without using any binary relations, as a classical unary domain. Fig. 6.5 shows initial abstract structure used for tracing. The example plan for six pairs of such items worked as follows: $T1$ moved a monitor from L1 to L3 and returned to L1; $T2$ then took a server from L2 to L4, picking up the monitor left by $T1$ at L3 on the way. Fig. 6.4 shows the main loop discovered by our algorithm. The computed preconditions for the final generalized plan are shown in Table 6.1, and constrain the counts of servers and monitors to be equal, and at least 2.

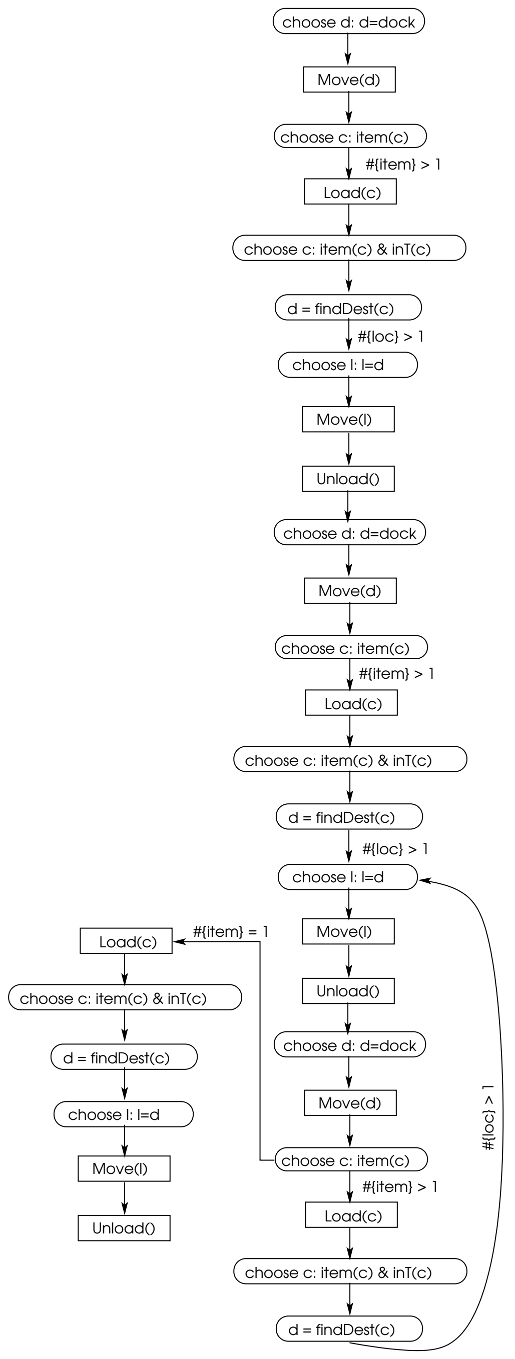


Figure 6.3: Generalized plan for unit delivery problem instances with at least 2 crates.

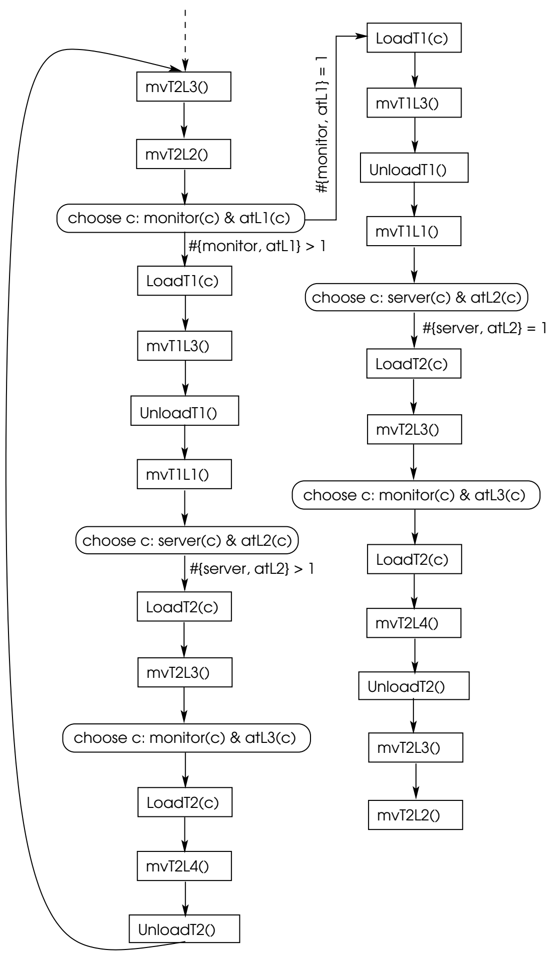


Figure 6.4: Main loop for Trucks

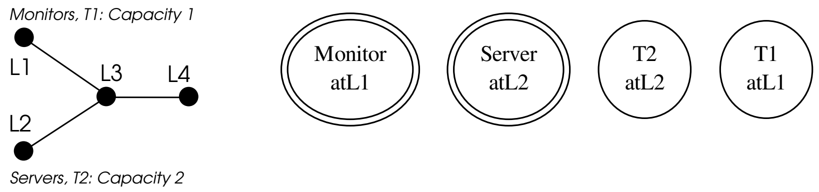


Figure 6.5: Map and the start structure for Trucks

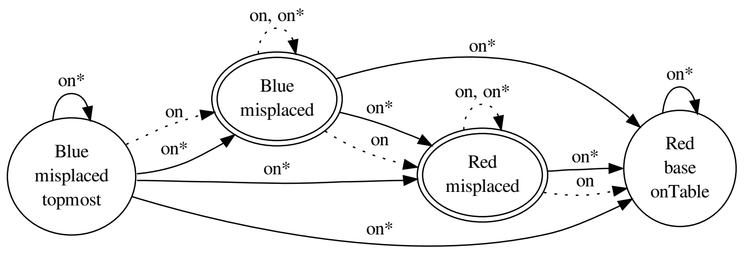


Figure 6.6: Abstract start structure for striped block tower

Striped Block Tower

Vocabulary: $\{Red^1, Blue^1, base^1, onTable^1, on^2, topmost^1, on^{*2}, misplaced^1\}$

Actions: $\{Move(x, y), moveToTable(x)\}$

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

Fig. 6.6 shows the abstract initial structure. The *misplaced* predicate is used to determine if the goal is reached. *misplaced* holds for a block iff either it is on a block of the same color, or above a block which is on a block of its own color.

The input example plan worked for six pairs of blocks, by first unstacking the whole tower, and then placing blocks of alternating colors back above the base block.

Our algorithm discovered three loops: unstack red, unstack blue, stack blue and red (Fig. 6.7). The preconditions shown in Table 6.1 describe possible role-counts at the start structure. These conditions capture a more general situation where the start structure may have some blocks on the table, corresponding to the roles $\{Red, misplaced, onTable, topmost\}$ and $\{Blue, misplaced, onTable, topmost\}$. If we set these quantities as zeros, we get $l_1 = l_3$ and $l_1 = l_2 + 1$, which constrain the number of red and blue blocks in the initial stack to be equal. Further, the number of blue blocks should be $3 + l_2 + 1$, counting the blocks with roles $\{Blue, misplaced\}$ and one extra block with the role $\{Blue, misplaced, topmost\}$.

Green Block

Vocabulary: $\{topmost^1, onTable^1, on^2, on{*}^2, isGreen^2\}$

Actions: $\{unstack(), senseColor(x), collect(), discard()\}$

The Green Block problem is to find a green block in a stack of blocks. We formulate this problem using a sensing action to determine the color of the topmost block (cf. note on problems above). Fig. 6.8 shows the abstract initial structure. We use the *isGreen(arg,x)* predicate in order to implement the sensing action for block x's color using the focus operation.

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

This problem domain does not belong to the extended-LL class because its goal depends on a sensing action whose result cannot be predicted based on role-counts alone. However, our methods for plan generalization still compute a deterministic generalized plan which can be proved to work for almost all possible problem instances

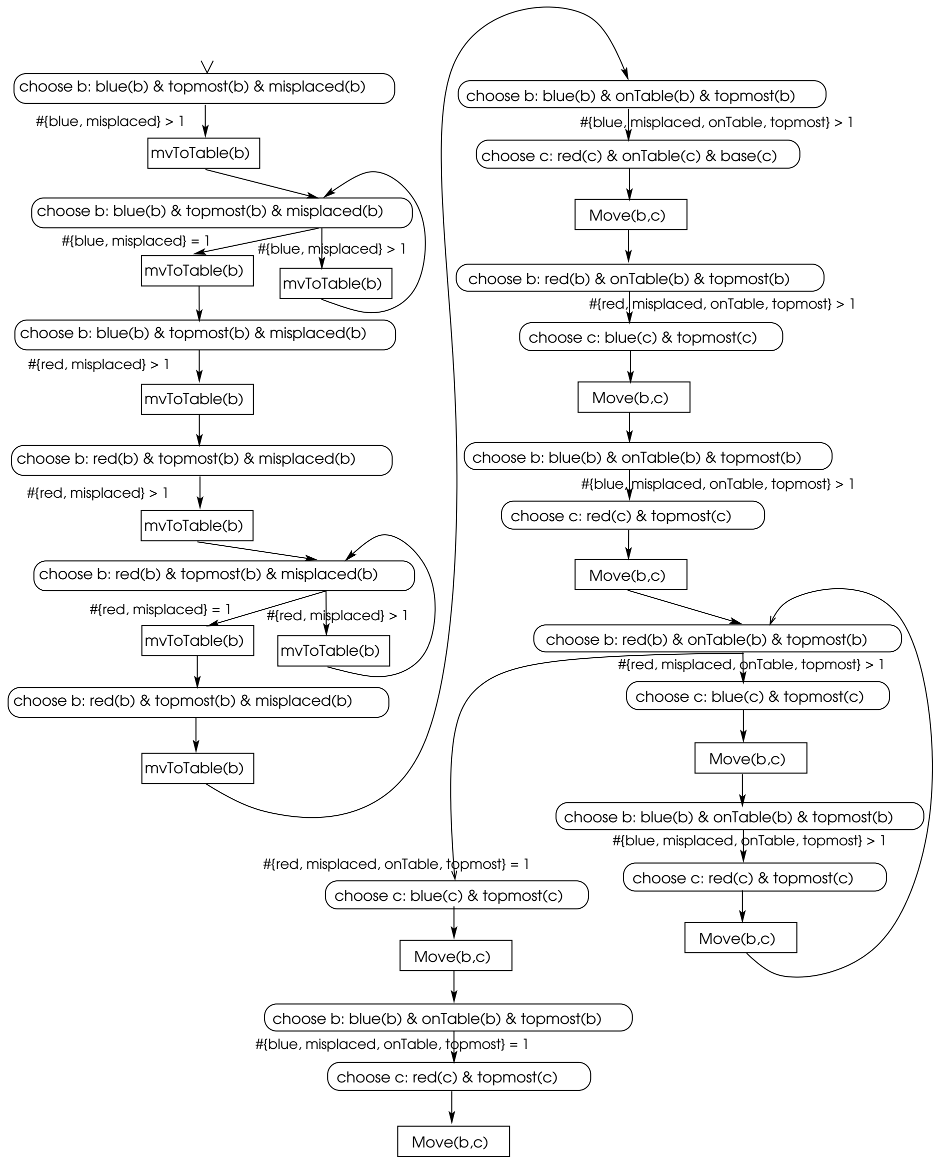

choose b: blue(b) & topmost(b) & misplaced(b)
#{blue, misplaced} > 1
mvToTable(b)
choose b: blue(b) & topmost(b) & misplaced(b)
#{blue, misplaced} = 1
#{blue, misplaced} > 1
mvToTable(b)
mvToTable(b)
choose b: blue(b) & topmost(b) & misplaced(b)
#{red, misplaced} > 1
mvToTable(b)
choose b: red(b) & topmost(b) & misplaced(b)
#{red, misplaced} > 1
mvToTable(b)
choose b: red(b) & topmost(b) & misplaced(b)
#{red, misplaced} = 1
#{red, misplaced} > 1
mvToTable(b)
mvToTable(b)
choose b: red(b) & topmost(b) & misplaced(b)
mvToTable(b)
choose b: blue(b) & onTable(b) & topmost(b)
#{blue, misplaced, onTable, topmost} > 1
choose c: red(c) & onTable(c) & base(c)
Move(b,c)
choose b: red(b) & onTable(b) & topmost(b)
#{red, misplaced, onTable, topmost} > 1
choose c: blue(c) & topmost(c)
Move(b,c)
choose b: blue(b) & onTable(b) & topmost(b)
#{blue, misplaced, onTable, topmost} > 1
choose c: red(c) & topmost(c)
Move(b,c)
choose b: red(b) & onTable(b) & topmost(b)
#{red, misplaced, onTable, topmost} > 1
choose c: blue(c) & topmost(c)
Move(b,c)
choose b: blue(b) & onTable(b) & topmost(b)
#{blue, misplaced, onTable, topmost} > 1
choose c: red(c) & topmost(c)
Move(b,c)
#{red, misplaced, onTable, topmost} = 1
choose c: blue(c) & topmost(c)
Move(b,c)
choose b: blue(b) & onTable(b) & topmost(b)
#{blue, misplaced, onTable, topmost} = 1
choose c: red(c) & topmost(c)
Move(b,c)

Figure 6.7: Generalized Plan for Striped Block Tower. In choice actions, only the predicates belonging to the role being chosen are shown.

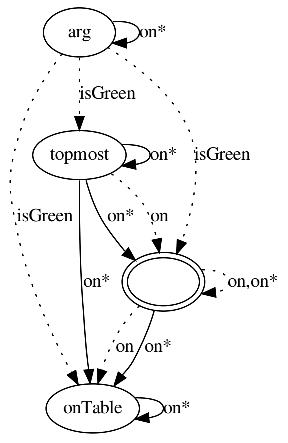


Figure 6.8: Initial abstract structure for the green block problem.

(whenever except when the number of blocks is more than 4; see the preconditions on page 109).

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

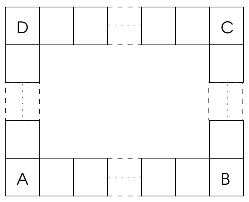
The learned generalized plan is thus correct and deterministic, but its preconditions are not expressible in terms of the counts of available roles. Because of this, the preconditions we obtain are necessary, but not sufficient.

Hall-A

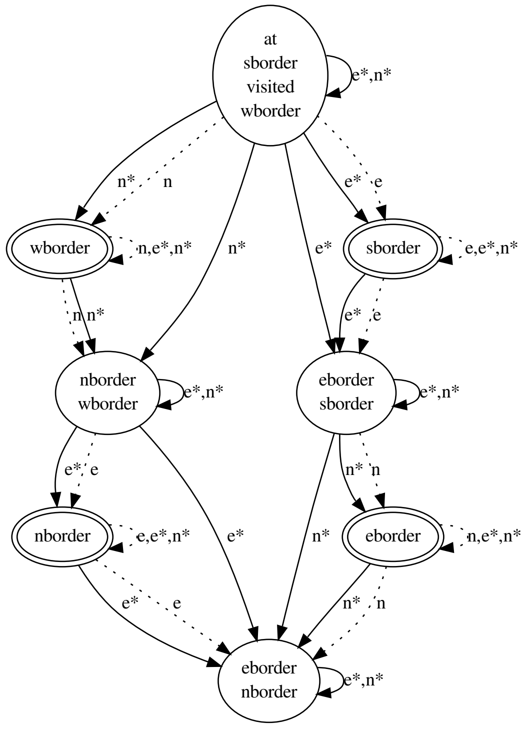
Vocabulary: $\{e^2, n^2, e^{*2}, n^{*2}, wborder^1, nborder^1, eborder^1, nborder^1, visited^1\}$

Actions: $\{mvE(), mvW(), mvN(), mvS()\}$

The hall-A problem was designed by Bonet, Palacios, and Geffner (2009) for evaluating methods for computing looping finite-state controllers for solving contingent planning problems. It consists of 4 corridors arranged to form a quadrilateral (Fig. 6.9a). Each corridor consists of multiple adjacent rooms which have to be traversed to cross



(a) A hall of unknown dimensions



(b) Initial abstract structure

Figure 6.9: Representation of the Hall-A problem

the hall; the problem is to find a plan for visiting all four corners and returning to the starting point, for an agent starting at a given corner.

The *e* and *n* relations represent the east and north relations between rooms and the e^* and n^* relations, their corresponding transitive closures. The *visited* relation is used to determine the goal condition and $x - border$ predicates define different segments of the quadrilateral corridor system. The smallest example plan from which our approach could identify loops solved this problem for a square arrangement with 7 rooms in each hall. The canonical abstraction of this initial state, used as the initial abstract structure during tracing, is shown in Fig. 6.9b.

The example plan traversed the halls in a square arrangement with 7 rooms along each wing. The generalized plan consists of four loops. The fact that the numbers of iterations of each of these loops do not have to match is captured by the preconditions. The generalized plan can therefore solve any trapezoidal arrangement of halls with at least 6 rooms in each wing.

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

Representing Grid-world Problems

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

The vocabulary and actions for each of the following problems therefore, are:

Vocabulary: $\{dFromE, dFromW, dFromN, dFromS\}$

Actions: $\{mvE(), mvW(), mvN(), mvS()\}$

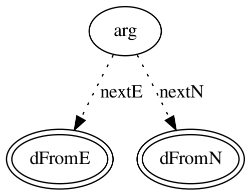


Figure 6.10: Initial abstract structure for the Prize-A and Diagonal Return problems.

Prize-A

In Prize-A, the agent must completely traverse all the squares of a given rectangular grid, starting at a given corner. The abstract start structure is shown in Fig. 6.10.

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

Utilizing example plans that traversed the grid row-wise, our approach easily scales to grids with higher numbers of rows. We present timing results with 5 and 7 row grids in Table 6.2. Note that a complete general solution to a grid with n rows is quadratic in n, and consequently cannot be learned from such example plans because of our restriction to generalized plans with simple loops. The obtained generalized plans have a different simple loop for each row in the grid.

The preconditions constrain the number of iterations of all but the last loop to be equal; as in the blocks problem, they are more general than the initial abstract structure in Fig. 6.10 and allow the starting location to be at a distance from the West corner. Consequently, the number of iterations of every loop other than the first eastward traversal are constrained to be equal, restricting the plan to square grids. Further, if $\#\{dFromW\}$ is set to zero, denoting a start at the southwest corner, the number of iterations of the first loop (l_6) also get constrained to be equal to the others.

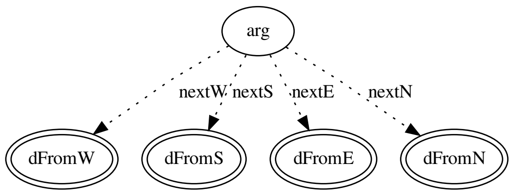


Figure 6.11: Initial abstract structure for the corner problem.

Corner-A

In the Corner-A problem, the agent must reach the top right corner of the grid. The start structure for this problem is shown in Fig. 6.11.

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

Diagonal Return

In this problem, the agent must start at the south-west corner, reach the north-east corner and return along the diagonal. This problem was not included in the set of problems used by Bonet et al; we use it to illustrate how relationships between iterations of different loops can be captured effectively. The abstract start structure for this problem is identical to the one for Prize-A (Fig. 6.10). The example plan first moved the agent as far east as possible, then moved north to the north-east corner before returning along the diagonal grid squares using alternating *mvS()* and *mvW()* actions.

The generalized plan captures the three loops of actions and its preconditions are shown in Table 6.1. These preconditions are again much more general than the provided initial abstract structure. They show that the abstract goal structure with the agent at the south-west corner will be achieved by the generalized plan as long as the grid is square. More precisely, the obtained preconditions constrain the problem instances to those where $\#\{dFromE\} + \#\{dFromW\} = \#\{dFromN\} + \#\{dFromS\}$ (this can be obtained explicitly by solving the presented preconditions for l_1).

Summary of Timing Results

Timing results for all the test problems are shown in Table 6.2. While the results show good scalability, many engineering optimizations are possible on our prototype implementation of the presented algorithms. Our implementation is written in Python, which is an interpreted language. Faster results can be obtained from an implementation in a compiled language. Profiler outputs show that most of the time is spent in calls to TVLA and in Python's module for adding or removing edges from graphs that we use to implement logical structures. Optimization of these data structures can also improve the running times.

While the entire process of tracing can be understood as contributing to the information utilized for computing preconditions, the actual time required for computing preconditions from the obtained generalized plan is negligible. In conclusion, this approach is very scalable, and provides an efficient method for finding generalized plans with preconditions.

Discussion and Evaluation of the Results

Table 6.3 shows an evaluation of the generalized plans found by ARANDA-Learn, and described in the previous section. Testing for applicability requires only counts of elements of different roles in the start structure. Table 6.3 lists this cost as $O(n)$ but it can be reduced to a constant number of numeric comparison operations if these counts are provided with the initial concrete state.

Plan instantiation cost is always $O(n)$ because we find plans with simple loops, all of which reduce the count of some role(s) and thus can be iterated at most $O(n)$ times.

Problem	Start Structure	Preconditions on Start Structure
Delivery	Fig. 6.2	$\#\{item\} = 3 + l_1; \#\{loc\} > 1$
Trucks	Fig. 6.5	$\#\{monitor, atL2\} = 2 + l_1;$ $\#\{server, atL1\} = 2 + l_1$
Blocks	Fig. 6.6	$\#\{Blue, misplaced\} = 2 + l_3$ $\#\{Red, misplaced\} = 3 + l_2$ $\#\{Red, misplaced, onTable, topmost\} = -1 + l_1 - l_2$ $\#\{Blue, misplaced, onTable, topmost\} = l_1 - l_3$
Green Block	Fig. 6.8	$\#\{\} = 2 + l_1$
Hall-A	Fig. 6.9b	$\#\{wborder\} = 4+l_1$; $\#\{nborder\} = 4+l_2$ $\#\{eborder\} = 4+l_3$; $\#\{sborder\} = 4+l_4$
Prize-A(5 rows)	Fig. 6.10	$l_1 = l_2 = l_3 = l_4 = l_5; \#\{dFromE\} = 3 + l_6$ $\#\{dFromW\} = l_5 - l_6; \#\{dFromN\} = 5$
Prize-A(7 rows)	Fig. 6.10	$l_1 = l_2 = l_3 = l_4 = l_5 = l_6 = l_7; \#\{dFromE\} = 3 + l_8$ $\#\{dFromW\} = l_7 - l_8; \#\{dFromN\} = 7$
Corner-A	Fig. 6.11	$\#\{dFromN\} = 2+l_1; \#\{dFromE\} = 3+l_2$
Diagonal Return	Fig. 6.10	$\#\{dFromE\} = 3 + l_3; \#\{dFromS\} = l_1 - l_2;$ $\#\{dFromW\} = l_1 - l_3; \#\{dFromN\} = 3 + l_2$

Table 6.1: Preconditions for example problems. l_i denote the number of iterations of loop i in the corresponding plan; preconditions for the Green Block problem are necessary, but not sufficient.

Problem	Tracing	Loop Finding	Computing Preconditions	Total
Delivery	66.12	3.93	0.01	70.05
Trucks	85.79	2.94	0.01	88.74
Blocks	65.01	2.04	0.02	67.06
Green Block	17.04	0.52	0.00	17.56
Hall-A	32.30	1.89	0.01	34.19
Prize-A(5 rows)	35.73	0.51	0.01	36.24
Prize-A(7 rows)	47.93	0.79	0.02	48.73
Corner-A	6.94	0.04	0.00	6.98
Diagonal Return	20.89	0.22	0.01	21.12

Table 6.2: Timing results for ARANDA-Learn. All results in seconds; runs were carried out on a Linux machine with an Intel Core2 Duo 1.6GHz processor and 1.5GB RAM.

Each iteration has a constant number of choice operations, and each of these can be executed in constant time by maintaining look-up tables containing elements of each role, as a part of the action updates.

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

6.2 Merging Multiple Plans

In the previous section we presented an approach for finding generalized plans by identifying sequences of actions which could be converted into useful loops making progress towards the goal. However, as shown in Table 6.1, a single plan may not be sufficient to cover all the possible results of actions on abstract structures. This problem is more pronounced in the presence of partial observability, where sensing actions may

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

Table 6.3: Evaluation of some generalized plans. n denotes the size of the problem instance and l, k are variables ≥ 0. See Sec. 6.1.3 for discussion of optimality of the Trucks solution.

be required to discern certain state properties during plan execution. Plans that use sensing actions need to be able to solve every possible outcome of these actions. The input plans in these scenarios may only encounter one of the possible results of each application of a sensing action. Our objective is to be able to compose different such example plans into a coherent, generalized plan. In doing so, we need to be able to work with uncertainties in both object quantities and properties.

Let us consider the following example of a contingent planning problem with unknown quantities of objects: a recycling robot must pick up objects from a set of bins, perform a sensing action to determine recyclability of the drawn object, and store it in an appropriate container.

Example 7. *The recycling problem can be modeled using the following vocabulary:* $\mathscr{V} = \{\text{bin}^1, \text{visited}^1, \text{object}^1, \text{collected}^1, \text{empty}^1, \text{container}^1, \text{forPaper}^1, \text{forGlass}^1, \text{in}^2, \text{isPaper}^1, \text{isGlass}^1, \text{robotAt}^1\}$.

An example structure, S, can be described as follows: the universe, $|S| = \{b, o, c_1, c_2\}$, $\text{bin}^S = \{b\}$, $\text{object}^S = \{o\}$, $\text{container}^S = \{c_1, c_2\}$, $\text{forPaper}^S = \{c_1\}$, $\text{forGlass}^S = \{c_2\}$, $\text{in}^S = \{(o, b)\}$, $\text{isPaper}^S = \{o\}$, $\text{robotAt}^S = \{b\}$, $\text{visited}^S = \{b\}$. *We omit the predicates not satisfied by any tuples.*

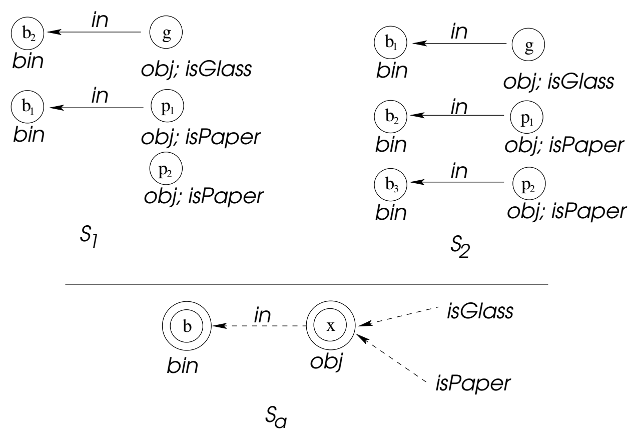


Figure 6.12: Representing belief states in the recycling problem using state abstraction.

Integrity constraints for the recycling domain would include among others the formulas $\forall uvw(\text{in}(u,v) \wedge \text{in}(u,w) \rightarrow (v = w \wedge (\text{bin}(v) \vee \text{container}(v))))$ *meaning that each object can be in at most one bin or container, and* $\forall u(\text{object}(u) \rightarrow (\text{isGlass}(u) \leftrightarrow \neg\text{isPaper}(u)))$ *meaning that objects are either of type paper or of type glass.*

To keep the presentation simple, we assume here that no bin contains more than one object. The goal condition is that all bins are empty: $\forall x(\text{bin}(x) \rightarrow \text{empty}(x))$*. The precondition and updates for the action* collect(o, c) *are:*

$$\begin{array}{rcl}
(\text{isGlass}(o) & \leftrightarrow & \text{forGlass}(c)) \wedge \text{container}(c) \wedge \\
 & & \exists b(\text{bin}(b) \wedge \text{in}(o,b) \wedge \text{robotAt}(b)) \\
in'(u,v) & := & (in(u,v) \wedge u \neq o) \vee \\
 & & (\neg in(u,v) \wedge u = o \wedge v = c) \\
empty'(u) & := & empty(u) \vee in(o,u) \\
collected'(u) & := & collected(u) \vee o = u
\end{array}$$

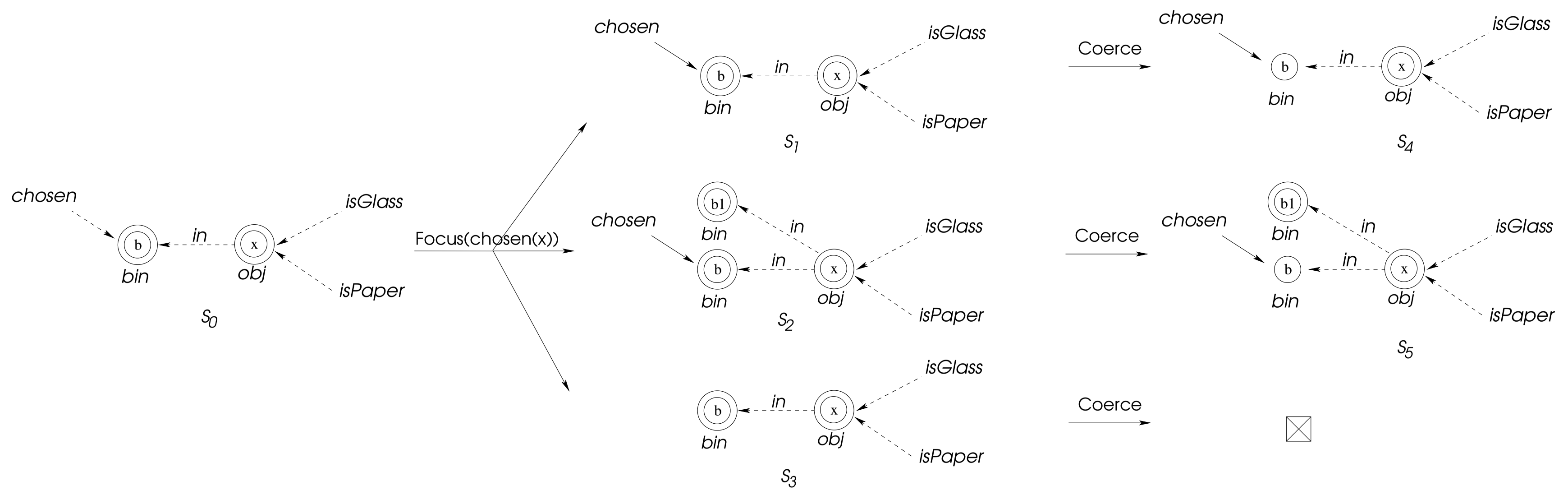


Figure 6.13: Choosing an action argument in an abstract state in the recycling problem.

6.2.1 Observation Model and Sensing Actions

Contingent plans deal with uncertainty about predicates in the agent's belief state using *observation* or *sensing actions* (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005). We model sensing actions as focus operations w.r.t the respective formulas being sensed. As discussed in Sec 3.4.1, the focus operation on an abstract state returns a set of more precise belief states corresponding to the different possible definite truth values of the formula being focused on.

For instance, the recycling domain has only one sensing action applicable to a chosen bin marked with the new (not in the domain's vocabulary) abstraction predicate *chosen*: *senseType()*, with the focus formula $\exists x(chosen(x) \wedge in(o, x) \wedge isPaper(o))$. When applied to an abstract structure (such as S_4 or S_5 in Fig. 6.13), it returns structures with different possible types of a single object in the chosen bin. Note that the integrity constraint that each object has a unique type makes either of the predicates *isPaper*, *isGlass* sufficient for sensing an object's type.

In addition to uncertainty about predicates, the agent does not have precise information about object quantities. We only require that it has sufficient knowledge to determine whether there are zero, exactly one, or more than one objects of each role at any step.

The contingent planning problem Our objective in dealing with partial observability continues to be that of generalized planning: given a set of domain-specific actions, integrity constraints, a goal formula, and an initial belief state S_{init}, our objective is to find a generalized plan solving the initial belief state S_{init}.

6.2.2 The Branch and Merge Algorithm

The most significant challenge faced by approaches combining multiple example plans is to determine positions in an existing plan where segments of a new example plan would be useful. This becomes more difficult when the existing plan contains

Algorithm 6: Branch and Merge

```
   Input: Existing plan $\Pi$, $\pi = (a_1, \ldots, a_n), S_0^\#$
   Output: Extended version of $\Pi$
1  $bp_\pi, bp_t \leftarrow 0$
3  t$\leftarrow$ Trace($\pi, S_0^\#$) $\mathrm{mp}_\Pi, \mathrm{mp}_t \leftarrow$ findMergePoint($\Pi$, t, $\mathrm{bp}_\Pi$, $\mathrm{bp}_t$)
4   repeat
5      if $mp_\Pi$ found then
6          $\mathrm{bp}_\Pi, \mathrm{bp}_t \leftarrow$ findBranchPoint($\Pi$, t, $\mathrm{mp}_\Pi$, $\mathrm{mp}_t$)
       end
7      if $bp_\Pi$ found then
8          $\mathrm{mp}_\Pi, \mathrm{mp}_t \leftarrow$ findMergePoint($\Pi$, $t$, $\mathrm{bp}_\Pi$, $\mathrm{bp}_t$)
9          addEdges($\Pi$, $t$, $bp_t$, $mp_t$, $mp_\Pi$, $bp_\Pi$)
       end
   until new $bp_\Pi$ or $mp_\Pi$ not found
   if $bp_\pi$ found and $mp_\pi$ not found then
       /* A terminal segment of t was not merged                    */
10     remainderT $\leftarrow$ path added to $\Pi$ after $bp_\Pi$
       /* Try to create loops in remainderT                          */
11     formLoops(remainderT)
   end
12 return $\Pi$
```

loops. *BranchAndMerge* (Alg. 6) is a greedy algorithm for addressing this problem. It uses abstract structures in plan traces as a compact representation of the infinitely many situations where the subsequent sequence of actions would be useful. The input to Alg. 6 is the existing plan (initially $\emptyset$), a new linear example plan and a concrete structure solved by the example plan.

Such example plans can be provided from prior experience. Given an abstract structure S_0 representing the initial belief state, they can be also generated by existing *classical* planners as follows: (a) create a concrete member state $S_0^\# \in \gamma(S_0)$ with specific truth values for the unobservable predicates. The number of universe elements in $S_0^\#$ corresponding to a summary element in S_0 can vary; we used a heuristic process to add at least six elements in $S_0^\#$ for every summary element in S_0. (b) make the appropriate sensing actions for the unobservable predicates as prerequisites for actions

Algorithm 7: Generalizing and merging examples

```
Input: $S_{init}$, the initial belief state
Output: Plan $\Pi$
$\Pi \leftarrow \emptyset$; looseEnds $\leftarrow S_{init}$
while looseEnds $\neq \emptyset$ do
    Remove $S_0 \in$ looseEnds
    $S_0^{\#} \leftarrow$ concrete instance of $S_0$
    $\pi_0 \leftarrow$ invokeClassicalPlanner($S_0^{\#}$)
    Merge($\Pi, \pi_0, S_0^{\#}$)
    looseEnds $\leftarrow$ getLooseEnds($\Pi$)
end
return $\Pi$
```

which use those predicates (c) solve this problem instance using a classical planner like FF (Hoffmann and Nebel, 2001).

In the recycling problem, the input to a classical planner can be a problem instance with multiple non-empty bins where each object's type is "paper". The collect action's formulation will require a predicate "sensed" to hold for the object being collected. The sensed predicate on the other hand will only be set by a "senseType" action with no other effect. This problem's solution plan will use "senseType" actions, but will only solve the problem for "paper" objects.

BranchAndMerge proceeds as follows. The example plan is first generalized (line 2) using the *Trace* subroutine discussed in Section 6.1.1 on page 92; we provide a brief overview of this process for readability. The input to *Trace* is a pair $(\pi, S_0^{\#})$, where $\pi = (a_1, \ldots, a_n)$ is a solution plan for the concrete structure $S_0^{\#}$. Plan π is generalized by replacing the action a_i's arguments by their roles in the concrete structure $S_{i-1}^{\#}$ ($S_i^{\#} = a_i(S_{i-1}^{\#})$, $i = 1, \ldots, n$) and including the automatically generated (Sec. 3.4) focus formulas. This results in a modified linear plan applicable in the abstract state space, say π'. The sequence of intermediate concrete states is then generalized by applying π' on the canonical abstraction S_0 of $S_0^{\#}$, and keeping only those results $S_i = a_i'(S_{i-1})$ which are consistent with the $S_i^{\#}$. This results in an interleaved sequence of structures and actions because only one of the results of the focus operation can be consistent

Object/Method	Description
`Trace`	integer indexed list of `Transitions`
`Transition`	tuple of `(init structure, action, final structure)`
`Transition` Methods	`initStruc(), action(), finalStruc()` – return corresponding elements
`GeneralizedPlan`	Graph with `Nodes` and `Edges`
`Node`	labeled with structure. `node.struc()` returns the labelled structure
`Edge`	tuple of `(node1, node2, action)`
`Edge` Methods	`n1(), n2(), action()` return start node, end node and action name respectively

Table 6.4: Data structures used in Algorithms 8 and 9

with a concrete state. Structures which are not consistent with the result seen in π at the same step represent possible situations that were *not* handled by π. These abstract structures can be indexed and stored in a list of “looseEnds” if suggestions for further examples are needed or in a hybrid implementation (Alg. 7).

Given an example trace t and an existing plan Π, *BranchAndMerge* uses [in lines 3 & 8] *findMergePoint* (Alg. 8) to find the index of the earliest structure in t that is embeddable in a structure in Π. If successful, *findMergePoint* returns mp_Π and mp_t, the node in Π and the index in t corresponding to these structures. A successful search indicates that the new example encountered an instance of a belief state present in Π. However, the subsequent actions in t may not be different from those following mp_Π in Π, or may not handle any new problem instances in addition to those already handled by Π. In order to minimize the new edges added to Π, after finding the merge points, Alg. 6 conducts a search for a branch point using the procedure *findBranchPoint* (Alg. 9).

findBranchPoint simultaneously traverses the actions of t and Π starting from the last known merge points mp_t and mp_Π, and returns the last node and index where the example trace matched the plan Π. More precisely, starting at the previous merge

Algorithm 8: findMergePoint(GeneralizedPlan Π, Trace t, Node bp_Π, integer bp_t)

```
1  $i \leftarrow bp_t$
2  while $i < length(t)$ do
3      egStruc ← t[$i$].finalStruc()
4      for node in $bp_\Pi$'s loop do
           /* traverse nodes in order of addition to $\Pi$          */
5          if egStruc $\sqsubseteq$ node.struc() and resulting loop will terminate then
6              return node, $i$
           end
       end
7      $i \leftarrow i + 1$
   end
   /* Merge point not found in nodes of $bp_\Pi$'s loop             */
   /* Conduct a search over all non-ancestors of $bp_\Pi$           */
8  candidates ← nonAncestorNodes($\Pi$, $bp_\Pi$)
   /* candidates are ordered by non-decreasing distance from $bp_\Pi$
      */
9  $i \leftarrow bp_t$
10 while $i < length(t)$ do
11     egStruc = t[$i$].finalStruc()
12     for node in candidates do
13         if egStruc $\sqsubseteq$ node.struc() then
14             return node, $i$
           end
       end
15     $i \leftarrow i + 1$
   end
16 return NULL, NULL
```

Algorithm 9: findBranchPoint(GeneralizedPlan Π, Trace t, Node mp_Π, integer mp_t)

```
1  $i \leftarrow mp_t$
2  gpNode $\leftarrow mp_\Pi$
3  while $gpNode \neq NULL$ and $i < length(t)$ do
       /* Match edge-actions and node-structures in Π              */
       /* with actions and structures of transitions in $t$        */
4      egStruc $\leftarrow$ t[$i$].finalStruc()
5      egAction $\leftarrow$ t[$i$].action()
6      embeddingFound $\leftarrow$ False
7      for $e$ in outEdges(gpNode) do
           /* Search the list of outgoing edges from gpNode          */
           /* for one that can embed egAction and egStruc            */
8          GPStruc $\leftarrow$ e.n2().struc()
9          GPAction $\leftarrow$ e.action()
10         if $GPAction = egAction$ and $egStruc \sqsubseteq GPStruc$ then
               /* an edge from gpNode with matching action           */
               /* and subsuming result node is found.  Move to the
                  next                                               */
               /* gpNode in Π and transition in $t$                  */
11             gpNode $\leftarrow$ e.n2()
12             $i \leftarrow i + 1$
13             embeddingFound $\leftarrow$ True
14             break /* End the for loop                             */
           end
       end
15     if (not embeddingFound) {return gpNode, $i$}
   end
16 return NULL, NULL /* No branch point found                        */
```

points mp_t, mp_Π it matches successive elements of t with action edges and structure nodes in Π until it finds a node bp_Π in Π and an index bp_t for a structure in t such that either (a) none of the successor actions of bp_Π in Π match any of the successor actions of bp_t in t, or (b) there is a matching successor action in Π, but its resulting structure does not embed the resulting structure in t. A branch point will not be found only if the example trace after the last merge point is completely subsumed by a path in Π.

In this way *findBranchPoint* gives us a situation where the example trace behaved differently from the existing plan. In general, the search for subsequent merge points can range over all nodes in Π. Allowing merges with any node in Π introduces loops of increasing complexity, which makes it difficult to determine vital properties such as the guaranteed termination of the resulting plan. From this point of view, we limit the set of allowed merge points to non-ancestors of the last branch point and nodes within the same loop. The list of non-ancestors is obtained by running BFS on Π with its edges inverted, and taking the complement of the obtained set of reachable nodes. The resulting plans can be analyzed for preconditions very efficiently using methods from Chapters 4 and 5.

The overall *BranchAndMerge* algorithm works by adding nodes for structures and edges labeled with actions from the branch point to the merge point (bp_t, mp_t respectively) in the trace t, starting at bp_Π in Π and ending at mp_Π. If the merge point in Π coincides with the previous branch point, Alg. 6 introduces a new loop. If a merge point is not found, all the actions and structures from bp_t are added to Π, in a linear path starting at bp_Π. Alg. 6 then calls the *formLoops* algorithm (Alg. 5) in order to find loops in the path of actions that was added after bp_Π.

Given a generalized plan Π with Π_E edges and a new trace t with t_n nodes, Alg. 6 runs in time $O(\Pi_E \cdot t_n)$ and satisfies the following property:

Observation 1 In any plan produced by Alg. 6, the shortest path to the goal from any concrete member of the initial belief state is smaller than or equal to the best provided example that solved it. This is because action sequences from example traces are either merged with existing edges that subsume them, or are added to the existing plan.

A Hybrid Approach Alg. 6 can be implemented as a part of a proactive algorithm for incrementally generating example plans and merging them (Alg. 7). Alg. 7 uses the list of looseEnds which can be created by *Trace*. It requires a book-keeping subroutine for removing structures which have been solved from the list *looseEnds* when example traces are merged with the existing plan Π.

6.2.3 A Detailed Example

Fig. 6.14(a) shows a plan segment that collects one object of type paper, moves to the next bin and finds a glass object. $S_0^\#$ is a concrete structure in which more than 2 objects each of type paper and glass have been collected, and two bins remain to be visited. Two of the actions in this example, *gotoNextBin* and *senseType*, can have multiple abstract results due to the focus operations described earlier. When applied on an abstract structure with an unknown number of unvisited bins, the two results of the *gotoNextBin* action correspond to whether or not the next bin is the last unvisited bin, as per the drawing-out operation described earlier (Fig. 6.13). The *senseType* action uses the focus operation to enumerate the different possibilities for the type of the object being sensed. Dotted edges in Fig. 6.14 represent results of these actions that did not occur in the execution of the given example plan on $S_0^\#$.

Fig. 6.14(b) shows the result of generalizing Fig. 6.14(a). $S_0^\#$'s canonical abstraction, S_0, is identical to S_4, the abstract result of collecting another object of type paper. This is recognized by *formLoops* (Alg. 6, line 10) because at this stage, the plan Π is empty. *formLoops* creates a loop by attaching the "*collectPaper()*" edge to S_0 (Fig. 6.14(c)). The following action edge (*gotoNextBin()*) from $S_4^\#$ however, is not merged with the edge between S_0 and S_1 because $S_5^\#$ and its abstraction S_5 do not have any elements with the role of "unvisited bins", thus differing from S_1.

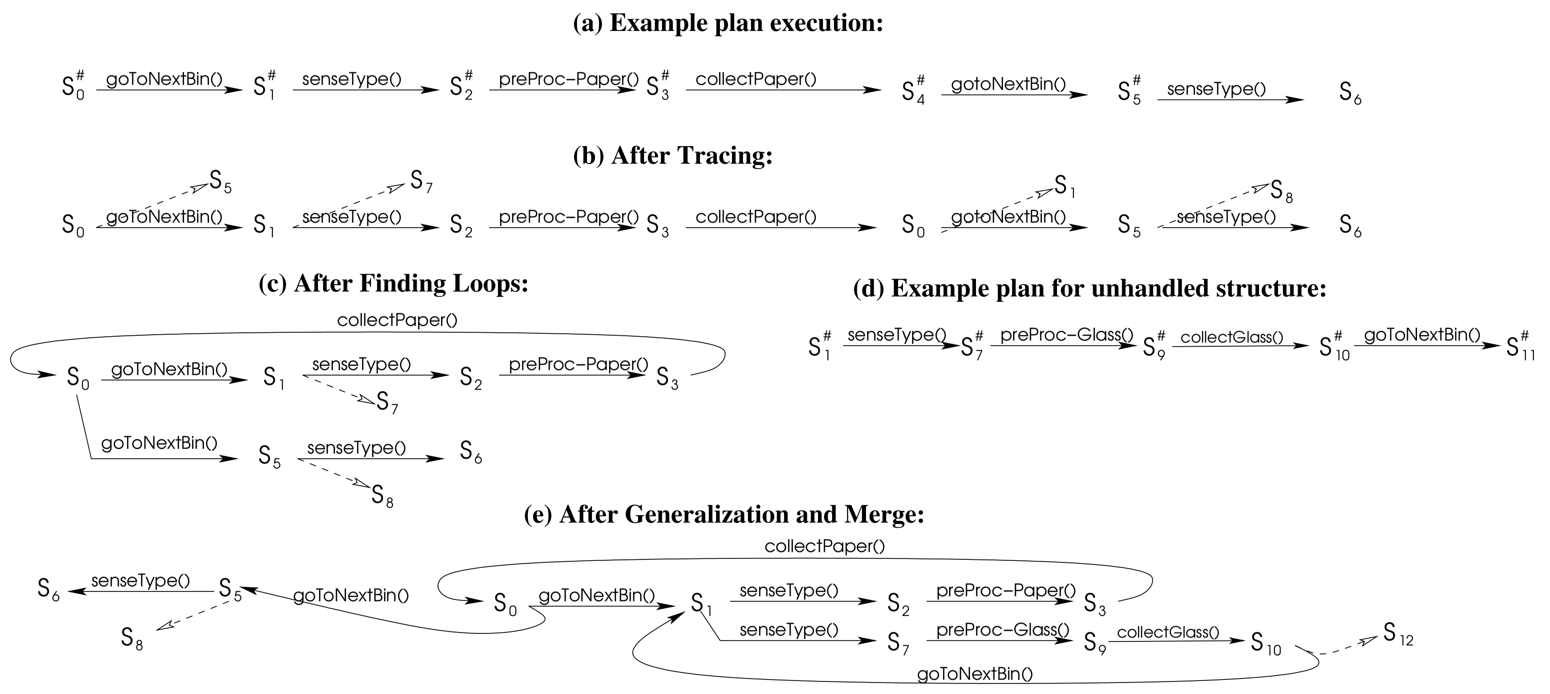


Figure 6.14: A detailed example for *Merge*. Dotted edges represent results that did not occur in the example.

Fig. 6.14(d) shows an example plan for handling a structure identical to $S_1^{\#}$, but with the type of the object in the bin set to glass. This plan is also traced in the abstract state space and Alg. 6 is called with the resulting trace and the current generalized plan (shown in Fig. 6.14(c)). Alg. 6 in turn calls findMergePoint, which identifies S_1 as a merge point. It then invokes findBranchPoint, which also returns S_1. This is because the result of the *senseType* action on S_1 is S_7 in the generalized trace, where the chosen bin has an object of type glass (unlike S_2, where it was paper).

After finding this branch point, Alg. 6 calls findMergePoint again, and this time, cannot find any merge points in the example trace before S_{11}, which it determines can be embedded in S_1. It returns S_1 in Π and S_{11} in t as the merge point, following which the subroutine *addEdges* is used to add the structures and actions between S_7 and S_{11} to Π.

6.2.4 Implementation and Results

We present the results of some of our experiments with an implementation of *BranchAndMerge*. The test problems were motivated by benchmarks from the international planning competitions and require solutions with different combinations of loops and branches. Incremental results for each problem are shown in Fig. 6.15, with segments added due to different examples labeled and drawn with different edge types. The actual outputs are more detailed, and include one iteration of the loop learned using the first example prior to the topmost action shown in the figures. Since the loops tend to get too complex to understand visually, we present modified outputs in order to aid readability: structure-nodes and edge labels for results of sensing actions are not drawn and some action operands are summarized into action names. We present a summary of these results with their incremental domain coverages, and provide representative detailed results and execution times for the recycling problem.

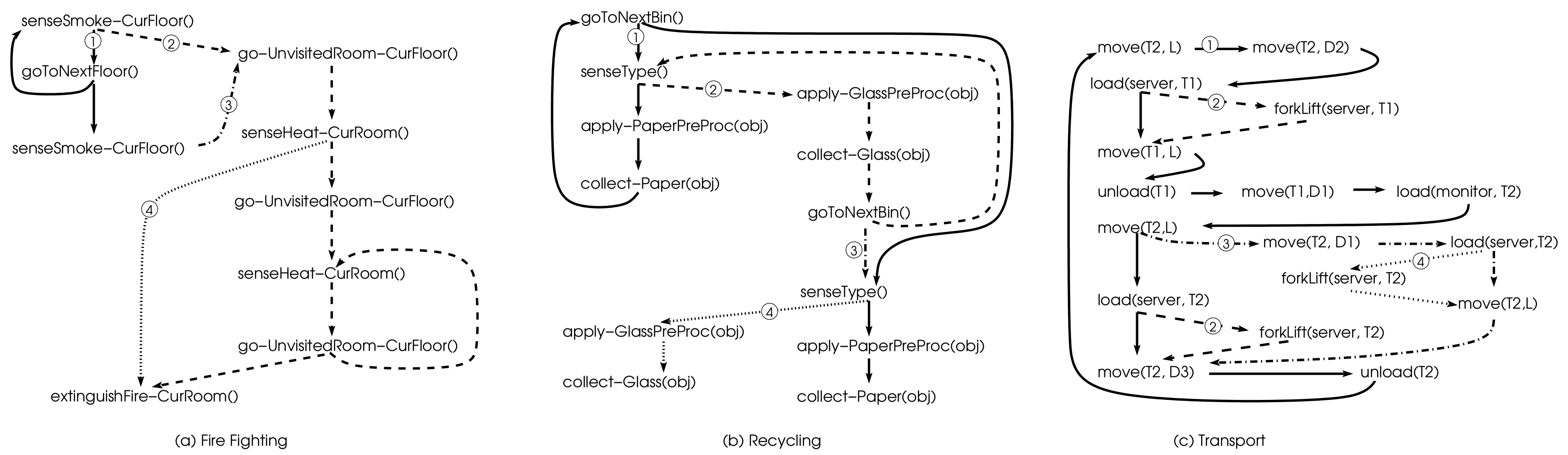


Figure 6.15: Segments of computed plans. Circled numbers and edge types label components from different examples.

The fact that all the loops make progress and terminate can be determined automatically by computing the role-count changes using methods described in Chapter 5.

Fire Fighting This problem was discussed in the introduction. Smoke can be detected from anywhere on a floor iff one of its rooms is on fire. The agent has smoke and heat sensors; it can use the *senseSmoke* and *goToNextFloor* actions to reach the correct floor, and the *senseHeat* and *goToRm* actions to find the room on fire. The *extinguishFire* action can be used to extinguish a fire. The number of rooms and floors in the building are unknown, and unbounded.

The first example plan solved an instance of the problem with 6 floors, with 1 room on each floor. None of the floors were smoky in this problem instance (we did this to stress *BranchAndMerge*; a problem instance with a smoky floor would have extracted more of the solution plan from the first example itself.). The example plan used *goToNextFloor* to traverse all the floors but found none to be smoky. Since this was the first example, *BranchAndMerge* called *formLoops* which created the loop labeled (1) (Fig. 6.15(a)).

The second example plan solved a smaller problem instance. In its initial state, the agent is on the fourth floor of a building with 6 floors and this floor is smoky, but the smoke has not been detected. The fire is in room 4; there are 5 rooms on this floor (the agent starts at the fourth floor to make it harder to identify the context – starting at the first floor would have provided a large prefix of actions matching those of Π). The example plan used *senseHeat* and *goToRm* actions to visit rooms 1,2, and 3 before reaching room 4, sensing heat, and extinguishing the fire. *BranchAndMerge* found that the initial structure of this plan was embeddable in the abstract structure in loop 1 (Fig. 6.15(a)(1)), corresponding to the agent being at any floor of the building. The first *senseSmoke* action was also merged with (1), but its result and the remainder of the example trace was not embeddable anywhere in the existing plan (Fig. 6.15(a)(1)). A loop was also detected in the remainder of this trace (Fig. 6.15(a)(2)). The generalized plan formed using examples (1) and (2) does not solve some boundary cases, for instance when the first floor is smoky or when the first room in a floor has fire. Example plans 3 and 4 handled these situations. However, only two edges were added from these plans, connecting structures already in the generalized plan. In the final plan, there are no unresolved action branches indicating that the goal structure with the fire extinguished is always reached.

Recycling This problem was used as the running example and its solution was described in Sec. 6.2.3. *BranchAndMerge* creates a loop in this example, illustrating how small examples can be used to identify powerful loops. Example 3 dealt with an unhandled branch caused due to the drawing out of elements from a summary element (last bin was reached), and example 4 handled the case where the last object was of type glass.

Transport We have a Y-shaped transport map with depots D_1, D_2, D_3 on the end points. Two trucks, T_1 and T_2 with capacities one and two are originally at D_1 and D_2, respectively. The problem is to deliver an unknown number of *server* crates (from D_1) and *monitor* crates (from D_2) in pairs with one of each kind to D_3. Location L at the center of the Y can be used to transfer cargo between the two trucks. There are two non-deterministic factors in this problem: *server* crates may be heavy, in which case the simple load action drops them and a forkLift action must be used; crates left at L may get lost if no truck is present.

The first example plan delivered 6 pairs of crates to D_3 without experiencing heavy crates or losses. The second example found a heavy crate, and delivered it using forkLift actions instead of load; in the third plan a crate left at L was found missing when T_2 reached L, and another crate had to be picked up from D_1. The plan computed using these three examples does not handle one case of a *server* crate being heavy (Fig. 6.15). This was was handled by example plan 4.

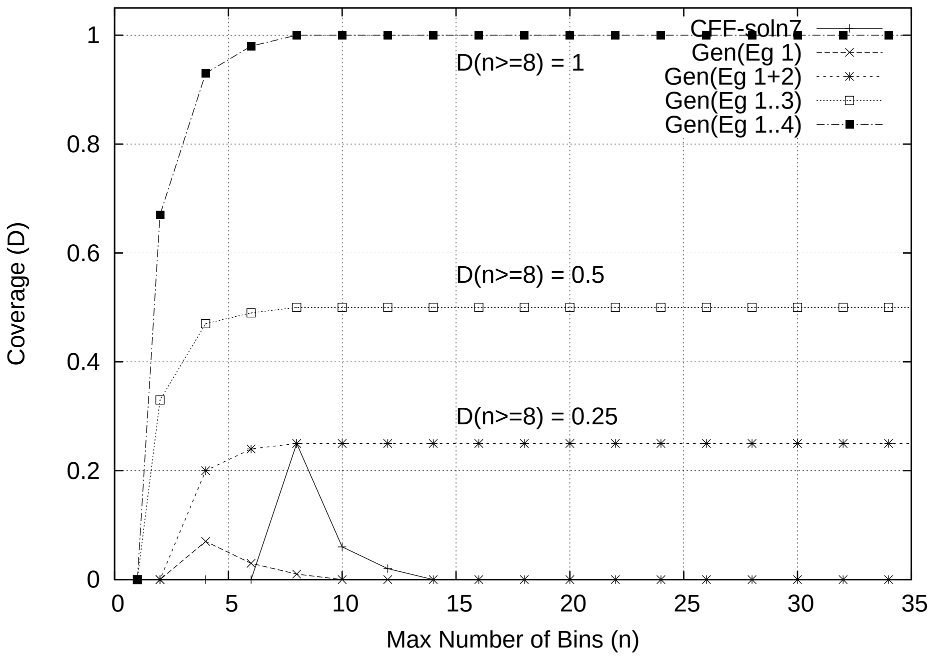


Figure 6.16: Domain coverage of recycling problem plans.

Key Observations All the presented solutions solve problems of unbounded sizes. *BranchAndMerge* adds only necessary segments from example plans. For instance, only edges for the two *forkLift* actions from the entire second example in transport were added. In fire fighting, the result of *senseHeat* action in example 4 of the fire fighting problem was directly merged to a structure that had already been handled. Merging plan segments within loops is a powerful technique for increasing the scope of the plan far beyond the individual examples: in recycling, the plan learned using the first example solves only n of the $2^{n+1} - 1$ possible problem instances of size at most n. The second plan covers a single specific problem instance. The generalized, merged result using these two plans solves 2^{n-1} instances (it assumes that the last two bins have paper).

Plan	Gen(1)	Gen(1..2)	Gen(1..3)	Gen(1..4)	CFF-soln7
Time(s)	110	129	134	144	262

Table 6.5: Solution Times

Further Details and Comparison We illustrate the incremental increases in domain coverage discussed above with plots (Fig. 6.16) and the times (Table 6.5) taken to generalize and merge input example plans for the recycling problem. Fig. 6.16 shows that the domain coverage $D_{\pi}(n)$ increases significantly with each new example plan, and approaches 1 with four examples. Since no other approach can solve these problems due to uncertainties in object quantities, direct comparisons are not feasible. However, to put this in perspective, we compare these results with the domain coverage and execution time for the largest recycling problem instance (with 7 bins) that we could solve using contingent-FF (Hoffmann and Brafman, 2005), a state-of-the-art contingent planner. Given the four example plans for recycling described above, the generalization and merging process produces a near complete solution while taking 45% less time than the time taken by contingent-FF to find a plan (CFF-soln7) for 7 bins. Generalized plans for all the other problems discussed above were generated in under 300 seconds and showed similar comparative performance with contingent-FF. Tests were conducted on a 2.5GHz AMD Dual-Core machine with 2GB of RAM.

6.3 Discussion

Summary This chapter presents two fundamentally new approaches for creating generalized plans with loops from example plans. During the process of construction of loops, we focus on two main aspects: (a) the loop must have a justification in terms of a recurring set of properties observed in the example, and (b) the loop must make measurable progress, and terminate in a bounded number of steps. In doing so, we developed the only known approach for merging a loop of actions with a potentially useful sequence of actions from a new plan. Our use of abstract states allows us to observe these two conditions and also in storing the set of abstract states possible at each step in the generalized plan. In the next chapter, we will use these states to identify situations that are not handled by the existing generalized plan, and ultimately to extend the generalized plan to solve them.

Directions for Future Work Although many of the presented algorithms in this chapter are the only known ones for safely constructing generalized plans, are demonstrably scalable, and produce generalizations with a large class of loops, they can be optimized and enhanced functionally. In particular, the construction of loops can be directed so as to maximize the progress along a certain specific role-count. The current implementation is in Python, which is an interpreted language. Better performance could also be achieved by an implementation of the same algorithms in a compiled language.

In the presented algorithms, loops are recognized when an abstract structure is revisited. However, a concrete plan may have used different orderings of an action sequence to achieve the effects corresponding to each iteration of the identified loop. Although this will not effect loop identification, reordered actions will result in fewer merges and more plan branches accomplishing the same effects. Another interesting direction for future work is to optimize the plan merging process by eliminating these spurious re-orderings of actions in a plan. This can be done either after plan generation, or even during the generation of classical plans by introducing a lexicographic ordering on actions with the same heuristic impact during search.

Related Work Recent approaches to finding plans with loops include KPLANNER, DISTILL and the work of Bonet et al. (2009). Of these approaches, only KPLANNER provides a partial applicability test. A more detailed discussion of these approaches was presented in Chapters 1 and 4. The objective of learning loops by generalizing concrete plans is similar to the objectives of programming by demonstration (PBD). In practice, approaches for PBD follow very different assumptions compared to those in the field of AI Planning. Lau et al. (2003) address the significantly different problem of using a given segment of a user's actions (e.g. keystrokes in a text editing task) to predict the remainder of the program being executed. In this approach, loop iterations in training examples are explicitly annotated by the user. The SHEEPDOG (Lau et al., 2004) system on the other hand uses an extension of Hidden Markov Models to predict the next most likely action required during a technical support task. Instead of computing the preconditions for their learned structures, both of these systems provide probabilistic quality and usability guarantees. Such guarantees can be useful in many settings, particularly those where a limited domain theory prevents precondition analysis. The PLOW system (Allen et al., 2007) also captures loops via demonstration, but uses a mixed-initiative approach where the user provides cues to the system for beginning a loop recognition process, proactively corrects the system's errors while demonstrating a solution, and provides explicit loop termination conditions.

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

CHAPTER 7

GENERALIZED PLAN SYNTHESIS

This chapter presents two approaches for computing generalized plans without a priori, user-provided sample plans. The first approach (Section 7.1) computes generalized plans with loops by conducting a search in the abstract state space, using the abstract action mechanism developed in Chapter 3. In practice, this approach would require heuristics applicable to paths with loops in the abstract state space to make the search process efficient. The second approach (Section 7.2) exploits the directed search mechanism of classical planners themselves, for finding generalized plans.

7.1 Synthesis via Search

The abstract action mechanism developed in Chapter 3 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 ($\{crate\}$; $\{crate, chosen\}$; $\{crate, delivered\}$; $\{crate, chosen, delivered\}$), 1 possible truck role, and 2 possible location roles ($\{location\}$; $\{location, targetDest\}$). 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 10). It proceeds in phases of search and annotation. During the search phase it uses the procedure *getNextSmallest* to find a path π (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 π, if we are in an extended-LL domain. Methods for *findPrecon* were described in Chapters 4 and 5. 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.

Algorithm 10: ARANDA-Synth

```
Input: Generalized Planning Problem $\langle S_0, \mathscr{D}, \varphi_g \rangle$, where $S_0$ is an abstract
       structure
Output: Generalized Plan $\Pi$
$\Pi \leftarrow \emptyset$
while $\Pi$ does not cover $\mathscr{I} = \gamma(S_0)$ 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 $C_\pi$
    end
end
```

Algorithm 11: getNextSmallest(S_0, φ_g)

```
   { /* All data structures are global, and maintained across calls
   if $S_0$ does not have queues then
      MsgInQ($S_0$) ←{[ ]}; MsgOutQ($S_0$) ←∅
                                                                        */
1  $\pi \leftarrow \emptyset$; Pause← False
2  while $\pi$ is not a path to $\varphi_g$ do
3      forall S with non-empty MsgInQ or MsgOutQ do
4          Activate(S)
   }
Procedure Activate(S) {
5  while MsgInQ(S) $\neq \emptyset$ and not(Pause) do
6      m ← PopQ(MsgInQ(S)); genMsg(S,m)
7  forall $MsgOutQ(S) \neq \emptyset$ do
8      m ← PopQ(MsgOutQ(S)); sendMsg(S,m)
   }
Procedure genMsg(S, m) {
9  if $S \in m \setminus loops$ then
       /* $m \setminus loops = [S_0,S_1,\ldots,S_{k-1},S,\ldots,S_l]$; A loop is formed   */
10     m'← createLoop(m,S)
       /* $m' \leftarrow [S_0,S_1,\ldots,S_{k-1},[S,S_{k+1},\ldots,S_l]]$                    */
11     PushQ(MsgOutQ(S),m')
12     $m_S \leftarrow$ append(m',S)
13 else                                                  /* Loop not formed */
14     $m_S \leftarrow$ append(m,S)
15 PushQ(MsgOutQ(S), $m_S$)
16 if $S \models \varphi_g$ then
       $\pi \leftarrow m_S$;Pause← True                      /* The next path */
   }
Procedure sendMsg(S, m) {
17 if not newLoop(m) then
18     for $S_n \in$ 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         $S_n \leftarrow$ successor of S in the loop
24         rcvMsg($S_n$,m)
   }
Procedure rcvMsg(S, m) {
25 PushQ(MsgInQ(S), m)
   }
```

Procedure *getNextSmallest* (Algorithm 11) 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. 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 structure's *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 go only to those structures that 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 10 realizes the following theorem.

Theorem 9. (Generalized planning for extended-LL domains) *Algorithm 10 finds most general plans amongst those with simple loops in extended-LL domains.*

Example 8. *Fig. 7.1 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. In practice, heuristics would be required

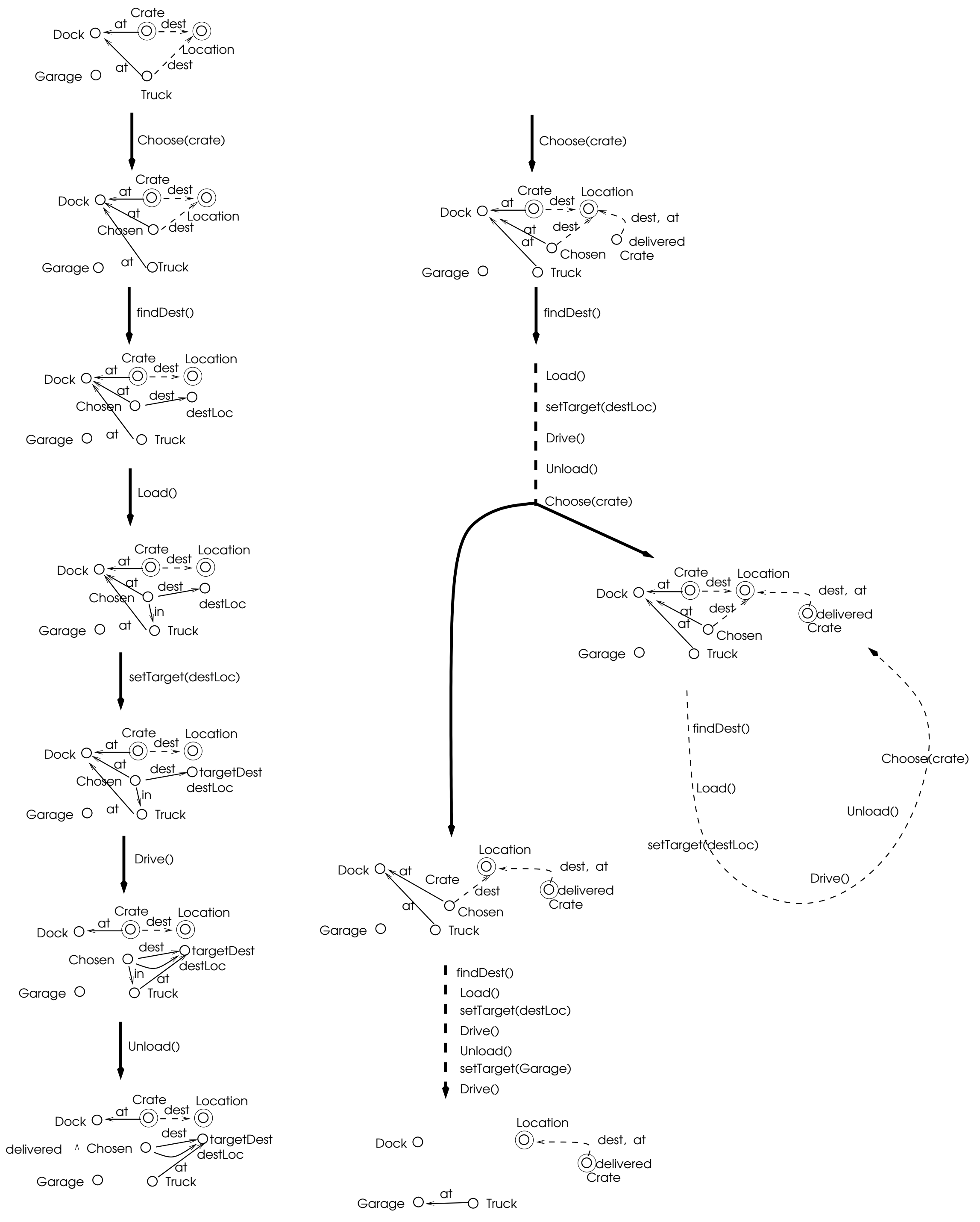


Figure 7.1: Example for ARANDA-Synth

to guide the search process in a manner similar to classical planners. However, in this case the heuristics would have to be loop “aware", to allow an action to lead back to a previously visited state in situations where the resulting loop would be beneficial.

7.2 Hybrid Plan Synthesis

We now present a hybrid approach for computing generalized plans that are guaranteed to terminate. Upon termination, these plans either achieve the goal, or result in one of a known set of possibly unsolved situations. In many planning domains, the quality of provided guarantees is significantly better, and includes a precise characterization of the set of problems solved along with the number of steps required to reach the goal from a given problem instance. The overall architecture of the approach (Fig. 7.2) relies upon generating an instance of the current class of unsolved states and solving it using a classical planner. A generalized version of this solution is then incorporated into the generalized plan. Maintaining the set of unsolved problem instances thus allows the process to be directed towards the currently unsolved problem classes. At any stage during the construction of a plan, the partial generalized plan demonstrates similar correctness guarantees and can be effectively used.

The overall algorithm is presented as Alg. 12. The generalized plan is initialized with a single node labelled with a set of initial states represented using an abstract structure S_0. We will assume that S_0 is at least *role-abstracted*: that it has at most one element (either a singleton or a summary element) for each role. During plan synthesis, a list of un-resolved nodes (non-goal nodes with out-degree zero), called *open nodes*, is maintained. This list is initialized with n_0. Open nodes represent an upper bound on the set of problem states that are not solved by the generalized plan.

The main loop of the algorithm iteratively picks an open node n, obtains a classical plan solving one of the instances represented by *Struc(n)*, generalizes that plan, and finally, merges it back into the existing generalized plan. The choice of the next open

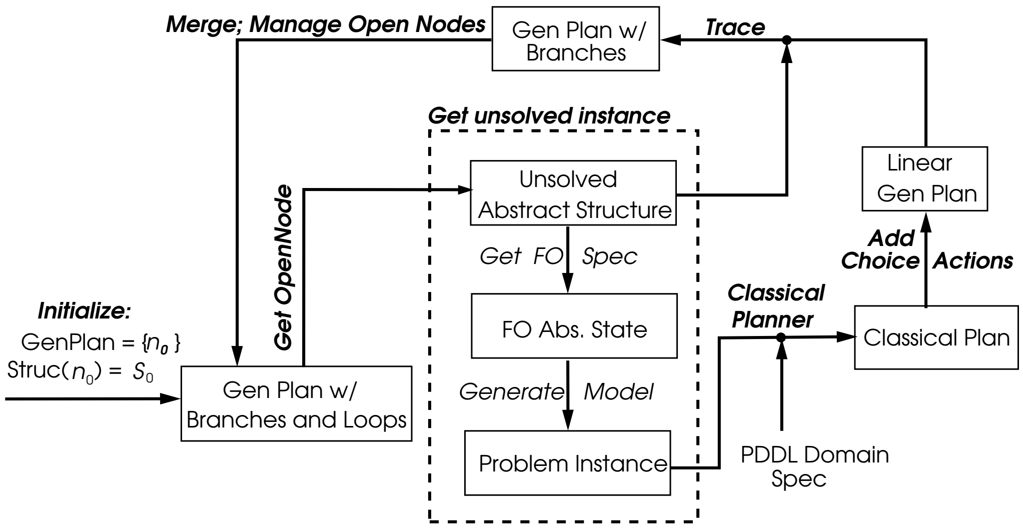


Figure 7.2: Hybrid plan synthesis

Algorithm 12: Hybrid Generalized Plan Synthesis (ARANDA-Hybrid)

```
Input: Abstract structure $S_0$, domain $\mathscr{D}$, goal formula $\varphi_g$
Output: Generalized Plan $\Pi$
1  $\Pi \leftarrow \langle V = n_0, E = \emptyset, Struc(n_0) = S_0, OpenNodes = (n_0)\rangle$
2  repeat
3      Pick an open node $n$
4      RoleCounts $\leftarrow$ GetValidRoleCounts($S_n$)
5      $\varphi_n \leftarrow$ GetFOSpec(Struc($n$), RoleCounts)
6      $C_n \leftarrow$ ModelGenerator($\varphi_n$)
7      $\pi_n \leftarrow$ ClassicalPlanner(PDDL_Translation($C_n$), $\mathscr{D}_{PDDL}$)
8      $\pi_n^c \leftarrow$ AddChoiceActions($\pi_n$, PDDL_Translation($C_n$))
9      $t_n \leftarrow$ Trace($\pi_n^c$, $Struc(n)$)
10     $\Pi \leftarrow$ MergeWithOpenNodes($\Pi, t_n, n, C_n, \pi_n^c$)
   until $OpenNodes = \emptyset$ or $InstanceLimitReached$
```

node to be solved can be prioritized; in our experiments we use a random ordering of open nodes.

Given an open node n, steps 4-6 generate a concrete instance of the set of states represented by *Struc(n)* using a first-order model generator. This process is discussed in detail below. In step 7, a classical planner is invoked with a PDDL version of this problem instance, an instantiated goal formula and a PDDL version of the domain $\mathscr{D}$. *AddChoiceActions* then generalizes the obtained plan π_n by inserting argument-selecting choice actions before each of its actions. The roles for action arguments are computed in *AddChoiceActions* by simulating an execution of the plan π_n on C_n.

Subroutine *Trace* in step 9 uses a modified version of the algorithm discussed in Section 6.1.1 to keep track of the problem instances which will *not* be solved by the generalization of a computed classical plan. Recall that in the trace subroutine, abstract action operators are applied to abstract structures starting with S_n. Whenever an action leads to multiple abstract states due to focus operations, the next action is applied only on the state that embeds the result obtained at that step in an execution of the plan π_n^c upon the concrete instance C_n. Traces are represented using the same representation as generalized plans but are much simpler, consisting of a single main branch going from S_n to the goal, with branches leading to leaf, or open nodes of the trace. These nodes capture possible results of actions that did not occur in the example plan; consequently, states represented by these nodes may not have a path to the goal. In this way, members of *Struc*(n) that are not solved by the generalized example π_n^c get collected in the trace's open nodes. Step 10 merges this trace with the existing generalized plan. In doing so it also determines if the new open nodes are subsumed by existing nodes in the generalized plan. This is described in detail in Section 7.2.2.

7.2.1 Generating Concrete Instances

The subroutine *GetValidRoleCounts* creates an instance of valid role-counts for a given structure using the domain's integrity constraints. We specified integrity constraints on role-counts as simple linear equalities, such as $(\#Red = \#Blue) \vee (\#Red = \#Blue + 1)$ for the striped tower problem discussed in the section on results. Instances of such inequalities can be generated automatically using mathematical packages (we used *Mathematica*). The subroutine *GetFOSpec* returns a first order representation φ_n of the three-valued abstract structure *Struc(n)*. φ_n consists of three sets of axioms, Ax_u,Ax_e, and Ax_i, capturing facts about the elements of the universe, its relations, and the integrity constraints respectively. Recall that every element in an abstract structure corresponds to a unique role, and that its summary elements may correspond to multiple elements in a concrete structure that it represents. Let $S_n = Struc(n)$; $r(S)$ the set of roles with non-empty instantiations in S and $u(S_n)$ be the set of roles that correspond to singleton elements in S_n. We use the abbreviation $r_i(x)$ to denote first-order formulas for roles. That is, $r_i(x) \iff (\wedge_{p_j \in role_i} p_j(x) \wedge_{p_k \notin role_i} \neg p_k(x))$. Ax_u, Ax_e ensure that every element has one of the roles $r(S_n)$ and that definite truth values in S_n are respected:

$$
\begin{aligned}
Ax_u(S_n) \quad &\equiv \quad \forall x \; (\vee_{r_i \in r(S_n)} r_i(x)) \wedge \\
&\qquad \wedge_{r_j \in u(S_n)} \forall x, y \; (r_j(x) \wedge r_j(y) \implies x = y) \qquad (7.1) \\
Ax_e(S_n) \quad &\equiv \quad \bigwedge_{v \in \{\top, \bot\}} \quad \bigwedge_{\substack{p, r_i, r_j: \\ [[p(r_i, r_j) = v]]_{S_n} = 1}} \forall x, y \; (r_i(x) \wedge r_j(x) \\
&\qquad \implies p(x, y) = v) \qquad (7.2)
\end{aligned}
$$

Ax_u is extended in a straightforward manner to assert that every role has the number of elements obtained using *GetValidRoleCounts*.

The integrity constraints of a domain may include statements about transitive closure. We use the notation p^{tc} to denote the transitive closure of a predicate p in the vocabulary of a domain-schema. In general, transitive closure cannot be accurately formalized in first-order logic. This makes it impossible, in general, to create models for formulas in $FO(TC)$ in which predicate p^{tc}'s interpretation is the true transitive closure of predicate p's interpretation. In restricted settings however it is possible to do so using methods described by Lev-Ami et al. (2009). In this thesis, we use their axiom schema $T_1[p]$:

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

Theorem 10. *(Lev-Ami et al., 2009) In any finite and acyclic model of $T_1[p]$, p^{tc} is equivalent to p^+, the non-reflexive transitive closure of p.*

In other words, if the relation p is constrained to be acyclic, then in every finite structure, $T_1[p]$ defines p^{tc} as exactly the non-reflexive, transitive closure of p. We found that transitive closure in this restricted setting suffices to express necessary state properties such as “every block in the tower is above the bottommost block", “every grid square in a row is to the west of the eastern border" etc. Such expressions are used in the Striped tower, Hall-A and Reverse problems presented in Section 7.2.4.

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

For a three-valued structure S, let $FO(S) \equiv Ax_u(S) \land Ax_e(S) \land Ax_i$. The following result shows that step 6 in Alg. 12 generates a concrete instance $C_n \in \gamma(S_n)$ as long as the model generator employed is sound.

Theorem 11. *Suppose that all $p \in PTC$ are acyclic and S is an abstract structure corresponding to an open node in a generalized plan. Then a concrete structure C belongs to $\gamma(S)$ iff $C \models FO(S)$.*

Proof. Suppose $C \in \gamma(S)$. Then there must be an embedding of C into S. Because this embedding can only make truth values imprecise, it will have to map an element in $|C|$ to an element in $|S|$ of the *same* role (see Definition 5 on page 34). Thus, C must satisfy the first conjunction in Ax_u, stating that every element must have one of the roles in S.

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

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

Conversely, suppose $C \models FO(S)$. We need to construct an embedding from $|C|$ into $|S|$ and show that C satisfies all the integrity constraints, and interprets p^{tc} correctly.

Because $C \models Ax_u$, we know every element's role corresponds to an element's role in S. Further, since S corresponds to an open node in a generalized plan, it must have at most one element of each role: this is assumed to be true for the initial structure; the action update mechanism (Definition 8 on page 42) ensures that it continues to hold for every structure resulting from action application. The required embedding therefore maps every element in $|C|$ to the element with the same role in $|S|$. Axiom Ax_e ensures that this mapping is an embedding.

Since all $p \in PTC$ are acyclic, C interprets p^{tc} correctly by Theorem 10. Finally, C must satisfy all the integrity constraints because we have $C \models Ax_i$. □

7.2.2 Merging Traces with Open Nodes

```
Algorithm 13: MergeWithOpenNodes
    Input: Generalized plan $\Pi$, open node $n_0$, new trace $t$, concrete state $C_n$, plan $\pi_n^c$
    Output: Extended version of $\Pi$
    /* $t_0$ = start node of $t$ */
1   $bp_\pi, bp_t \leftarrow 0$; $\mathrm{mp}_\Pi, \mathrm{mp}_t \leftarrow n_0, t_0$
2   repeat
3       if $mp_\Pi$ found then
4           $\mathrm{bp}_\Pi, \mathrm{bp}_t \leftarrow$ findBranchPoint($\Pi$, t, $\mathrm{mp}_\Pi$, $\mathrm{mp}_t$)
        end
5       if $bp_\Pi$ found then
6⋆          $\mathrm{mp}_\Pi, \mathrm{mp}_t \leftarrow$ findMergePoint($\Pi, t, \mathrm{bp}_\Pi, \mathrm{bp}_t$)
7⋆          addEdges($\Pi$, $t$, $bp_t$, $mp_t$, $mp_\Pi$, $bp_\Pi$)
        end
    until new $bp_\Pi$ or $mp_\Pi$ not found
8⋆  tryMergeOpenNodes($\Pi$)
9⋆  for $m \in$ Nodes($\Pi$): $C_n \sqsubseteq Struc(m)$ do
10⋆     MergeWithOpenNodes($\Pi$, $Trace(\pi_n^c, Struc(m))$, $m$, $\perp$, $\emptyset$)
    end
```

The *MergeWithOpenNodes* subroutine (Alg. 13) uses the overall structure of *BranchAndMerge* (Alg. 6 on page 115) while adding several functional and performance enhancements. Lines that were added or whose subroutines were significantly modified are labeled with a "⋆"; these changes are described below. The overall merge algorithm proceeds by identifying a node in the generalized plan which can embed a node in the trace. This node is considered as a merge point, where the trace and and the generalized plan can be merged. Once a merge point is found, the algorithm searches for a branch point, or a successive node where the example trace differs from the generalized plan. After finding a branch point, the algorithm recurses to find the next merge point and adds edges from the example trace between successive branch points and merge points in the generalized plan. In this way, it retains the control behavior shown in the example plan, while making the added segments usable from any of the merged nodes, thereby increasing the applicability of the generalized plan.

MergeWithOpenNodes makes a number of functional enhancements and optimizations upon this fundamental algorithm. When adding a new edge from the trace at a node v in the generalized plan, subroutine *addEdges* also adds any open nodes associated with that edge in the trace. During the addition of an edge, *addEdges* checks for two possible subsumptions: if the result node of the edge being added embeds any existing open neighbor of v, that neighbor is removed from the graph; symmetrically, if any of the open nodes being added are subsumed by an existing non-open neighbor of v, they are not added. Merging the trace produced for covering a single open node can thus resolve several other open nodes. Further, after finding a branch point v_b, *BranchAndMerge* limits the search for merge points to the non-ancestors of v_b and nodes within the loop containing v_b, in order to avoid creating loops that may be too complex to analyze. We extend this search to also include those ancestors v_a of v_b which satisfy the following constraints: (1) no path from v_a to v_b includes a node belonging to a strongly connected component of size more than 1, and (2) the loop created by merging v_b with v_a is guaranteed to terminate. (1) requires an enumeration of all acyclic paths from v_a to v_b. This is done using a modified version of DFS which marks a node as "visited" only after all its descendants have been explored, and always clears the temporary labels used during the exploration from a node. Termination is checked by searching for a role whose count undergoes a net decrease as a result of every possible new loop created by merging v_b with v_a. Since all input states of planning problems can be assumed to be finite (albeit unbounded), this ensures that every loop will terminate. Note that the strongly connected components produced in this manner are simple loops with shortcuts (see Chapter 4) — they consist of only acyclic paths between v_a and v_b prior to merge. This ensures that once the execution control exits a strongly connected component, it can never return to that component. These changes in Alg. 13 allow us to create many more progressive loops than in the original merge algorithm.

Finally, after completing a merge of the trace, *MergeWithOpenNodes* performs two extra steps pertinent to the management of open nodes. The subroutine *tryMergeOpenNodes* merges every open node in the plan that can be merged with *any* other node without creating a potentially non-terminating loop. Finally, step 10 in Alg. 13 checks if the example plan C_n can be embedded into another open node in the plan and if so, recurses with a call to trace and merge.

These enhancements not only allow us to effectively conduct a search for generalized plans, but also reduce the number of classical planner invocations and the size of the problem instance in each invocation. In the hall-A problem discussed in the Section 7.2.4 for instance, the ability of finding progressive loops with ancestors of branch points allowed the generation of the entire plan without ever invoking the planner with more than one element per role. This would not have been possible with the original merge algorithm.

7.2.3 Properties of Hybrid Plan Synthesis

Property 1 *When the generalized plan terminates on a problem instance, it does so either at an open node or at a goal node.* Action application on abstract structures succeeds only if its preconditions are definitely true (and thus will be satisfied in every situation possible at that stage in the generalized plan). Therefore, there is always an applicable action at any internal node n in the plan. The execution of a generalized plan created using Alg. 12 can therefore only terminate at a node with out-degree zero.

Property 2. *If a problem instance p cannot be solved by a generalized plan, its execution on p will necessarily terminate at an open node.* This is a direct consequence of Property 1. Structures represented by open nodes thus provide an upper limit on the class of problem instances that cannot be solved by a generalized plan. In this way, they constitute an index of unsolved situations and when they might occur.

Property 3. *Generalized plans created by Alg. 12 are guaranteed to terminate in any domain where changes in role-counts due to action operation on abstract structures can be computed automatically.* Computation of preconditions guaranteeing goal reachability, and consequently termination, was discussed in Chapters 4 and 5. Condition (1) enforced by Alg. 13 while creating loops ensures that execution control never returns to a strongly connected component after exiting from it. This, together with the fact that all loops terminate guarantees that the generalized plan will terminate. Further, the decreasing role-count for a loop also allows the computation of an upper bound on the number of times a particular loop can be executed. As discussed in Chapter 5, in extended-LL domains role-count changes can be computed automatically.

Property 4. *For plans computed in extended-LL domains, if all strongly connected components of the plan are simple loops or simple loops with shortcuts, then plan preconditions can be computed in terms of the role-counts in a given problem instance.*

This property follows from the precondition computation techniques described in Chapter 4 and Theorem 7 on page 81.

7.2.3.1 Reachability of Generated Instances

Action update on an abstract structure may result in a structure that embeds a superset of the actually possible concrete results (Sagiv, Reps, and Wilhelm, 2002). This over-approximation is neither entirely undesirable nor unintended — over-approximated state representations can potentially capture future states. Over-approximation is therefore fundamental to the mechanism of recognizing and creating loops. However, open nodes that represent only concrete states that are unreachable along any existing path in the plan can lead to wasteful model-generator and planner invocations. Such open nodes can be easily pruned using the precondition evaluation (Chapters 4 and 5): if the precondition for reaching a node is not consistent with the initial abstract state, then that node can be removed from the plan. If the preconditions are consistent with the initial abstract state, then they can be used to construct a concrete instance which can reach the open node. Thus, we have:

Property 5. *In extended-LL domains the set of problem instances covered by the generalized plan strictly increases with every iteration of the main loop in Alg. 12.*

Reachable, yet unsolvable open nodes may be produced in domains where action effects cannot be reversed. Determining that no concrete instance of an open node is solvable requires reasoning abilities beyond the scope of this thesis; however, if a node can be identified as unsolvable, and if preconditions can be computed for reaching that node, then an instance of the initial state satisfying those preconditions can be generated. The classical planner solution for this instance can then be merged with the generalized plan's start node as the initial merge point. In this way, the scope of the generalized plan can be extended to include problem instances that may have otherwise resulted in the unsolvable open node.

7.2.4 Implementation and Results

We implemented Alg. 12 in Python using Mace4 (`http://www.cs.unm.edu/~mccune/mace4/`) as the model generator and the classical planner FF (Hoffmann and Nebel, 2001). The implementation automatically computes changes in role-counts when possible and creates loops only when termination can be guaranteed. Table 7.1 shows a summary of the results including the number of planner calls, the largest problem instance generated, the total time taken for computing the generalized plan, and applicability status of the resulting plan, where "T" indicates proven termination, "C" indicates a complete solution, and "P" indicates an automatic proof of completeness. The experiments were carried out on a 1.6GHz-Intel Core2Duo laptop with 1.5GB of RAM.

We tested the applicability of this approach on an automated programming problem ("Reverse") of reversing a singly linked list. Actions in this domain consist of program

Problem	N_{calls}	$\|\|S\|\|_{max}$	T(s)	Applicability
Delivery	9	9	346	T, C, P
Hall-A	14	10	283	T, C, P
Reverse	9	8	153	T, C
Striped Tower	11	9	837	T, C, P
Transport	7	13	725	T, C, P

Table 7.1: Summary of results

statements over three variables, such as `ptr1 = ptr1->next`, `ptr1->next=ptr2`, and `ptr1=ptr2`. Pointer assignment actions in this domain can be irreversible. Linked list manipulation programs are particularly difficult to verify because of the possibility of unreachable data elements. Our approach computes the correct program with a loop for reversing the list pointers, but also generates two unreachable and unsolvable nodes. The computed program is guaranteed to terminate as per Property 1 and forms a complete solution, but a proof of completeness requires the pruning of these unreachable nodes.

We also tested the implementation on some open problems in the planning literature for which there are currently no approaches capable of computing generalized solutions with termination guarantees. The first two loops in the obtained plan for the striped tower problem move blocks to the table; the last loop moves blocks back in alternation; the plan includes edges by-passing these loops in cases with too few blocks. This plan has 10 terminal nodes, of which five are unsolvable (marked with a triangle), resulting from unsolvable initial instances; the goal nodes are marked with circles. Preconditions for reaching any node in this plan with given role-counts can be computed automatically using methods presented in Chapters 4 and 5. Other problems included in the tests are Delivery and Transport (Srivastava, Immerman, and Zilberstein, 2008) and Hall-A (Bonet, Palacios, and Geffner, 2009). The solutions to all of these problems are provably complete and terminate.

7.3 Discussion

Summary The algorithms presented in this chapter bring together all the ideas developed in this thesis. Both the approaches for plan synthesis presented here make extensive use of methods for analyzing loops of actions developed in Chapters 4 and 5. Both approaches also use abstract state representations to recognize potential loops; ARANDA-Hybrid also utilizes these representations to merge similar plan segments, in the vein of the *BranchAndMerge* algorithm developed in Section 6.2. The experiments conducted using ARANDA-Hybrid show a close proximity between generalized planning and automated programming. As such, these algorithms demonstrate the utility of the methods for precondition analysis and state abstraction presented early in the thesis.

Future Work The ARANDA-Hybrid approach presents several opportunities for future research and optimization. The use of model generators can be studied further, to develop a better understanding of when finite models of restricted classes of first-order logic can be found efficiently. A backtracking version of ARANDA-Hybrid would be able to avoid problems associated with domains where the effects of actions cannot always be reversed. However, doing so requires methods for rolling a possibly “dead-end" structure back along the applied actions to a point where it was still solvable. As in the plan generation process, this may be easier to do with concrete instances of these dead-end structures. The development of loop-aware heuristics for guiding the search process in ARANDA-Synth also comprises an entirely new direction for future research.

Related Work In itself, the idea of building plans by adding segments covering new instances can be considered to be of low novelty. What makes these approaches stand out from the rest of the literature on planning is the ability to do this reliably, in the presence of loops of actions which result in a different sequence of concrete states in every iteration. Few approaches attempt to do so. Of the two most closely related approaches, KPLANNER recognizes patterns that are common across all the example plans it generates – in contrast, the example plans generated by our approach handle different aspects of a problem and can amount to partial solutions which cannot be generated as instantiations of the final generalized plan. Instead, these example plans capture the required set of actions starting from intermediate steps in the plan. On the other hand, while DISTILL does consider merging multiple example plans into dsPlanners, it does not handle the problems of (a) identifying problem instances not covered by a partial plan (b) merging plan segments into dsPlanners with loops, and (c) applicability and termination conditions of the generated dsPlanners.

As such, there is no systematic approach for a directed search for generalized plans that provides strong guarantees about execution outcomes or termination. Kuter et al. (2008) provide an approach for incrementally generating strong cyclic plans. Some of the ideas in this approach are similar to those behind ARANDA-Hybrid. This approach differs from ours most significantly along two dimensions: first, Kuter et al. deal with domains where actions may have multiple outcomes when applied on a state; in our formulation, this can only happen if there is partial observability, or if the state is an abstract state. In either case, these states represent collections of potentially unbounded states, and obtaining a classical plan requires us to generate a concrete instance. In contrast to Kuter et al.'s approach, our objective then, is to *generalize* the obtained plan before merging it with the existing generalized plan. Secondly, Kuter et al. use the framework of strong cyclic planning (Cimatti et al., 2003) where loops are introduced, or are desirable only when a linear plan cannot solve the given problem. In contrast, we create loops which make measurable progress in each iteration, with the objective of compactly representing solutions to multiple problems.

CHAPTER 8

CONCLUSION

The central investigation of this thesis has been on the analysis and computation of structures which can be efficiently constructed and used for solving broad classes of related, yet distinct problem instances. We introduced this problem, in its commonly studied form as follows:

Given a "class" of problem instances of interest, construct a "generalized plan" for "efficiently" solving them.

The motivation behind finding such generalized plans is natural, given the complexity of the planning problem. Modern classical planners have witnessed immense performance gains by treating the planning process as heuristic search, with automated methods for computing domain independent heuristic functions (Bonet and Geffner, 2001; Hoffmann and Nebel, 2001; Helmert, Haslum, and Hoffmann, 2007). However, scalability with increasing numbers of objects remains a challenging problem, especially in the presence of partial observability where "sensing" actions lead to multiple plan branches and an exponential growth in plan size. As a result, the problems motivating generalized planning have remained; research focus on them however, has decreased since their conception (Fikes, Hart, and Nilsson, 1972). Recent advances in constructing plans with loops (Levesque, 2005; Winner and Veloso, 2007) have revived interest in the field by demonstrating tangible methods for constructing plans which could potentially solve unbounded classes of problems.

At the time of presentation of this thesis, clarity on the fundamental problems in generalized planning had not improved much beyond the original efforts; and while the inclusion of loops in plans could arguably be the most significant factor in the current development of the field, their impact (both in terms of risks and benefits) was not well understood. In Chapter 2, we started our investigation with a clearer formulation of what we need from a generalized plan. This definition allowed us to distinguish the desirable factors of a generalized plan; a key component of this understanding is the fact that a generalized plan may be incomplete from the point of view of the original class of problem instances of interest, but may still be useful from the point of view of the problems that it does solve – provided, of course, that the two classes can be recognized efficiently. Given the nature of what we expect from generalized plans, the most general form of this problem of determining the class of problems that a generalized plan can solve (its *preconditions*) is unsolvable due to undecidability of the halting problem for Turing machines. However, we need to confront the same problems even during the construction of generalized plans with loops, where determining the utility or the preconditions of a potential loop of actions becomes a crucial step.

The architecture of generalized plans presented in Chapter 2 unifies a diverse array of planning frameworks, including classical planning, contingent planning, case-based-planning, and planning with loops, as well as approaches for representing partial plans or domain control knowledge (Baier, Fritz, and McIlraith, 2007). For the latter, it allows the generalized plan "structure" to be partial, or abstract. Plan instantiation methods for such plans will have to conduct a non-trivial amount of computation. The *cost of instantiation* of a generalized plan distinguishes generalized plans of varying specificity in terms of the amount of pre-computation embedded in them. In addition to this cost, the measures presented in Section 2.2 highlight the multiple objectives of generalized planning and the conflicting nature of these objectives.

In terms of these objectives, the approaches we presented create plans that have a worst case cost of instantiation $O(nL)$ where n is the total number of elements in the problem instance and L is the number of connected components in the generalized plan (every complete iteration of a simple loop or simple loop with shortcuts must decrease the role-count of some role by one). The applicability test can be constructed in time independent of the number of objects in the problem instance: $O(Lb)$ for plans with L simple loops and $O(RLK^2b)$ for plans with R affected role-counts, L strongly connected components with at most K shortcuts each, and at most b branches exiting the loop. Most of the output plans presented in this thesis have an asymptotic domain coverage of 1; loops are used to reduce the complexity of plan representation and computation. However, a greater understanding of measures of plan representation constitutes one of the directions for future research. The optimality of the produced plans is also 1 in most cases (all but the transport problem create optimal instantiations). Construction of generalized plans can be directed toward greater domain coverage or better optimality. Trade-offs between these two aspects and further development of optimizations focusing on either aspect will be considered in future work on the presented algorithms.

8.1 Representation

The state and action representation we use is reminiscent of classical approaches to AI. However, although we use a compact first-order representation, action updates are applied as queries on states defined as logical structures. The separation of operand-selection components of actions as "choice" actions makes it possible for us to generalize operands of actions easily, which is particularly useful when they are in a loop. Choice actions need to be instantiated at run time, and separating them brings out this important but often overlooked factor in the creation of generalized plans.

The representation of graph-based generalized plans is based on a well-established representation of control structures. This representation aids in complexity analysis and categorization of precondition evaluation, as demonstrated by the analysis of different classes of abacus programs in Chapter 4. The graph-based representation also makes it easier to conceptualize algorithms for constructing generalized plans by adding edges and merging structure nodes which subsume each other (Alg. 6).

Perhaps one of the key components in the development of this thesis is the use of state abstraction for representing sets of states, which can also be treated as belief states. Originally developed as a part of the TVLA system (Sagiv, Reps, and Wilhelm, 2002), this mechanism was used to represent supersets of program states possible at different steps in a program. Our use of this framework hinges on the mechanism for "drawing out" action operands prior to action application. The algorithms presented in this thesis restrict choice operations to selecting objects described only in terms of their roles. In future work these specifications can be extended to more general formulas and combined with approaches for symbolic computation of preconditions.

8.2 Abacus Programs and Their Relationship to Generalized Planning

The notion of abacus programs allows us to study the fundamental problems behind computing the effects of loops of actions as well as to solve them efficiently for a class of planning domains. Our analysis shows that computation of reachable states and preconditions is very easy in the case of abacus programs with simple loops. On the other hand, with non-deterministic actions common to many planning domains, arbitrary nested loops even with cycle rank one (simple loops with shortcuts) can be as hard as VAS reachability. The key factors responsible for this complexity are non-monotonicity and order-dependence. Without these factors, this computation becomes much more tractable (Theorem 3 on page 59 and Alg. 1 on page 62).

The methods developed in Chapter 4 are generally applicable to any system where action effects and branch conditions can be summarized in terms of numeric changes on a fixed set of registers. In Chapter 5 we describe how the state abstraction technique discussed earlier can sometimes allow us to achieve this; the general requirements for any abstraction process to be analyzed in this way are captured by the FC^3 conditions.

Extended-LL domains were defined as a special case of these conditions, by restricting the syntax of action update formulas. All domains specified in PDDL using only unary predicates can be expressed in this way. In addition, domains with binary predicates whose values can be determined (or derived based on the integrity constraints) using unary predicates of objects fall under this class.

The complexity of determining the numeric changes due to actions and branch conditions will depend upon the specific mechanism used for abstraction. In extended-LL domains, this can be accomplished in $O(r)$ operations, where r is the largest number of roles in the abstract structures encountered in the generalized plan. Extended-LL domains form a one to one correspondence with abacus programs (Corollary 2). Consequently, any control flow or plan can be translated into a plan with extended-LL domain actions (or even abacus actions). In principle, a single extended-LL domain is therefore sufficient to express any systematic procedure of computation and its effects. Extended-LL domains have an added benefit over abacus programs: they can be used to convert plans with loops into abacus programs in situ, without changing the loop structure. This gives us greater flexibility during plan construction, when structures that are not analyzable can be avoided.

More generally, branches caused due to actions or focus operations not within the extended-LL category can all be treated as non-deterministic from the point of view of precondition evaluation (reflecting the fact that we don't know how to categorize the results of those actions in terms of role-counts). This could allow us to compute guarantees on termination, but without precise preconditions for reaching desired loop termination nodes. Even in this situation, "open nodes" (Section 7.2) would represent the possible loop termination situations and the problem instances which need to be solved by classical plans during hybrid search. In future work, the scope of these methods can also be extended by identifying broader classes of problems where action effects can be summarized in terms of changes in the counts of objects satisfying properties that are more general than object roles.

8.3 Approaches for Plan Generalization

Approaches for constructing plans with loops typically use example plans to guide the addition of loops of actions. Various techniques can be used to detect these sequences: KPLANNER uses a logical definition of “loop un-windings” to identify patterns of actions in example plans; DISTILL uses an annotated partial ordering to identify repeating sequences of the same block of partially ordered operations, and controllers generated by Bonet, Palacios, and Geffner (2009) are essentially computed as the smallest correct representations of concrete plans solving an example problem instance.

In all of these approaches, the justification for adding a loop is based on experience with prior example(s). Our approach for generalizing example plans is similar to these approaches in the sense of relying on working examples; it differs significantly in the ideas behind recognizing, identifying, and constructing loops of actions. A potential loop is identified when a previously visited abstract state is revisited. Identifying a loop therefore does not require a repeating pattern of actions to occur in the given plan. Further, the identified loop is added to the plan only if it makes a net change in the count of at least one role. This ensures that the loop will not represent a recurring sequence of *concrete* states (the loop will not be “static”). Further iterations of the loop are merged with it if the following actions are exact un-windings of the loop.

This approach works very well on problems which require solutions with only simple loops. However, for generating compact plans it also requires that the next iteration of the identified loop in the example plan follow the same action ordering. In future work, we plan to develop methods for alleviating this constraint by preventing spurious action re-orderings in plans generated by classical planners.

Although example plans provide vital clues for identifying loops, there are no approaches for determining how to incorporate the knowledge provided by additional example plans into a partially constructed plan with loops. The root of this problem is that the states possible in intermediate stages of the plan are not known – adding a branch to a loop will therefore produce unpredictable results. In our approach however, loops are recognized in terms of their *invariants*, or the properties that *must* hold after each action in every iteration of the loop. These invariants are represented compactly in terms of abstract structures on which we can directly trace and attach useful operator segments from new example plans.

8.4 Approaches for Plan Synthesis

We discussed two approaches for plan synthesis which do not rely upon user-provided examples for producing generalized plans. The first (Section 7.1) offers a state-search based solution to generalized planning which has never been developed before. Heuristic search has proved to be a very successful paradigm in general by providing domain-independent mechanisms for conducting directed search. In future work we plan to utilize the vast literature on heuristic search to develop methods for guiding the proposed search for paths with loops in the abstract state space.

We also presented a hybrid search algorithm which achieves goal-directed behavior by employing classical planners themselves. While any approach for plan generalization can be converted into an approach for hybrid plan synthesis by using an automated planner to generate example plans, the challenge is to be able to (1) give such a planner the right problems required for extending the existing plan, and (2) to be able to generalize the resulting examples while also utilizing the structure of the existing generalized plan.

In order to address (1), we use a first-order model generator to construct instances of the non-goal abstract states that a generalized plan may terminate in (Section 7.2.1

on page 141). With our framework for merging example plans (Section 6.2), classical plans which solve these instances can be merged into the generalized plan and can also be used to create loops with existing nodes of the generalized plan, thereby addressing (2). As a result, the hybrid search algorithm provides an incremental, goal-directed approach for constructing generalized plans.

As with any procedure for incremental plan generation, hybrid search can get stuck if the domain has unsolvable, "dead-end" states. In domains with "dead-ends", hybrid search requires a mechanism for backtracking in order to avoid committing to action sequences which can lead to such states. Development of such methods is an important direction for future work.

BIBLIOGRAPHY

Allen, J. F.; Chambers, N.; Ferguson, G.; Galescu, L.; Jung, H.; Swift, M. D.; and Taysom, W. 2007. Plow: A collaborative task learning agent. In *Proc. of the Twenty Second National Conference on AI*.

Alur, R.; Henzinger, T. A.; and Ho, P.-H. 1996. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering* 22:181–201.

Baier, J. A.; Fritz, C.; Bienvenu, M.; and McIlraith, S. A. 2009. Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners revisited. In *Proceedings of the ICAPS 2009 Workshop on Generalized Planning: Macros, Loops, Domain Control. September 20th, 2009, Thessaloniki, Greece*.

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proc. of ICAPS*, 26–33.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proc. of AIPS*, 52–61.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129:5–33.

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

Bryce, D.; Kambhampati, S.; and Smith, D. E. 2006. Planning graph heuristics for belief space search. *J. Artif. Intell. Res. (JAIR)* 26:35–99.

Bylander, T. 1994. The computational complexity of propositional strips planning. *Artif. Intell.* 69(1-2):165–204.

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

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

Cook, B.; Podelski, A.; and Rybalchenko, A. 2006. Termination proofs for systems code. In *Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

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

Eggan, L. C. 1963. Transition graphs and the star-height of regular events. *Michigan Mathematical Journal* 10:385–397.

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

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

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

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

Fox, M., and Long, D. 2003. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:2003.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in pddl3 - the language of the fifth international planning competition. Technical report, University of Brescia, Italy.

Ghallab, M.; Isi, C. K.; Penberthy, S.; Smith, D. E.; Sun, Y.; and Weld, D. 1998. Pddl - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Hammond, K. 1986. CHEF: A Model of Case-Based Planning. In *Proceedings of AAAI-86*, 267–271.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. of ICAPS*.

Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *AIPS*, 44–53.

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

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

Hoffmann, J., and Brafman, R. I. 2005. Contingent planning via heuristic forward search with implicit belief states. In *Proc. of ICAPS*, 71–80.

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

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

Hopcroft, J., and Pansiot, J.-J. 1979. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science* 8(2):135 – 159.

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

Kosaraju, S. R. 1982. Decidability of reachability in vector addition systems (preliminary version). In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, 267–281.

Kuter, U.; Nau, D. S.; Reisner, E.; and Goldman, R. 2008. Using classical planners to solve nondeterministic planning problems. In *Proc. of ICAPS*.

Lambek, J. 1961. How to program an infinite abacus. *Canadian Mathematical Bulletin* 4(3).

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

Lau, T. A.; Bergman, L. D.; Castelli, V.; and Oblinger, D. 2004. Sheepdog: learning procedures for technical support. In *Proc. of IUI*.

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

Levesque, H. J.; Reiter, R.; LespÃľrance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming* 31.

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

Levesque, H. J. 2005. Planning with loops. In *Proc. of IJCAI*, 509–515.

Minsky, M. L. 1967. *Computation: finite and infinite machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Peot, M. A., and Smith, D. E. 1992. Conditional nonlinear planning. In *Proceedings of the first international conference on Artificial intelligence planning systems*, 189–197.

Podelski, A., and 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.; and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems* 24(3):217–298.

Sanner, S., and Boutilier, C. 2009. Practical solution techniques for first-order mdps. *Artif. Intell.* 173(5-6):748–788.

Shavlik, J. W. 1990. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning* 5:39–40.

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

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *Proc. of AAAI*, 991–997.

Winner, E., and Veloso, M. M. 2003. Distill: Learning domain-specific planners by example. In *Proc. of ICML*, 800–807.

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

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