Acquiring Domain-Specific Planners by Example

Elly Winner Manuela Veloso

January 2003

CMU-CS-03-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number No. F30602-00-2-0549. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of DARPA or AFRL.

Keywords: plan execution, formation, and generation; program synthesis; inductive learning; agent modelling

Abstract

Intelligent problem solving requires the ability to select actions autonomously from a specific state to reach objectives. Planning algorithms provide approaches to look ahead and select a complete sequence of actions. Given a domain description consisting of preconditions and effects of the actions the planner can take, an initial state, and a goal, a planning program returns a sequence of actions to transform the initial state into a state in which the goal is statisfied. Classical planning research has addressed this problem in a domain-independent manner—the same algorithm generates a complete plan for any domain specification. This feature comes at a cost which domain-independent planners incur either in high search efforts or in tedious hand-coded domain knowledge.

Previous approaches to efficient general-purpose planning have focused on reducing the search involved in an existing general-purpose planning algorithm. An interesting alternative is to use example plans in a particular domain to demonstrate how to solve problems in that domain and to use that information to solve new problems independently of a domain-independent planner. Others have used example plans for case-based planning, but the retrieval and adaptation mechanisms were still domain-independent and efficiency issues were still a concern.

In my thesis, I propose to introduce algorithms to extract complex, repeating processes, in the form of domain-specific planning programs, from example plans. I will investigate the application of these learned programs to modelling agent preferences and choices. I will also investigate how the programs can be used, extended, and repaired dynamically as an agent encounters new problems and acquires new experience. Finally, I will compare the template-based planning paradigm to existing general-purpose and domain-specific planning programs with a full evaluation on new and existing planning domains. I expect the core contribution of this thesis to be a new planning paradigm in which domain-specific planners are learned by example.

Contents

1 Introduction

1.1 Motivation

Intelligent agents must be able to autonomously develop and execute a strategy for achieving their goals in a complex environment, and must adapt that strategy quickly to deal with unexpected changes. Solving complex problems with classical domain-independent planning techniques has required prohibitively high search efforts or tedious hand-coded domain knowledge, while universal planning and action-selection techniques have proven difficult to extend to complex environments.

Researchers have focused on making general-purpose planning more efficient by using either learned or hand-coded control knowledge to reduce search and thereby speed up the planning process. Machine learning approaches have relied on automatically extracting control information from domain and example plan analysis, with relative success in simple domains. Hand-coded control knowledge (or hand-written domain-specific planners) has proved more useful for more complex domains. However, it frequently requires weeks of work and great specific knowledge of the details of the underlying domain-independent planner for humans to formalize useful rules.

In my previous work, I have shown that example plans can reveal more than control information: they can also reveal the process behind their generation. I have introduced Sprawl, an algorithm for finding the rationale underlying observed example plans. I have also introduced Distill, an algorithm for extracting simple non-repeating programs, or templates, from example plans supplemented with their rationales. For my thesis, I will extend these techniques to extract rich algorithmic models of behavior from observed executions by developing methods of identifying and extracting complex repeated structures in example plan bases.

The applications of this work are much broader than rapid action selection. Agents operate in a world populated by many other agents, both human and machine. A fundamental task of these intelligent systems is to reason about the behavior of the agents around them so they can interact appropriately. My work on extracting algorithmic models of behavior from observed excutions could allow computers to be programmed by demonstration, allowing anyone, not just trained professionals, to program computers to perform complex tasks. It could help create general-purpose robots that could be trained to do a new task in minutes, simply by watching it be done. It could facilitate the cooperation of heterogenous agents by allowing them to quickly build models of each others' behavior, or could allow agents to predict and avoid the troublesome behavior of adversarial or non-cooperative agents. It could allow software to predict accurately and pre-execute commands for users, or even to automatically complete tasks like planning travel and scheduling meetings based on observations of the user's preferences.

1.2 Approach

There are three main steps towards achieving this goal. The first is to find the rationale underlying example plans by identifying and explaining the choices made by the agent in constructing the plan. The second step is to construct a domain-specific planner that reflects the choices and preferences observed in the example plans and identified in the rationale. The third step is to use this domain-specific planner to predict behavior or to solve new problems, and to extend or repair it in the presence of unsolved problems or new example plans.

1.2.1 Identifying Underlying Rationales

In order to predict and duplicate the behavior patterns of an observed agent, we must understand how the actions chosen by that agent help to achieve its goals. I call this understanding the *rationale* behind the plan. Previous approaches to plan analysis focus either on annotating plans or on finding partially ordered plans. Approaches to annotating orderings have been unable to handle domains with conditional effects or have relied on the underlying planner to provide the annotation, making them unable to analyze observed plans. Some approaches to partial ordering have explored finding a partial ordering of a totally ordered plan, though most deal with finding a partially ordered plan from scratch. While building on this previous work, we focus instead on identifying and explaining the choices made in creating an observed plan.

In Section 3.1, I present an algorithm that reveals the rationale underlying observed plans with conditional effects in the form of an annotated partial ordering. The partial ordering reflects the ordering constraints in the plan, helping to identify independent tracks of execution. The ordering constraints are annotated with the reasons they are necessary, allowing subplans to be matched and extracted from complete plans in a principled way [61].

1.2.2 Extracting Domain-Specific Planners

A principle difficulty in learning from example plans and in agent modelling is using the choices and preferences reflected in observed plans to construct a theory of behavior that can then be used to imitate or predict the actions of an observed planner. Previous approaches fall into two categories: those that extract control knowledge from the example plans that is then used to guide later planning search and those that compile exhaustive databases of observed plans and predict or generate actions by matching those databases.

I propose to develop learning techniques that avoid the cost of planning from scratch and of maintaining exhaustive databases by compiling observed example plans into compact domain-specific planners which are able to duplicate the behavior shown in the example plans and to solve new problems based on that behavior. In Section 3.2, I present an algorithm that moves towards this goal. The algorithm extracts simple, non-looping programs, or templates, from example plans supplemented with their rationales. I show that these templates, while storing little information, succeed in capturing observed behavior and in solving many new problems. In fact, templates extracted from only a few example plans are able to solve all problems in limited domains [62].

1.2.3 Using and Maintaining Domain-Specific Planners

Because my work will focus on extracting domain-specific planners from example plans, it will also have to address how to use and maintain planners which may be incomplete and incorrect. An incomplete planner can result when the example plans given to the learning algorithm cause it to learn a program that cannot solve all problems in a given domain. This will become evident when the planner is given a problem not solvable by its current program. An incorrect planner may result when the learning algorithm optimistically identifies loops in the observed example plans, or identifies loop conditions incorrectly. The planner will be revealed to be incorrect when subsequent execution on a new problem identifies non-executable steps in the loop.

Incomplete planners can be extended to cover new problems by querying an external generative planner for the solutions to the new problems and then incorporating the resulting example plans into the domain-specific planner, either by extracting a new domain specific planner from the updated database of solutions or by online modification. Extracting a new planner from scratch could allow the learning algorithm to develop a better (more compact, more general) program, but online modification would save the learning algorithm the substantial cost of maintaining a database of observed plans and of generating the new planner from scratch each time it is found to be incomplete.

It is not clear how to repair incorrect planners. One possibility is to remove loops found to be incorrect. Another possiblity is to add new conditions to the loops to better characterize their looping and stopping conditions. The problem may even be solvable simply by incorporating the new, troublesome plans into the domain-specific planner. I propose to investigate different methods for repairing incorrect planners and for generating planners unlikely to be incorrect.

1.3 Expected Contributions

1. A domain-independent algorithm for extracting the rationale behind an observed plan.

2. An algorithm for extracting domain-specific planners from example plans.

3. An algorithm for repairing or extending domain-specific planners in the presence of new example plans.

2 Background

2.1 Plan Analysis

Many researchers have addressed the problems of annotating orderings and of finding partially ordered plans. We discuss a selection of the research investigating annotation and partial ordering.

2.1.1 Triangle Tables

Triangle tables are one of the earliest forms of annotation [14]. In this approach, totally ordered plans are expanded into triangle tables that display which add-effects of each operator remain after the execution of each subsequent operator. From this, it is easy to compute which operators supply preconditions to other operators, and thus to identify the relevant effects of each operator and why they are needed in the plan. Fikes, Hart, and Nilsson used triangle tables for plan reuse and modification. The annotations help to identify which sub-plans are useful for solving the new problem and which operators in these sub-plans are not relevant or applicable in the new situation.

Regnier and Fade alter the calculation of the triangle table by finding which add-effects of each operator are *needed* by subsequent operators (instead of which add-effects remain after the execution of subsequent operators) [41]. They use the dependencies computed in this modified triangle table to create a partial ordering of the totally ordered plan.

The triangle table approach has been applied only to plans without conditional effects. When conditional effects are introduced, it is no longer obvious what conditions each operator "needs" in order for the plan to work correctly. Although we do not use the triangle table structure, our needs analysis approach can be seen as an extension of the triangle table approach to handle conditional effects.

2.1.2 Validation Structures

Another powerful approach to annotation is the validation structure [19, 20, 21]. This structure is an annotated partial order created during the planning process. Each partial order link is a 4-tuple called a validation: $< e, t', c, t >$, where the effect e of step t' satisfies the condition C of the step t. The validation structure acts as a proof of correctness of the plan, and allows plan modification to be cast as fixing inconsistencies in the proof. This approach is shown to be effective for plan reuse and modification [20] and for explanation-based generalization of partially ordered and partially instantiated plans [21]. The approach has not been applied to plans with conditional effects. Although [19] presents an algorithm for using the validation structures of plans with conditional effects to enable modification and reuse, no method is presented for finding these structures. And since the structures are created during the planning process, no method is presented for finding validation structures of any observed plans, even those without conditional effects.

2.1.3 Derivational Analogy

Derivational analogy [56] is another interesting approach to and use of annotation. In this approach, decisions made during the planning process are explicitly recorded along with the justifications for making them and unexplored alternate decisions. This approach has been shown to be effective for reusing not only previous plans, but also previous lines of reasoning. The approach can handle conditional effects, but, like the validation structure approach, is applicable only to plans that have been created and annotated by the underlying planner.

2.1.4 Operator Graphs

The final approach to annotation that we will discuss is the operator graph [50, 51]. This approach does not analyze plans, but rather interactions between operators relevant to a problem. The operator graph includes one node per operator, and one node per precondition of each operator. A link is made between each node representing a preconditions of an operator and the operator node, and between the node of each operator which satisfies a particular precondition and the node representing that precondition. A threat link is also added between the node of each operator which deletes a particular precondition and the node representing that precondition. Smith and Peot use these operator graphs before the planning process to discover when threat resolution may be postponed [50] and to analyze potential recursion [51]. Operator graphs do not apply to domains with conditional effects, and are less applicable to plan reuse and behavior modeling than other approaches, since they analyze operator interactions, not plans.

2.1.5 Partially Ordering Totally Ordered Plans

There has been some previous work on finding partial orderings of totally ordered plans. As previously mentioned, Regnier and Fade [41] used triangle tables to do this for plans without conditional effects. Veloso et al also presented a polynomial-time algorithm for finding a partial ordering of a totally ordered plan without conditional effects [54]. The algorithm adds links between each operator precondition and the most recent previous operator to add the condition. It then resolves threats and eliminates transitive edges. However, Bäckström shows that this method is not guaranteed to find the most parallel partial ordering, and that, in fact, finding the *optimal* partial ordering according to any metric is NP-complete [3].

2.1.6 Partial Order Planning

There has been a great deal of research into generating partially ordered plans from scratch. UCPOP [39] is one of the most prominent partial-order planners that can handle conditional effects. One of the strengths of UCPOP is its non-determinism; it is able to find all partially ordered plans that solve a particular problem. However, it is difficult to use the same technique to partially order a given totally ordered plan. The total order contains valuable information about dependencies and orderings, but the UCPOP method would discard this information and analyze the orderings from scratch. Not only is this inefficient, but it may result in a partial ordering of the totally ordered steps that is not consistent with the total order.

Graphplan [6], another well-known planner, is also able to find partially ordered plans in domains with conditional effects [2]. However, it produces suboptimal and non-minimal (overconstrained) partial orderings, which does not suit our purpose. Consider the plan in which the steps *op_a_1* ... *op_a_n* may run in parallel with the steps *op_b_1* ... *op_b_n*. Graphplan would find the partial ordering shown in Figure 1 because it only finds parallelism within an individual time step. In the first time step, *op_a_1* and *op_b_1* may run in parallel, but there is no other operator that may run in parallel with them, so Graphplan moves to the second time step (in which *op_a_2* and *op_b_2* may run in parallel). Graphplan constrains the ordering so that no operators from one time step may run in parallel with operators from another. None of the ordering constraints between *op_a* steps and *op_b* steps help achieve the goal, so they are not included in the partial ordering created by Sprawl, shown in Figure 2. Sprawl reveals the independence of the two sets of operators.

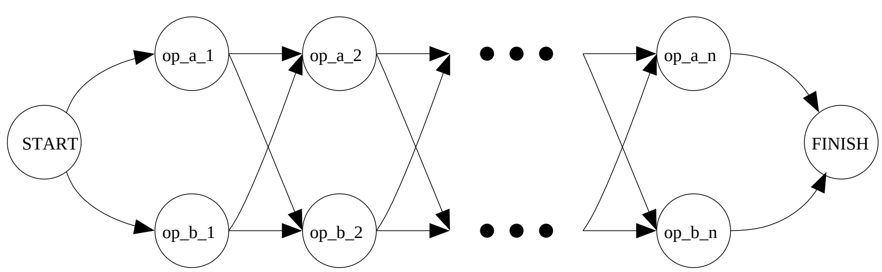

Figure 1: This partial ordering, found by Graphplan, contains many irrelevant ordering constraints.

2.2 Domain Knowledge to Reduce Planning Search

A large body of work has focussed on acquiring and using domain knowledge to reduce planning search. Our work seeks to avoid this search altogether by generating a domain-specific planning algorithm.

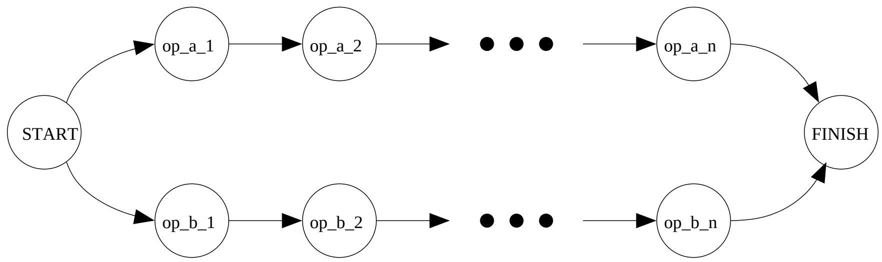

Figure 2: This partial ordering, found by SPRAWL, contains only necessary ordering constraints.

2.2.1 Control Rules

Much of the work on learning domain knowledge for planning is focused on learning control rules [33, 22, 12], which act as a search heuristic during the planning process by "recommending" at certain points which branch of the planning tree the planner should explore first. They do not reduce the complexity of the planning task, since they cannot *eliminate* branches of the search tree. They also capture only very local information (preference choices at specific branches of the planning search tree), ignoring common sequences of actions or repeated structures in example plans. However, they have been shown to reduce dramatically the planning time required for certain problems. Learning and managing control rules are the two biggest difficulties in using them. It is difficult for people to write good control rules, in part because one must know the problem-solving architecture of the planner in order to provide useful advice about how it should make choices [34], and computer-learned control rules are often ineffective [34]. Also, using control rules introduces a new problem for planners: when to create and save a new rule. Unrestricted learning creates a *utility* problem, in which learning more information can actually be counterproductive: it can take longer to search through a library of rules to find the ones that would help to solve a planning problem than to find the solution to the problem by planning from scratch [34].

2.2.2 Macro Operators

Another common type of learned control knowledge for planning is the macro operator [14, 27], which combines a frequently-occurring sequence of operations into one combined operator. A macro can then be applied by the planner in one step, thus eliminating the search required to find the entire sequence again. Macros have been shown to be very effective for reducing search in hierarchically decomposable domains, such as towers of Hanoi and sliding n-puzzle, because they can capture the common repeated sequences that make up almost every solution in these domains. However, Macros, like control rules, suffer from the utility problem: it is difficult to determine when to add a new macro operator, since adding too many can slow down the planning process. Each new macro operator adds a new branch to the planning tree at every search node. Although they can decrease the search depth, the added breadth can make planning searches slower, so, as with control rules, it is difficult to determine when to add a new macro operator. Some research has studied the problem of how to learn only the most useful macros [32], but the efficacy of macros has, in general, been limited to hierarchically decomposable domains.

2.2.3 Case-Based Reasoning

Another approach to learning planning knowledge, case-based reasoning, attempts to avoid generative planning entirely for many problems [16, 20, 29]. Entire plans are stored and indexed as *cases* for later retrieval. When a new problem is presented, the case-based reasoner searches through its case library for similar problems. If an exact match is found, the previous plan may be returned with no changes. Otherwise, the reasoner must either try to modify a previous case to solve the new problem or to plan from scratch. Utility is also a problem for case-based planners; many handle libraries of tens of thousands of cases [55], but, as with control rules, as the libraries get larger, the search times for relevant cases can exceed the time required to plan from scratch for a new case. Other difficulties with case-based planning include finding an appropriate similarity metric between problems, determining how to modify an existing plan to solve a new problem, and determining when it would be faster simply to plan for the new problem from scratch. However, case-based planners have succeeded in solving larger problems than can be solved by generative planning alone, and, in general, find solutions faster than generative planners [16].

2.2.4 Analogical Reasoning

A variant of case-based reasoning that deserves mention is analogical reasoning, which also stores case libraries and attempts to modify previous cases to solve new problems [55, 56]. However, in addition to storing the problem and the plan, analogical reasoners also store the problem-solving rationale behind each plan step. This makes it easier to modify previous cases to solve new problems. However, deciding when to abandon modification and plan from scratch is still a problem, as are retrieving cases from the library and determining whether to save new cases.

2.2.5 Hierarchical Planning

Domain knowledge is also used to hierarchically divide planning domains [43, 25, 26]. This allows the planner to simplify the problem by reasoning about it at a higher level of abstraction. The planner must then use the abstracted solution to find a solution to the full problem. The greatest difficulty with hierarchical planning is identifying a good abstraction hierarchy. Such a hierarchy will abstract away enough details that problem solving is easy at the highest level, but will not abstract away important details; the abstract solution must be easily transformable into a solution at lower levels of abstraction. Information about how to hierarchically divide planning problems may be user-supplied [43] or may be discovered automatically through an analysis of the planning domain [43, 25, 26].

2.2.6 Skeletal Planning

Skeletal planning combines hierarchical planning and case-based reasoning by storing previously generated plans and selectively abstracting them to reduce the size of the case library. The theory behind skeletal planning is that plans in a particular domain are divided into a limited number of plan classes and that a single abstract skeletal plan can represent the solutions for all plans in a given class [15]. Skeletal planning work does not assume that these classes are given to the problem solver. Instead, problem instances are encountered and stored in the case library. When several similar plans are encountered, they are generalized into a skeletal plan that is an abstracted version of all of them. Most skeletal planning work has relied on human users to do this generalization, but some work has been done on applying explanation-based learning to the automatic acquisition of skeletal plans [5].

2.2.7 Meta-Planning

Work in meta-planning seeks to allow planners to reason about the planning process as well as about the problem they are solving. Meta-planning is used both to reason explicitly about which planning steps to take next (such as working on a particular goal, refining an operator, or propagating a constraint) [52], to plan how to resolve goal conflicts or other problems in planning [59], or to explicitly formulate constraints about the kind of plan that should be found (one which uses the fewest resources, achieves the most goals, maximizes the value of the goals achieved, or avoids impossible goals) [59].

2.3 Automatic Program Generation

A great deal of work in many fields has addressed the problem of automatically generating programs. This work can be divided into two main classes: deductive program synthesis, in which programs are generated from specifications; and inductive program synthesis, in which, as in our work, programs are generated from example executions. Descriptions of work applying inductive program synthesis to planning and action selection follow a brief summary of general deductive and inductive program synthesis.

Deductive Program Synthesis Deductive program synthesis is the automatic generation of programs from specifications. Researchers have identified many different approaches to attacking this problem. Manna and Waldinger [31] approach this task as theorem proving: a user describes the input and the desired output of the algorithm and the program synthesis system uses a constructive proof of the algorithm's existence to identify the algorithm. Smith's KIDS system [49] solves program synthesis in a transformative way: after specifing types and basic functions over those types, the user writes program specifications in a formal language. The KIDS system then applies correctness-preserving transformations to those specifications to create an algorithm that fulfills them. Williams describes the TA system [60], which learns to generate programs that satisfy specifications by examining and modifying other programs.

Rich and Waters explore many of the difficulties of deductive program synthesis [42]. For example, although many claim that deductive program synthesis will become feasible for end users who are not familiar with programming, program synthesis systems require extensive domain knowledge in the form of types and functions over those types. Generating this domain knowledge requires familiarity with programming methods. Also, these systems depend on program specifications written by their users, so the specifications must be complete and correct in every situation for the resulting program to be correct. But writing complete and correct specifications is a difficult task, and Rich and Waters argue that user-written specifications actually cannot be complete; not only would completeness require that the user specified how the system should act in every possible situation, but it would also require the user to specify fuzzy requirements for efficiency and hard-to-formulate tradeoffs between different aspects of the generated system's performance.

Inductive Program Synthesis In part to sidestep the excessive demands of completely and correctly describing a desired system, many researchers have investigated inductive program synthesis, or the automatic generation of an algorithm from examples of its desired execution (either example traces of the execution or example input-output pairs). Programs generated inductively cannot be guaranteed to be correct or complete, but input-output pairs (or traces) are often easier to generate than complete and correct specifications, and the generated programs usually cover the "common cases" (the problems like those they are induced from).

Our work on extracting algorithmic models of behavior from observed executions falls within the category of inductive program synthesis (IPS). However, previously developed approaches to IPS are not immediately applicable to our problem. Some IPS algorithms induce programs from input-output pairs [36, 37]. In planning, this corresponds to inducing an algorithm from example initial and goal states, a formidable task. The problem is made easier in general IPS systems because they are also given a good deal of background information about programming techniques and methods to apply to a problem. This corresponds to background knowledge about how to solve different planning problems in a particular domain. We have chosen not to provide such knowledge.

Some approaches to IPS use program traces as input [4, 28], as does our work. However, the traces are annotated to mark the beginnings and ends of loops, to specify loop invariants and stopping conditions, to mark conditionals, etc. This kind of labelling cannot be obtained automatically from observed executions, so we do not allow it in our work.

Finally, whereas many approaches to IPS must attempt to induce the purpose of the steps from many examples [28], in our planning-based approach, the purpose of each step is automatically deduced via plan analysis. This information is critical to rapidly and correctly identifying the conditions for executing a sequence of steps or for terminating a loop.

2.3.1 Iterative and Recursive Macro Operators

Inductive program synthesis has been used to generate iterative and recursive macro operators [44, 45, 48, 47]. These macros capture repetative behavior and can drastically reduce planning search by encapsulating an arbitrarily long string of operators. However, this technique does not attempt to replace the generative planner, and so does not eliminate planning search.

2.3.2 Decision Lists for Planning

Some work has focussed on analyzing example plans to reveal a strategy for planning in a particular domain in the form of a decision list [24], or a list of condition-action pairs. Conditions consist of lists of predicates true in the current state and in the goal state and are of limited length. The decision list is created by transforming the example plans into a set of state-action pairs, enumerating all possible condition-action pairs, evaluating each based on coverage and accuracy on the state-action pairs extracted from the example plans, and adding the best pair (according to some quality criterion) to the decision list, eliminating covered state-action examples, until all examples are covered. Planning with the decision list consists of repeatedly checking the condition-action rules against the current state and goal. When a rule applies, the action is attempted, and matching begins again from the beginning of the decision list.

The decision list algorithm can find a strategy even when the plans it is given cannot be described by a simple strategy (e.g., they are optimal solutions to NP-hard problems). However, it is able to solve fewer than 50% of 20-block Blocksworld problems, and requires over a thousand state-action pairs to achieve that coverage [24]. Part of the reason it may need so many examples is that by breaking up example plans into state-action pairs, it also disposes of much of the information contained in the plans, such as sequencing and looping behaviors. Also, enumerating all possible conditions and evaluating them against every observed state-action pair may become prohibitively slow for more complex domains than Blocksworld.

2.4 Universal Planning

Some researchers have sought to avoid the planning search problem by aquiring and using "universal plans," or pre-computed functions that map state and goal combinations to actions. Our work can be seen as a new method of storing and acquiring universal plans.

The simplest form of a universal plan is a table with an entry for every possible situation (state and goal) that specifies the action to take in that situation, as in general solutions to Markov decision processes, or the Q tables of standard reinforcement learning [35]. In complex domains, the size of these tables is prohibitively large, as is the cost of acquiring them. Many methods have been developed to store this information more compactly.

Many researchers have used reinforcement learning to acquire compact Q functions rather than the prohibitively large Q tables. One common form of Q function is a neural network [1]. Q functions may also be represented as decision trees. This format has been found to be very expressive, frequently outperforming neural network representations [40], however, the decision tree format is less compressed than neural networks. Some work has been done on generating more compressed decision tree Q functions by avoiding the repetition of common substructures within the tree [53]. Relational reinforcement learning provides a method for learning a Q function represented as a logical regression tree over parameterized operators and predicates [11, 10]. This allows a solution to one problem to be used to solve another similar problem. However, Q functions learned in this way from small problems cannot be applied to larger or more complex problems, and it is unclear whether the size of the learned regression tree is prohibitive for interesting problems in complex domains.

Decision trees have also been used in a purely planning context. Schoppers suggests decision trees splitting on state and goal predicates [46], but finds these trees by conducting a breadth-first search for solutions—a method which is too time-consuming for most domains.

Other researchers have used Ordered Binary Decision Diagrams (OBDDs) to represent universal plans [8, 9, 18]. OBDDs provide an effective way to compress a universal plan without losing precision, however are currently generated via blind backwards search from goal states, a method that is impractical in complex domains.

3 Previous Work

3.1 Analyzing Plans with Conditional Effects

Analyzing example plans and executions is crucial for plan adaptation and reuse, e.g., [14], and could be useful for plan recognition and agent modeling, e.g., [23]. One of the most common approaches to plan analysis has been to create an *annotated ordering* of the example plan, e.g., [14, 41, 19, 20, 56], in which an ordered plan is supplemented with a rationale for the ordering constraints. Annotated orderings allow systems not only to reuse more flexibly portions of the plans they have observed, but also to reuse the reasoning that created those plans in order to solve new problems.

In recent years, the focus of the planning and agent modeling community has shifted from the simple STRIPS domain-specification language [13] toward richer languages like ADL [38] that capture the conditional effects of real-world actions. Despite the success of the annotated ordering approach for simple domain-specification languages, it has not been applied to plans with conditional effects.

In this paper, we introduce the SPRAWL algorithm for finding the rationale behind an observed totally ordered plan: the purpose for which each step is used in the plan and the reason behind each of the ordering constraints. We store this information in a structure we call a *minimal annotated consistent partial ordering* (MACPO). A consistent partial ordering $\mathcal{P}$ of a totally ordered plan $\mathcal{T}$ is one in which all *relevant* effects (those which affect the fulfillment of the goal) active in $\mathcal{P}$ are also active in $\mathcal{T}$. We call the partial orderings found by SPRAWL *minimal* because they do not include extraneous ordering constraints; each constraint either:

- provides a term upon which a relevant effect depends, or
- prevents a threat to such a term.

Finally, SPRAWL annotates each ordering constraint with the term the constraint provides or protects. Given an evaluation function for partial order quality, SPRAWL is capable of identifying the optimal MACPO of an observed total order.

We assume that we are given or that we observe a plan that is valid, i.e., all preconditions of the steps are satisfied, and, when executed, the plan produces the goal state. SPRAWL links the steps of the plan through the literals or terms that they support. Partial orderings are capable of representing these dependencies. [1] In addition, partial orderings can isolate independent sub-plans that can be reused or recognized separately, and they also identify potential parallelism.

We assume that observed example plans are totally ordered as plans of single executors. The annotations on the ordering constraints should *explain* the rationale behind the plans and allow portions of them easily to be matched, removed, and used independently.

Conditional effects make the task much more difficult because they cause the effects of a given step to change depending on what steps come before it, thus making step behavior difficult to predict. In fact, any ordering must treat each conditional effect in the plan in one of three ways:

- **Use:** make sure the effect occurs;
- **Prevent:** make sure the effect does not occur;
- **Ignore:** don't care whether the effect occurs or not.

Figure 3 illustrates three totally ordered plans that demonstrate these cases. Note that all three plans have the same initial state and the same operators. We are able to demonstrate all three cases by changing only the goals. The preconditions (pre) are listed, as are the effects, which are represented as conditional effects $\{a\} \rightarrow b$, *i.e.*, if a then add b. A non-conditional effect that adds a literal b is then represented as $\{\} \rightarrow b$. Delete effects are represented as negated terms (*e.g.*, $\{a\} \rightarrow NOT b$). In the first plan, the conditional effect

[1] A partial order is a precedence relation $\preceq$ with the following three properties 1) reflexivity: $a \preceq a$; 2) non-symmetric (no cycles): if $a \preceq b$ then not $b \preceq a$, unless $a = b$; and 3) transitivity: if $a \preceq b$ and $b \preceq c$, then $a \preceq c$. The relation is a "partial" order because there may be incomparable elements: i.e., elements a, b such that neither $a \preceq b$ nor $b \preceq a$. Note that a DAG is a partial order if we define $a \preceq b$ as a path from a to b.

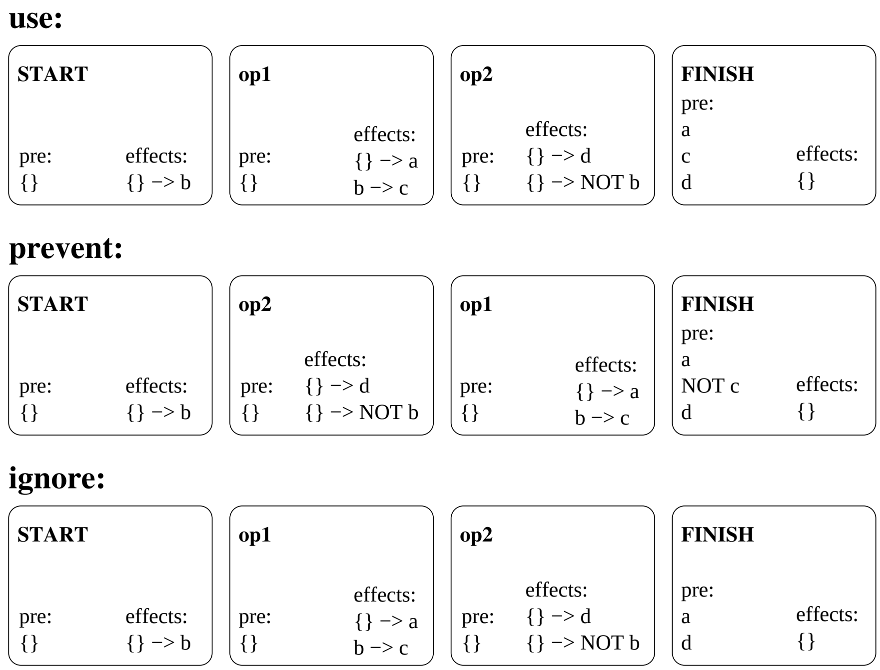

Figure 3: Three totally ordered plans that illustrate the three possible ways of treating a conditional effect in an ordering: using it to achieve a goal, preventing it in order to achieve a goal, or ignoring its effect.

of op1 is *used* to generate the goal term c. In the second plan, it is *prevented* from generating the term c, and in the third plan, the effect is irrelevant, so it is *ignored*.

Figure 4 shows the annotated partial orderings generated by SPRAWL for each of these cases. The ordering constraints are annotated with a rationale explaining why they are necessary. Although the plans for these three cases are composed of the same steps, SPRAWL reveals that the partial orderings are very different. In the "use" case, SPRAWL identifies that op2 threatens the goal term c, which is created by op1, and enforces the ordering op1 $\rightarrow$ op2 to protect c. In the "prevent" case, SPRAWL identifies that the step op1 must not be able to execute the conditional effect that adds the term c, and so ensures that the condition of this effect, the term b, is not true before the step executes. In this way, SPRAWL finds the ordering constraint op2 $\overset{NOTb}{\rightarrow}$ op1. It also finds that the START step, since it adds b, is a threat to this link, and must therefore come before op2. Finally, SPRAWL identifies that, in the "ignore" case, the conditional effect is irrelevant, so op1 and op2 may run in parallel.

Treating *any* conditional effect in a plan in a different way will result in a different partial ordering, creating exponentially (in the number of conditional effects) many partial orderings, many of which may be invalid. [2] One way to deal with this difficulty is to insist that exactly the same conditional effects must be active in the partial ordering as are active in the totally ordered plan, but this will result in an overly restrictive partial ordering in which some ordering constraints may not contribute to goal achievement. Instead, we analyze the totally ordered plan to discover which conditional effects are relevant. This allows us to ignore incidental conditional effects in the totally ordered plan.

Instead of finding the optimal partially ordered plan to solve a given problem, we chose to focus on finding optimal partial orderings *consistent* with given totally ordered plan, or those in which all relevant effects were also active in the total ordering. There are two reasons for this. The first is that the totally ordered plan contains a wealth of valuable information about how to solve the problem, including which operators to use and which conditional effects are relevant. Using this information reduces the search required to solve the problem. The second is that for many applications, including plan modification and reuse and agent

[2]The number of possible partial orderings is also exponential in the number of steps and the number of conditions on each step.

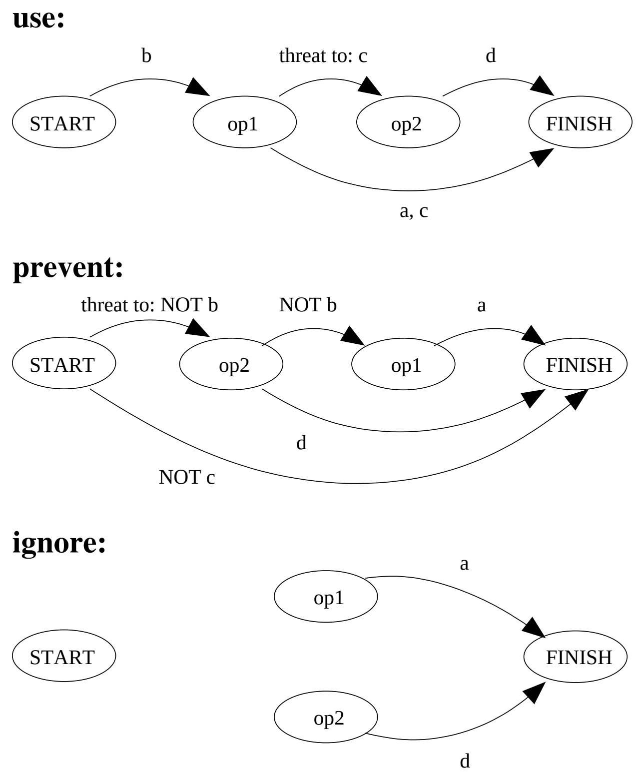

Figure 4: The annotated partially orderings generated by SPRAWL for the three totally ordered plans shown in Figure 3.

modeling, it is important to be able to analyze an observed or previously generated plan (for example, to find characteristic patterns of behavior or to identify unnecessary steps).

3.1.1 Needs Analysis

Needs analysis, the first step of the SPRAWL algorithm, computes a tree of needs for the totally ordered plan. We first create a goal step called FINISH with the terms of the goal state as preconditions. Needs analysis calculates which terms need to be true before the last step in the plan in order for the preconditions of FINISH to be true afterward. Then it calculates which need to be true before the second-to-last plan step in order for *those* terms to be true. This calculation is executed for each step of the plan, starting from the last step and finishing at the START step, creating a tree of "needs." This needs tree allows us to identify the relevant effects of a given step and most of the dependencies in the plan. However, not all threats are identified in Needs Analysis; SPRAWL uses the needs tree to calculate the remaining threats.

Needs Tree Structure In this section, we will discuss the needs that compose the needs tree as well as the structure of the tree. The needs tree consists of three kinds of needs:

1. **Precondition Needs:** the preconditions of a step are called *precondition needs* of the step—they must be true for the step to be executable. For example, the precondition needs of the FINISH step are the goals of the plan.

2. **Existence Needs:** terms that must be true before a step n in order for n to create a particular term or to maintain a previously existing term are called *existence needs* of the term at the step n. In the "use" example in Figure 4, one existence need of the term c at the step op1 is b, since op1 will generate c if b is true before it executes.

3. **Protection Needs:** terms that must be true before step n in order for n not to delete a particular term are called *protection needs* of the term at the step n. In the "prevent" example in Figure 4, one protection need of the term NOT c at the step op1 is NOT b, since if NOT b is not true before step op1, then op1 will add c (thereby deleting NOT c).

For the sake of simplicity, instead of abstract plan steps, we will illustrate the three kinds of needs using plan steps from a domain in which we have a sprinkler that, if on, can wet the yard as well as any object that may be in the yard. Figure 5 shows the operator sprinkle front-yard. The term on sprinkler is a *precondition need* of the step sprinkle front-yard.

sprinkle front–yard	
	effects:
pre:	{} –> wet front–yard
on sprinkler	at ?obj front–yard –> wet ?obj

Figure 5: The step sprinkle front-yard.

To illustrate *existence needs*, let us assume that, after executing the step sprinkle front-yard, wet shoe must be true. This could be accomplished in two ways:

- by ensuring that at shoe front-yard was true before sprinkle front-yard executed, or
- by ensuring that wet shoe was already true before sprinkle front-yard executed, as shown in Figure 6. [3]

These two terms are called *existence needs* of wet shoe at the step sprinkle front-yard, since they provide ways for the term wet shoe to be true after the step sprinkle front-yard.

We must also make a distinction between *maintain* existence needs and *create* existence needs. [4] As mentioned above, there are two ways to ensure that wet shoe is true after the execution of the step sprinkle front-yard, both illustrated in Figure 6. One way is for wet shoe to have been true previously. We call this a *maintain* existence need since the step does not generate the term, but simply maintains a term that was previously true. However, the step sprinkle front-yard could generate the term wet shoe if at shoe front-yard were true before the step executed. We call this an *create* existence need, since we have introduced a new need in order to satisfy another.

Note that, because there may be multiple ways to ensure the existence of a term, the description of needs must include the OR logical operator, as shown in Figure 6. It must also include the AND logical operator, since we allow a conditional effect to have multiple conditions, and in order to guarantee that the effect occurs, we must be able to specify that all must be true.

To illustrate *protection needs*, assume that, after executing the step sprinkle front-yard, the term NOT wet shoe must be true. In order to protect the term NOT wet shoe, we must ensure that NOT at shoe front-yard is true before sprinkle front-yard executes. This is called a *protection need* because it protects the term from being deleted (i.e., prevents wet shoe from being added).

It is not always necessary to generate new needs to satisfy a need term; it may also be satisfied if a non-conditional effect of the step satisfies it, as illustrated in Figure 7. We call such needs *accomplished*, and indicate this in our diagrams with a double circle.

Needs Analysis Algorithm The needs analysis algorithm is shown in Table 1. We now describe in detail how needs analysis generates the needs of an individual term. Each needed term t must be created and protected from deletion; we represent this as two branches of needs: existence needs and protection needs. As explained previously, t's existence needs at a particular step n are terms which must be true before step

[3] In the remainder of the sprinkler examples, we abbreviate the literals sprinkler as sp, front-yard as fy, back-yard as by, and shoe as sh.

[4] Precondition needs and protection needs are always *create* needs.

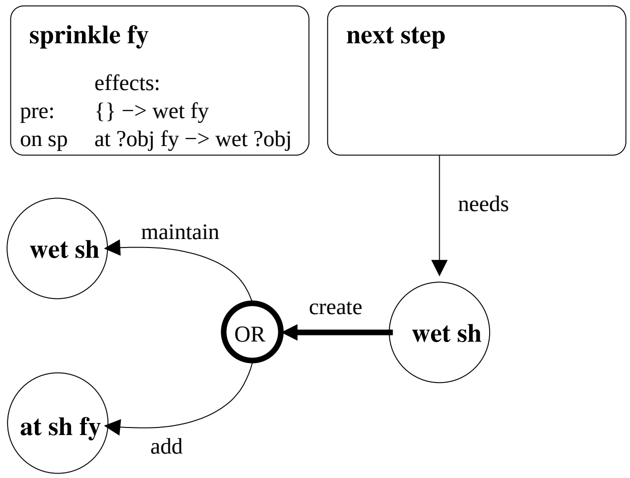

Figure 6: Expanding the need wet shoe in the step sprinkle front-yard. The term wet shoe may be satisfied in either of two ways; this is represented by an OR operator.

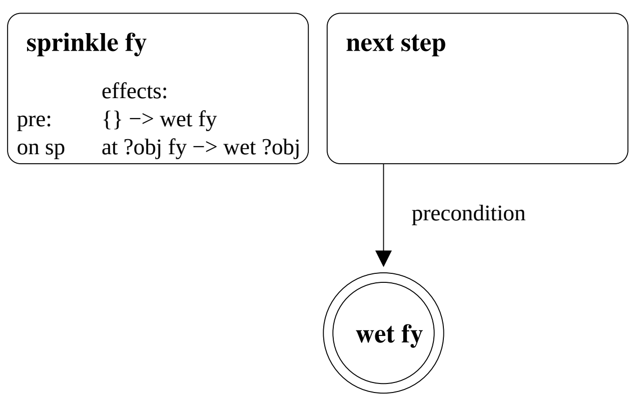

Figure 7: A term may be true after a particular step if a non-conditional effect of the previous step accomplishes it. We indicate this with a double circle around the term.

n to ensure that t is true after step n. There are two possibilities for existence needs: either t may have been true before step n, or a conditional effect of step n may generate t [5]. The protection needs of t at step n are terms which must be true before step n to ensure that step n does not delete t. Prevention needs are therefore negated conditions of any conditional effects of step n that delete t. [6] Figure 8 illustrates the needs tree created to satisfy each needed term.

```
Input: A totally ordered plan T = S_1, S_2, ..., S_n,
       the START operator S_0 with add effects set to the
       initial state, and the FINISH operator S_n + 1 with
       preconditions set to the goal state.
Output: A needs tree N.

procedure Needs_Analysis(T, S_0, S_n + 1):
    for c ← n+1 down-to 1 do
        for each precond of S_c do
            Expand_Term(c, precond)

procedure Expand_Term(c, term):
    Find_Existence(c, term)
    Find_Protection(c, term)

procedure Find_Existence(c, term):
    for each conditional effect of S_c do
        if effect unconditionally adds term then
            term.accomplished ← true
        otherwise if effect conditionally adds term then
            Add_Conditions_To_Existence_Needs(effect, term)
            for each condition of effect do
                Expand_Term(c-1, condition)

procedure Find_Protection(c, term):
    for each conditional effect of S_c do
        if effect unconditionally deletes term then
            term.impossible ← true
            return
        otherwise if effect conditionally deletes term then
            Add_Conditions_To_Protection_Needs(effect, term)
            for each condition of effect do
                Expand_Term(c-1, condition)
```

Table 1: Needs Analysis algorithm.

We will use the totally ordered plan from the sprinkler domain shown in Figure 9 to illustrate the behavior of the needs analysis algorithm. First, the algorithm will analyze the last plan step (sprinkle front-yard), which has one precondition need (on sprinkler), to determine how to satisfy the needs of the subsequent step FINISH (wet shoe and wet front-yard). As previously discussed, there are two ways for the step sprinkle front-yard to satisfy wet shoe: either wet shoe could be true before this step executes, or at shoe front-yard must be true before this step executes. So the needs of the term wet shoe are *maintain* wet shoe OR *create* at shoe front-yard. As for wet front-yard, the other precondition need of the FINISH step, it is accomplished by the step sprinkle front-yard since it is a non-conditional effect of the step. However, the algorithm continues to look for other ways to accomplish the term. Since there are no conditional effects of sprinkle front-yard that

[5] Non-conditional effects of step n that add t do not add needs—nothing needs to be true before step n in order for them to occur

[6] If t is deleted by a non-conditional effect of step n, then we call it *unsatisfiable* and end its branch of the needs tree.

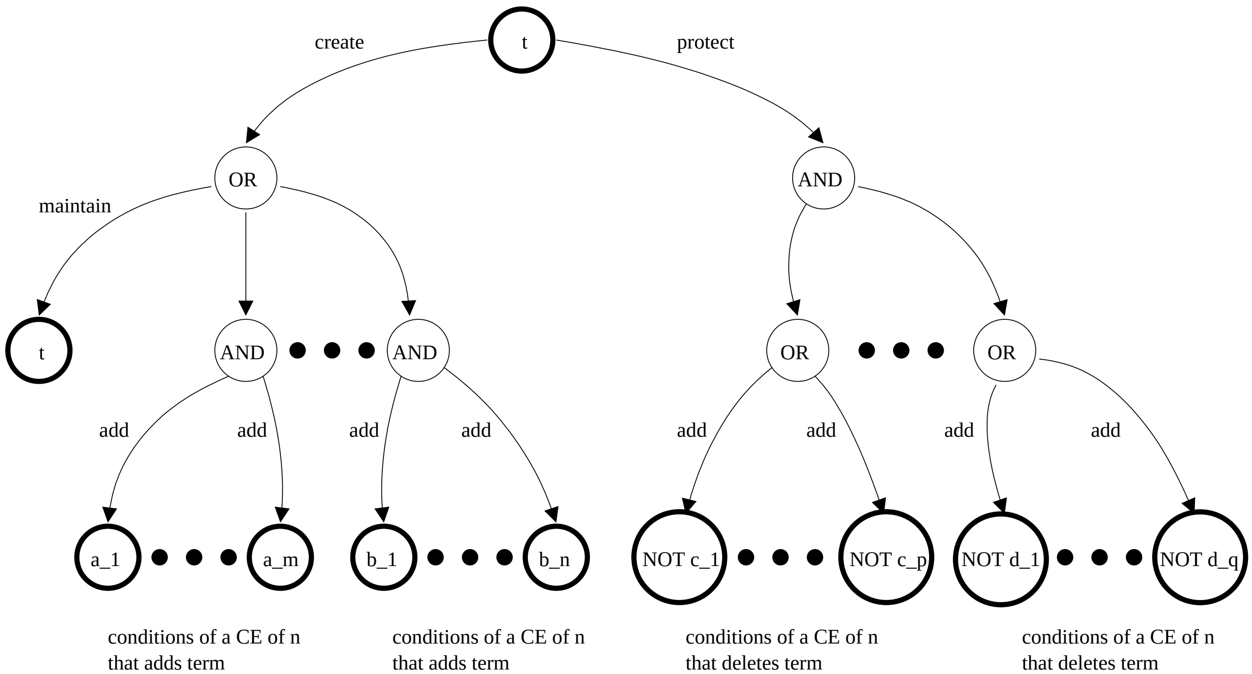

Figure 8: The existence needs of a need at a particular step n are calculated by finding all possible ways it can be generated in the previous step and ensuring that at least one of these occurs. The protection needs are calculated by finding all possible ways it can be deleted in the previous step and ensuring that none of these occurs.

either generate or delete wet front-yard, the algorithm just adds the maintain existence need, *maintain* wet fy.

Next, the algorithm moves back to the previous plan step, move shoe back-yard front-yard, which has the precondition need at shoe back-yard. The needs carried over from previous steps are *maintain* wet shoe OR *create* at shoe front-yard, the existence needs of wet shoe from the FINISH step; *maintain* wet front-yard, the existence need of wet front-yard from the FINISH step; and on sprinkler, the precondition need of the step sprinkle front-yard. The term at shoe front-yard is a non-conditional effect of this step, so it is accomplished, but, as with wet fy in the previous step, the algorithm adds a maintain existence need (*maintain* at shoe front-yard) in order to find other ways to accomplish the term. The terms *maintain* wet shoe, *maintain* wet front-yard, and on sprinkler cannot be prevented or created by this step, so each is satisfied by a maintain existence need (*maintain* wet shoe, *maintain* wet front-yard, and *maintain* on sprinkler).

Finally, the algorithm reaches the initial state, or START step, and is able to determine which branches of the needs tree can be accomplished and which can not. The remaining branches of the tree are at shoe back-yard, *maintain* at shoe front-yard, *maintain* wet shoe, *maintain* wet front-yard, and *maintain* on sprinkler. Two of the needs, at shoe back-yard and *maintain* on sprinkler are accomplished by the START step. However, all of the other remaining needs are not accomplished by the START step. We call these needs *unsatisfiable* and indicate this in our diagrams with a dashed circle.

The complexity of needs analysis is exponential in the number of conditional effects and the bound on the number of conditions in each effect, or $O(mP(EC)^n)$, where m is the number of steps without conditional effects, n is the number of steps with conditional effects, P is the bound on the number of preconditions, E is the bound on the number of conditional effects in each step, and C is the bound on the number of conditions per conditional effect. The complexity of needs analysis on a plan with no conditional effects is linear: $O(mP)$.

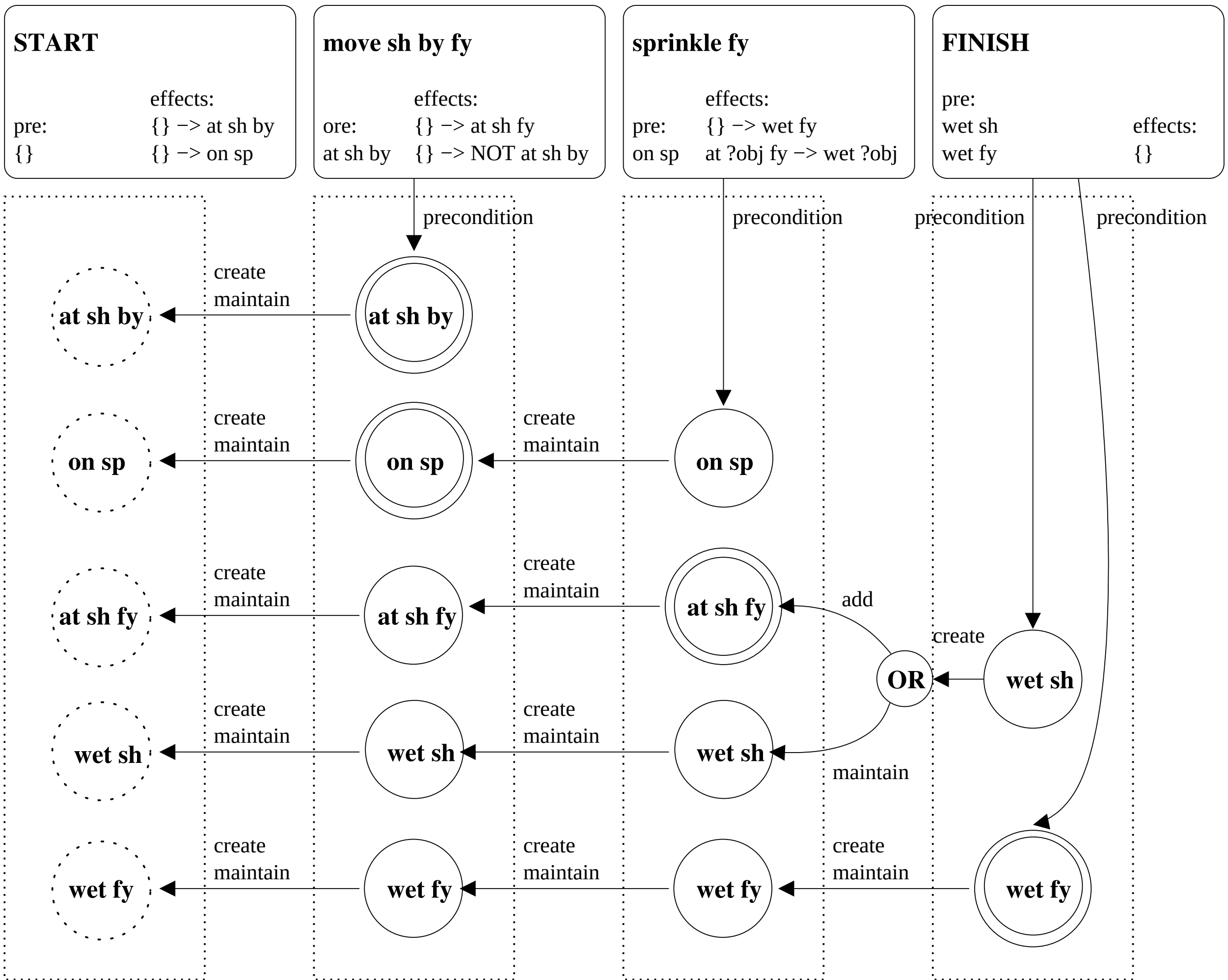

Figure 9: A totally ordered plan in the sprinkler domain and its complete needs tree.

3.1.2 The Sprawl Algorithm

Table 2 shows the Sprawl partial ordering algorithm. Sprawl performs needs analysis, then performs a depth-first search on the needs tree, adding causal links in the partial ordering between steps that need terms and the steps that generate them.

Resolving Threats We rely heavily on the totally ordered plan to help us resolve threats. There are three ways to resolve threats in a plan with conditional effects, as described in [58]:

1. **Promotion** moves the threatened operators before the threatening operator;
2. **Demotion** moves the threatened operator after the threatening operator;
3. **Confrontation** may take place when the threatening effect is conditional. It adds preconditions to the threatening operator to prevent the effect causing the threat from occurring.

To find all possible partial orderings, all these possibilities should be explored. However, since we are provided the totally ordered plan, we do not need to search at all to find a feasible way to resolve the threat; we can simply resolve it in the same way it was resolved in the totally ordered plan. In fact, if threats are resolved in a different way, then the resulting partial ordering would not be consistent with the totally ordered plan.

Input: A totally ordered plan $\mathcal{T} = S_1, S_2, \ldots, S_n$,
the START operator S_0 with add effects set to the initial state, the FINISH operator $S_n + 1$ with preconditions set to the goal state, and an evaluation function F, to determine the value of partial orderings.

Output: An optimal minimal annotated consistent partially ordered plan shown as a directed graph $\mathcal{P}$.

```
procedure SPRAWL(𝒯, S_0, S_n + 1, F):
    tree ← Needs_Analysis(𝒯, S_0, S_n + 1)
    tree ← Trim_Unaccomplished_Need_Tree_Branches(tree)
    queue ← tree.precondition_Needs
    𝒫 ← Find_MaxEval_MACPO(tree, queue, F, null)

procedure Find_MaxEval_MACPO(tree, queue, F, 𝒫):
    best_PO ← 𝒫
    best_val ← F(best_PO)
    if queue.empty() then
        return Resolve_Threats(𝒫)
    else
        need ← queue.top()
        for each way of satisfying need do
            if this way is via a CE then
                queue.add(CE.conditions)
            queue.add(need.Protection)
            new_PO ← Find_MaxEval_MACPO(tree, queue, F, 𝒫
                + Causal_Link(satisfying step, need.step, need))
            new_val ← F(best_PO)
            if new_val ¿ best_val then
                best_PO ← new_PO
        return best_PO

procedure Handle_Threats(tree, 𝒫):
    for each causal link S_i → S_j do
        for c ← 1 up-to i − 1 do
            if Threatens(S_c, S_i → S_j) then
                DEMOTE: Add_Causal_Link(S_c, S_i, 𝒫)
        for c ← j + 1 up-to n
            if Threatens(S_c, S_i → S_j) then
                PROMOTE: Add_Causal_Link(S_j, S_c, 𝒫)
```

Table 2: The SPRAWL algorithm.

If, in the totally ordered plan, the threatening operator occurs before the threatened operators, then promotion should be used to resolve the threat in the partial ordering. Similarly, if it occurs after the threatened operators, demotion should be used to resolve the threat in the partial ordering. If the threatening operator occurs between the threatened operators in the totally ordered plan, then we know that confrontation must have been used in the totally ordered plan to prevent the threatening conditional effect from occurring. Needs analysis takes care of confrontation with *protection needs*, shown in Figure 8, which ensure that steps that occur between a needed term's creation and use in the totally ordered plan do not delete the term.

3.1.3 Discussion

The SPRAWL algorithm does not create a partially ordered plan from scratch; its purpose is to create an annotated partial ordering of the steps of a given totally ordered plan to aid in our understanding of the structure of the plan. Because of this, we restrict SPRAWL to partial orderings consistent with the totally ordered plan.

However, frequently, there are many partial orderings consistent with the totally ordered plan. SPRAWL searches through these possibilities to find the optimal partial ordering. Here, we discuss the space of possibilities explored by SPRAWL and discuss a polynomial solution for finding a suboptimal minimal annotated consistent partial ordering.

Different Total Orderings of the Same Steps May Produce Different Partial Orderings In some cases, a different total ordering of the same plan steps would produce a different partial ordering, but these are cases in which the relevant effects differ. For example, the use and prevent cases shown in Figure 3 consist of the same initial states and the same operators. However, the relevant effects differ. SPRAWL would never produce the same partial ordering for both of them; the partial orderings would each preserve the same relevant effects as are active in the respective totally ordered plans. The minimal annotated consistent partial orderings found by SPRAWL are shown in Figure 4.

Active Conditional Effects May Differ from Those in Totally Ordered Plan Though SPRAWL is restricted to partial orderings consistent with the totally ordered plan it is given, this does not mean that all conditional effects active in the totally ordered plan must be active in the partial ordering, or vice versa. There are sometimes irrelevant conditional effects in the totally ordered plan or in the partial ordering, and SPRAWL does not seek to maintain or prevent these irrelevant effects. The ignore case shown as a totally ordered plan in Figure 3 demonstrates this. In this problem, one of the active conditional effects in the totally ordered plan is the effect $\mathsf{b} \rightarrow \mathsf{c}$ from step $op1$. However, this effect does not affect the fulfillment of the goal state, and so is not a relevant effect. In fact, as is shown in Figure 4, SPRAWL would enforce no ordering constraints between the two steps in its partial ordering. Though the different orderings produce different final states, the goal terms are true in each of these final states, so it doesn't matter which occurs.

Finding Multiple Partial Orderings Although, as we discussed, SPRAWL is restricted to partial orderings with no relevant effects not active in the given totally ordered plan, this does not mean that all relevant effects in the totally ordered plan must be relevant effects in the partial ordering. Thus, there could be several possible minimal annotated consistent partial orderings.

Sometimes, there are several relevant effects in the totally ordered plan that achieve the same aim. Bäckström presented an example that neatly illustrates this [3]. The totally ordered plan is shown with its needs tree in Figure 10. In this plan, two different relevant effects provide the term q to step c—both step a and step b generate q. Choosing a different relevant effect to generate q creates a different partial order. The two partial orders representing each of the two relevant effect choices are shown in Figures 11 and 12.

The needs analysis algorithm shown in Table 1 produces a needs tree that encompasses all possible partial orderings consistent with the totally ordered plan, and SPRAWL searches through these orderings to identify the optimal one according to a given measure. However, finding the optimal partial ordering "under reasonable optimality criteria" has been shown to be NP-hard [3]. In Table 3, we provide a polynomial-time algorithm for finding one (not necessarily optimal) minimal annotated consistent partial ordering. This algorithm is a variation on the one presented by [54], however, in order to handle conditional effects, we must calculate the state between each step to determine whether the conditional effects were active in the totally-ordered plan.

3.2 Automatically Acquiring Planning Templates from Example Plans

Planning is a powerful tool for action selection, since it offers a guarantee that a proposed plan achieves an agent's goals. If efficient, agents could re-plan to deal with unexpected situations. However, general-purpose planning is too slow to use in most real-time systems and does not scale to large problems. In order for

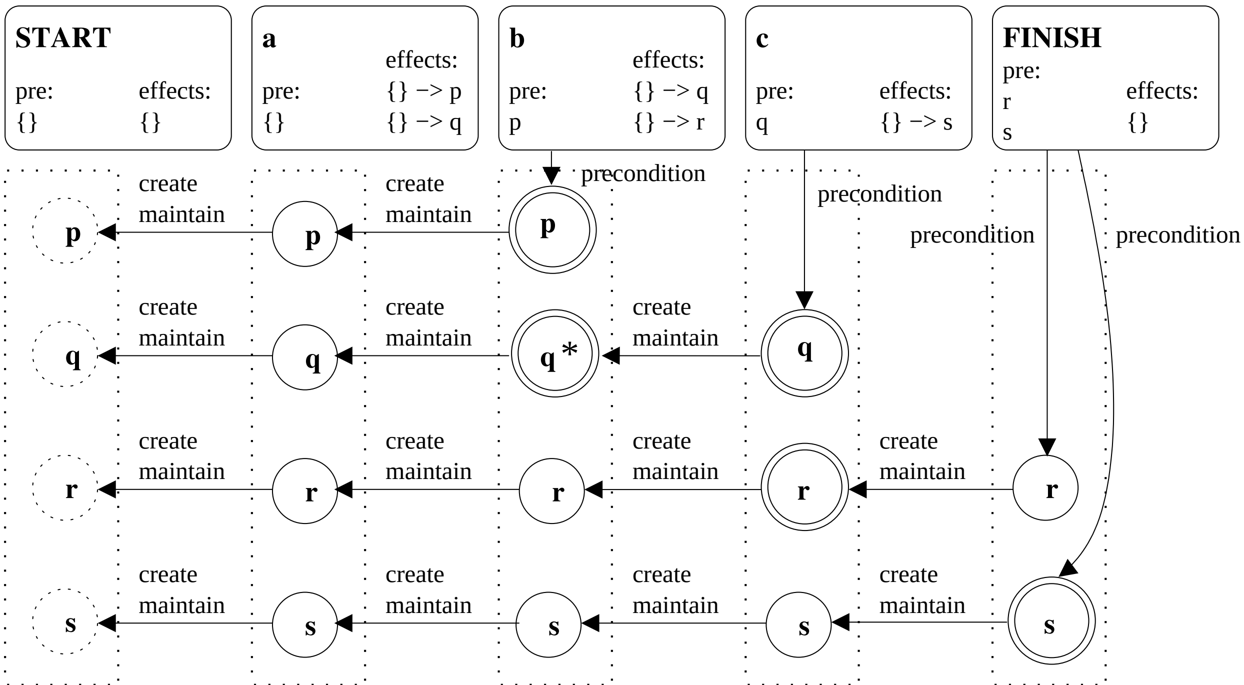

Figure 10: Bäckström's example plan and the needs tree created by needs analysis. Note that the term `q` is accomplished by two different steps: `a` and `b`. This means that two partial orderings are possible: one in which step `a` provides `q` to step `c`, and one in which `b` does.

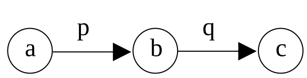

Figure 11: One possible partial ordering of Bäckström's example plan.

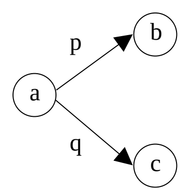

Figure 12: Another partial ordering of Bäckström's example plan, found by SPRAWL when the evaluation function favors shorter partial orderings.

planning to be feasible in these situations, some knowledge about the domain being solved must be used in the planning process, either by using a domain-specific planner or by using domain-specific knowledge to narrow the search.

Many researchers have focused on learning domain-specific control knowledge for planning automatically, usually in the forms of control rules, macro operators, and plan case libraries. There have also been several efforts focusing on writing domain-specific planners to quickly solve planning problems in particular domains without resorting to generative planning. These programs are currently handwritten, but this process is tedious and often quite difficult.

We introduce the DISTILL algorithm, which automatically extracts these domain-specific planning programs (which we call *templates*) from example plans, and show how to use them to solve planning problems. We call these domain-specific planning programs *templates*. Table 4 shows a simple example template that solves all problems in the gripper domain that involve moving balls from one room to another.

In some domains, finding optimal solutions is NP-complete. Therefore, templates learned automatically from a finite number of example plans cannot be guaranteed to find optimal plans. Our goal is to extend the *solvability horizon* for planning by reducing planning times and allowing much larger problem instances

Input: A totally ordered plan $\mathcal{T} = S_1, S_2, \ldots, S_n$,
the START operator S_0 with add effects set to the initial state, and the FINISH operator $S_n + 1$ with preconditions set to the goal state.
Output: A minimal annotated consistent partially ordered plan shown as a directed graph $\mathcal{P}$.

```
procedure Find_Any_MACPO(T, S_0, S_n + 1):
1.  Calculate intermediate states T_1..T_n + 1,
        where T_i is the state after step S_{i-1}
2.  Use states T_1..T_n + 1 to identify CEs active in T
3.  for step S_i ← S_n + 1 downto S_1 do
4.      for all P_{i_j} preconditions of S_i do
5.          Use states T_1..T_i to find last producer of P_{i_j}
6.          Add_Causal_Link(producer, S_i, P_{i_j}, P)
7.          if P_{i_j} was produced via a CE then
8.              add conditions of CE to preconditions of producer step
9.  return Resolve_Threats(P)
```

Table 3: A polynomial-time algorithm for finding one (not necessarily optimal) MACPO

```
while (in_goal_state (at(?1:ball ?2:room)) and
        in_current_state (at(?1:ball ?3:room)) and
        not same (?2:room ?3:room) and
        in_current_state (at-robby(?5:room))) do
    if (not same (?3:room ?5:room)) then
        move(?4 ?5 ?3)
    pick(?1 ?4 ?3)
    move(?4 ?3 ?2)
    drop(?1 ?4 ?2)
```

Table 4: A simple template that solves all gripper-domain problems involving moving balls from one room to another.

to be solved. We believe that post-processing plans can help improve plan quality.

Our work on the Distill algorithm for learning templates focuses on converting new example plans into templates in if-statement form and merging them, where possible. Our results show that merging templates produces a dramatic reduction in space usage compared to case-based or analogical plan libraries. We also show that by constructing and combining the if statements appropriately, we can achieve automatic *situational generalization*, which allows templates to solve problems that have not been encountered before without resorting to generative planning or requiring adaptation.

We first formalize the concept of templates. Next, we present our novel Distill algorithm for learning templates from example plans and present our results. We then discuss how to use templates to solve planning problems.

3.2.1 Defining Templates

A template is a domain-specific planning program that, given a planning problem (initial and goal states) as input, either returns a plan that solves the problem or returns failure, if it cannot do so. Templates are composed of the following programming constructs and planning-specific operators:

- **while** loops;

- **if** , **then** , **else** statements;
- logical structures (**and** , **or** , **not**);
- **in_goal_state** , **in_current_state** , **in_initial_state** operators;
- **same** operator;
- plan predicates; and
- plan operators.

In order for templates to capture repeated sequences in while loops and to determine that the same sequence of operators in two different plans has the same conditions, they must update a current state as they execute by simulating the effects of the operators they add to the plan. Without this capability, we would be unable to use such statements as: **while** (condition holds) **do** (body). Therefore, in order to use a template, it must be possible to simulate the execution of the plan. However, since template learning requires full models of the planning operators, this is not an additional problem.

Table 4 shows a template that solves all gripper-domain [30] problems involving moving balls between rooms. The template is composed of one while loop: while there is an ball that is not at its goal location, move to the ball (if necessary), pick up the ball, move to goal location of the object, and drop the ball.

3.2.2 Learning Templates: the Distill Algorithm

The Distill algorithm, shown in Table 5, learns templates from sequences of example plans, incrementally adapting the template with each new plan. One benefit of online learning is that it allows a learner with access to a planner to acquire templates on the fly in the course of its regular activity. And because templates are learned from example plans, they reflect the *style* of those plans, thus making them suitable not only for planning, but also for agent modeling.

Distill can handle domains with conditional effects, but we assume that it has access to a complete model of the operators and to a minimal annotated partial ordering of the observed total order plan. Previous work has shown that operator models are learnable through examples and experimentation [7, 57] and has shown how to find minimal annotated partial orderings of totally-ordered plans given a model of the operators [61].

The Distill algorithm converts observed plans into templates (see “Converting Plans into Templates”) and merges them by finding templates with overlapping solutions and combining them (see “Merging Templates”). In essence, this builds a highly compressed case library. However, another key benefit comes from merging templates with overlapping solutions: this allows the template to find *situational generalizations* [17] for individual sections of the plan, thus allowing it to reuse those sections when the same situation is encountered again, even in a completely different planning problem.

Generalizing Situations We make several assumptions about what makes one planning *situation* different than another, and about how the observed planner will solve problems. We assume that two objects of the same type will be treated the same by the planner. Thus, two situations are equivalent if they contain the same number and types of objects in the same relationships. We assume that the planner will respond to equivalent situations with the same plan. This allows the Distill algorithm to identify common situations that occur in the solutions of several planning problems, and to extract their solutions for independent use in other problems.

Converting Plans into Templates The first step of incorporating an example plan into the template is converting it into a parameterized if statement. First, the entire plan is parameterized. Distill chooses the first parameterization that allows part of the solution plan to match that of a previously-saved template. If no such parameterization exists, it randomly assigns variable names to the objects in the problem. [7]

Next, the parameterized plan is converted into a template, as formalized in the procedure Make_New_If_Statement in Table 5. The conditions on the new if statement are the initial- and goal-state terms that are *relevant* to the plan. Relevant initial-state terms are those which are needed for the plan to run correctly and achieve

[7]Two discrete objects in a plan are never allowed to map onto the same variable. As discussed in [14], this can lead to invalid plans.

Input: Minimal annotated consistent partial order $\mathcal{P}$,
current template T_i.
Output: New template T_{i+1}, updated with $\mathcal{P}$

```
procedure DISTILL (P, T_i):
    A ← Find_Variable_Assignment(P, T_i.variables, ∅)
    until match or can't match do
        if A = ∅ then
            can't match
        else
            N ← Make_New_If_Statement(Assign(P, A))
            match ← Is_A_Match(N, T_i)
        if not can't match and not match then
            A ← Find_Variable_Assignment(P, T_i.variables, A)
    if can't match then
        A ← Find_Variable_Assignment(P, T_i.variables, ∅)
        N ← Make_New_If_Statement(Assign(P, A))
    T_{i+1} ← Add_To_Template(N, T_i)

procedure Make_New_If_Statement(P_A):
    N ← empty if statement
    for all terms t_m in initial state of P_A do
        if exists a step s_n in plan body of P_A such that
                s_n needs t_m or goal state of P_A needs t_m then
            Add_To_Conditions(N, in_current_state (t_m))
    for all terms t_m in goal state of P_A do
        if exists a step s_n in plan body of P_A such that
                t_m relies on s_n then
            Add_To_Conditions(N, in_goal_state (t_m))
    for all steps s_n in plan body of P_A do
        Add_To_Body(N, s_n)
    return N

procedure Is_A_Match(N, T_i):
    for all if-statements I_n in T_i do
        if N matches of I_n then
            return true

procedure Add_To_Template(N, T_i):
    for all if-statements I_n in T_i do
        if N matches I_n then
            I_n ← Combine(I_n, N)
            return
    if N is unmatched then
        Add_To_End(N, T_i)
```

Table 5: The DISTILL algorithm: updating a template with a new observed plan.

the goals [55]. Relevant goal-state terms are those which the plan accomplishes. We use a minimal annotated partial ordering [61] of the observed plan to compute which initial- and goal-state terms are relevant. The steps of the example plan compose the body of the new if statement. We store the minimal annotated partial ordering information for use in merging the template into the previously-acquired knowledge base.

Figure 13 shows an example minimal annotated partially ordered plan with conditional effects. Table 6 shows the template DISTILL creates to represent that plan. Note that the conditions on the generated if statement do not include all terms in the initial and goal states of the plan. For example, the template does not require that e(z) be in the initial and goal states of the example plan. This is because the plan steps do not generate e(z), nor do they need it to achieve the goals. Similarly, b(x) and the conditional effects that could generate the term c(x) or prevent its generation are also ignored, since it is not relevant to achieving the goals.

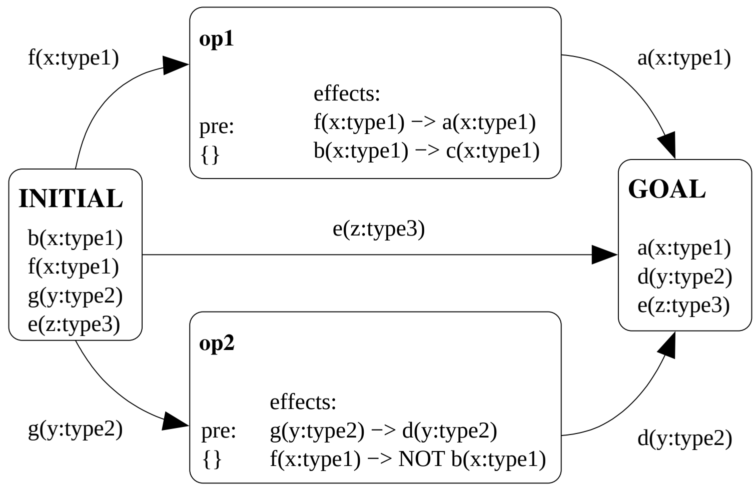

Figure 13: An example plan. The preconditions (pre) are listed, as are the effects, which are represented as conditional effects $a \rightarrow b$, i.e., if a then add b. A non-conditional effect that adds a literal b is then represented as $\{\} \rightarrow b$. Delete effects are represented as negated terms (e.g., $\{a\} \rightarrow NOT b$.

```
if (in_current_state (f(?0:type1)) and
        in_current_state (g(?1:type2)) and
        in_goal_state (a(?0:type1)) and
        in_goal_state (d(?1:type2))) then
    op1
    op2
```

Table 6: The template DISTILL would create to represent the plan shown in Figure 13.

Merging Templates The merging process is formalized in the procedure Add_To_Template in Table 5. The templates learned by the DISTILL algorithm are sequences of non-nested if statements. To merge a new template into its knowledge base, DISTILL searches through each of the if statements already in the template to find one whose body (the solution plan for that problem) matches that of the new problem. We consider two plans to match if:

- one is a sub-plan of the other, or
- they overlap: the steps that end one begin the other.

If such a match is found, the two if statements are combined. If no match is found, the new if statement is simply added to the end of the template.

We will now describe how to combine two if statement templates, if_1= **if** x **then** abc and if_2= **if** y **then** b, when the body of if_2 is a sub-plan of that of if_1. This process is illustrated in Figure 14. [8]. For any set

[8] Combining two if statements with overlapping bodies is similar. It is illustrated in Figure 15

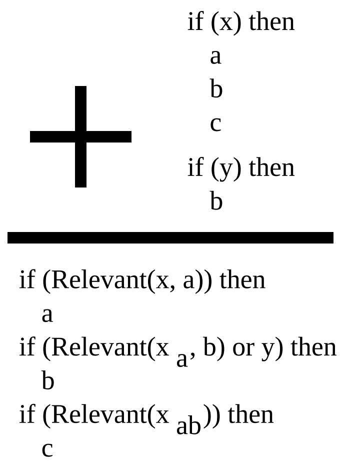

Figure 14: Combining two if statements when the body of one is a sub-plan of the body of the other.

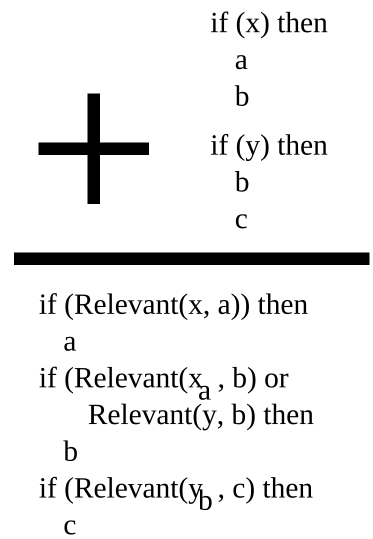

Figure 15: Combining two if statements when their bodies are overlapping.

of conditions C and any step s applicable in the situation C, we define C_s to be the set of conditions that hold after step s is executed in the situation C. We also define a new function, $Relevant(C, s)$, which, for any set of conditions C and any plan step s, returns the conditions in C that are relevant to the step s.

As shown in Figure 14, merging if_1 and if_2 will result in three new if statements. We will label them if_3, if_4, and if_5. The body of if_3 is set to a and its conditions are $Relevant(x, a)$. The body of if_4 is b and its conditions are $Relevant(x_a, b)$ **or** $Relevant(y, b)$. [9] Finally, the body of if_5 is c and its conditions are $Relevant(x_a b, c)$. Whichever of if_1 or if_2 is already a member of the template is removed and replaced by the three new if statements.

Illustrative Results Table 7 shows a template learned by the DISTILL algorithm that solves all problems in a blocks-world domain with two blocks. There are 555 such problems [10], but the template needs to store only two plan steps, and DISTILL is able to learn it from only 6 example plans.

Table 8 shows a template learned by the DISTILL algorithm to solve all gripper-domain problems with one ball, two rooms, and one robot with one gripper arm. Although there are 1932 such problems, [11] the DISTILL algorithm is able to learn the template from only five example plans. It successfully generalizes situations within individual plans for use in other plans. Also note that only five plan steps (the length of the longest plan) are stored in the template.

Our results show that templates achieve a significant reduction in space usage compared to case-based or analogical plan libraries. In addition, templates are also able to situationally generalize known problems to solve problems that have not been seen, but are composed of previously-seen situations.

3.2.3 Planning with Templates

Our algorithm for generating plans from templates is shown in Table 9. As previously mentioned, while executing the template, we must keep track of a current state and of the current solution plan. The current state is initialized to the initial state, and the solution plan is initialized to the empty plan. Executing the template consists of applying each of the statements to the current state. Each statement in the template is

[9] Note that, though $Relevant(x, a) \subseteq x$, $Relevant(y, b) = y$.

[10] Though the initial state must be fully-specified in a problem, the goal state need only be partially specified. There are only three possible fully specified states in the blocksworld domain with two blocks, but there are 185 valid partially specified states.

[11] As previously mentioned, each problem consists of one fully-specified initial state (in this case, there are 6 valid fully-specified initial states), and one partially-specified goal state (in this case, there are 322).

```
if (in_current_state (clear(?1:block)) and
        in_current_state (on(?1:block ?2:block)) and
        (in_goal_state (on(?2:block ?1:block)) or
            in_goal_state (on-table(?1:block)) or
            in_goal_state (clear(?2:block)) or
            in_goal_state (¬on(?1:block ?2:block)) or
            in_goal_state (¬clear(?1:block)) or
            in_goal_state (¬on-table(?2:block))
        )) then
    move-from-block-to-table(?1 ?2)
if (in_current_state (clear(?1:block)) and
        in_current_state (clear(?2:block)) and
        in_current_state (on-table(?2:block)) and
        (in_goal_state (on(?2:block ?1:block)) or
            in_goal_state (¬clear(?1:block)) or
            in_goal_state (¬on-table(?2:block))
        )) then
    move-from-table-to-block(?2 ?1)
```

Table 7: A template learned by the DISTILL algorithm that solves all two-block blocks-world problems.

either an plan step, an if statement, or a while loop. If the current statement is a plan step, make sure it is applicable, then append it to the solution plan and apply it to the current state. If the current statement is an if statement, check to see whether it applies to the current state. If it does, apply each of the statements in its body; if not, go on to the next statement. If the current statement is a while loop, check to see whether it applies to the current state. If it does, apply each of the statements in its body until the conditions of the loop no longer apply. Then go on to the next statement.

Sometimes there may be many ways to apply an if statement or a while loop to the current state. For example, if we have a statement like, "**if** (**in_current_state** (not-eaten(?a:apple))) **then** eat(?a)", and there are several uneaten apples in the current state, it is unclear which apple should be eaten. However, one of our primary assumptions is that all objects that match the conditions may be treated the same, so, in this case, it doesn't matter which apple is eaten.

Detecting and Handling Failures There are three ways a template may fail to generate the correct solution plan. It may have run through the whole template and found no solution steps at all, though the initial state is not the same as the goal state. Or, it may have found some plan steps to execute, but, by the end of the template, did not reach the goal state. Finally, it may have found some plan steps to execute, but found that they were not applicable to the current state. A failure is detected when we attempt to execute steps that are not applicable in the current state or when the template finishes executing and its final state does not match the goal state. The way we currently handle failures is by handing the problem off to a generative planner, and then to add that new solution to the template.

```
if (in_current_state (at(?3:ball ?2:room)) and
        in_current_state (at-robby(?1:room)) and
        (in_goal_state (at(?3:ball ?1:room)) or
            in_goal_state (¬at(?3:ball ?2:room)) or
            in_goal_state (holding(?3:ball)) or
            in_goal_state (¬free-arm)
        )) then
    Move(?1 ?2)
if (in_current_state (at(?3:ball ?2:room)) and
        in_current_state (at-robby(?2:room)) and
        (in_goal_state (at(?3:ball ?1:room)) or
            in_goal_state (¬at(?3:ball ?2:room)) or
            in_goal_state (holding(?3:ball)) or
            in_goal_state (¬free-arm)
        )) then
    Pick(?3 ?2)
if (in_current_state (holding(?3:ball)) and
        in_current_state (at-robby(?2:room)) and
        (in_goal_state (at(?3:ball ?1:room)) or
            (in_goal_state (¬at(?3:ball ?2:room)) and
                (in_goal_state (¬holding(?3:ball)) or
                    in_goal_state (free-arm)))
        )) then
    Move(?2 ?1)
if (in_current_state (holding(?3:ball)) and
        in_current_state (at-robby(?1:room)) and
        (in_goal_state (at(?3:ball ?1:room)) or
            in_goal_state (¬holding(?3:ball)) or
            in_goal_state (free-arm) )) then
    Drop(?3 ?1)
if (in_current_state (at-robby(?1:room)) and
        (in_goal_state (at-robby(?2:room)) or
            in_goal_state (¬at-robby(?1:room)))
        ) then
    Move(?1 ?2)
```

Table 8: A template learned by the DISTILL algorithm that solves all gripper-domain problems involving one ball two rooms, and one robot with one gripper.

Input: Template T, initial state $\mathcal{I}$, current state $\mathcal{C}$
(initialized to $\mathcal{I}$), and goal state $\mathcal{G}$.
Output: Plan P that solves the given problem.

```
procedure Apply_Template(T, I, C, G):
    P ← ∅
    for each statement S_n in T do
        P ← P + Apply_Statement(S_n, I, C, G)
    if G is satisfied by C then
        return P
    else
        FAIL

procedure Apply_Statement(S, I, C, G):
    P ← ∅
    if S is an if statement then
        if Applies_Now(S, C, G) then
            for each statement S_i in the body of S do
                P ← P + Apply_Statement(S_i, C, G)
    if S is a while statement then
        while Applies_Now(S, I, C, G) do
            for each statement S_i in the body of S do
                P ← P + Apply_Statement(S_i, C, G)
    if S is a plan step then
        if not Applicable(S, C) then
            FAIL
        C ← Apply_Step(S, C)
        P ← S
    return P
```

Table 9: Template-based plan generation.

4 Proposed Work

I propose to study the problem of dynamically learning a compact algorithmic model of agent behavior from observed action sequences. In general, there is no compact model, since an agent could act randomly. Even worse, it is impossible to learn a compact algorithmic model of the behavior of an agent acting optimally to solve NP-hard problems, since there is no such model. However, realistically, agents do not act randomly; they act to achieve their goals. They also do not act optimally when faced with NP-hard problems because of the prohibitive cost of generating optimal action sequences. For these reasons, I believe that it is possible to learn a compact algorithmic model of agent behavior in many realistic situations.

In particular, I propose to:

- Continue to develop algorithms to extract programs from observed executions;
- Investigate how to modify or repair the programs online in the presence of new information;
- Show that a program can be learned from example traces of its execution.

4.1 Main Focus

The main focus of my work will be on extracting rich domain-specific planning programs from example plans and on using and maintaining the programs as the planner gains more experience.

4.1.1 Acquiring Programs from Examples

They key contribution of this thesis will be an algorithm for extracting domain-specific planning programs from example plans. I have already made some progress towards this goal; as described in Section 3.2, I have demonstrated how to extract non-looping programs from example plans and to combine those programs into compressed representations of the learner's experience [62].

Three main steps in the learning algorithm remain to be completed. The first is extracting simple loops (loops without sub-loops or conditionals) from example plans. I have drafted an algorithm, shown in Table 10, to identify both parallel and serial simple loops in example plans. The second step is extracting complex loops (loops with sub-loops and conditionals) from example plans. The third step is merging templates with simple and complex loops.

4.1.2 Use and Repair of Acquired Programs

A necessary component of the use of acquired programs will be repairing the programs as new problems reveal flaws in the current program and extending them as new examples reveal more information about problem solving in the domain.

Since the domain-specific planning programs are acquired from a set of observed examples, their completeness cannot be guaranteed. And since the learning algorithm must infer the existence of loops in the observed plans, it may falsely identify some loops, fail to identify others, and incorrectly determine the invariants and stopping conditions of others. These errors in the program will be identified when new problems cannot be solved or are solved incorrectly by the program.

I plan to deal with incompleteness by querying the observed agent or an external generative planner for a solution to the new problem and merging that solution into the program as another example plan. It is not clear how to handle incorrectness in the program. I will investigate several possibilities. One is to remove incorrect loops. Another is to add new conditions to the loops to better characterize their looping and stopping conditions. It may also be possible to address the problem simply by merging a solution to the problem causing the incorrectness into the template.

As for incorporating new examples into the program, I envision the learning algorithm as one that can be executed either on- or offline. When the domain-specific planner is first constructed, it makes more sense to allow the learning algorithm to access many observed plans simultaneously, allowing it to merge the plans into a program in the best way. However, I expect that the planning system will also encounter new examples in the course of its planning experience. In this case, I expect we will find it to be better in terms of time

```
procedure Identify_Loops(Minimal annotated consistent partial order plan 𝒫):
    change ← true
    while (change) do
        change ← false
        ∀ fan outs
            ∀ fans with same sequences:
                identify varying parameter(s),
                introduce new variable for them
                if fans then have same init conds & results then
                    newloop ← empty while loop
                    newloop.conditions ← fans.init conds & results
                    newloop.body ← fan.sequence
                    replace fans with newloop
                    reconnect condition and result arcs
                    change ← true
        ∀ sequences
            use string-matching algs to find repeated seqs
            ∀ repeated sequences
                if last repetition has different outcome then
                    identify varying parameter(s) (if any)
                    introduce new variable for them
                    newloop ← empty while loop
                    newloop.conditions ← not last outcome and any common conditions
                    newloop.body ← sequence
                    replace sequence with newloop
                    reconnect condition and result arcs
                    change ← true
                else if sequences operate over all elements of a particular type then
                    identify which elements as varying parameter
                    introduce new variable for them
                    newloop ← empty while loop
                    newloop.conditions ← not all elements are taken care of and any common conditions
                    newloop.body ← sequence
                    replace sequence with newloop
                    reconnect condition and result arcs
                    change ← true
```

Table 10: Identifying loops in an observed plan

and space utilization to allow the system to merge the new plans into its current program online rather than constructing a new planner from scratch after including the new plans in a large database.

4.1.3 Example-Bounded Soundness, Completeness, and Optimality

Much planning research seeks to provide guarantees about the behavior of planning algorithms: that they are *sound* (any solution plan they find is guaranteed to be executable and to achieve the goals), *complete* (if there is a solution plan, they are guaranteed to find it), and *optimal* (the solution plan they find is guaranteed to be as good as the best possible solution, according to some quality criterion). Since templates are extracted from an arbitrary set of example plans, they cannot a priori be guaranteed to be sound, complete, or optimal. However, we can still make some guarantees.

To address soundness, I will supplement the automatically generated templates with a plan verifier that will determine whether the solution plan created by the template is executable and whether it satisfies the goals. Thus, all solution plans returned by the template planning system will be sound: they will be executable and will achieve the goals.

As for completeness, an acquired template will be guaranteed to be able to solve the problems from which it was acquired. However, it cannot currently be guaranteed to solve any other problems. I hope to be able to extend the template planning and learning algorithm to guarantee that it will be able to find all solutions composed of non-interacting subplans of observed example plans.

Finally, we can make no claims about optimality. Even if the examples used to generate the template were optimal, the template may not be, since a new problem may be solved suboptimally with the same technique that solved a previous problem optimally. However, the generated template will reflect the choices and preferences demonstrated in the example plans from which it was acquired. Therefore, the solutions it will generate will be similar in type to those from which it was learned.

4.2 Other Possible Directions

I am very interested in many other research directions. Following are descriptions of some of the problems I hope to have a chance to address in my thesis.

4.2.1 Agent Modelling

I would like to study the applicability of the algorithms I develop for learning programs from example executions to agent modelling problems. I believe the algorithms will be especially useful for user modelling problems that involve not only solving a problem, but generating a solution that reflects the observed user preferences, like travel planning or generating driving directions. Where needed, I will introduce new domains to describe these problems.

4.2.2 Using Partial Solutions

Even when an acquired program fails to solve a given problem, its execution may result in a partial solution. Although such a partial solution would not solve the complete problem, it might solve part of the problem, leaving a smaller and more manageable problem for the teacher or generative planner. It is also possible that the new problem would be solvable if a loop were revealed in the existing program. It may be possible to identify this situation and to reveal the location and conditions of the loop based on the structure of the program and of the new problem. This would allow the system to solve the new problem and extend the learned program without falling back on the teacher or generative planner. I am interested in investigating in which situations these partial solutions can be used to reduce or eliminate the system's reliance on external sources to find the complete solution.

There are also cases in which the acquired program fails to provide even a partial solution but may contain information useful for solving the given problem. For example, the program may be able to solve a similar problem, but for objects of different types. Or, the steps that would solve the problem have conditions not fulfilled by the current problem. The system could check whether these solutions are executable in the new situation. If they are, the program could be generalized to the new situation, and the problem could be solved without relying on external sources.

4.2.3 Active Learning

If the system were provided a generative planner or teacher, it could query for solutions to unsolved problems instead of waiting for them to be provided. I am interested in how the system could identify gaps in its knowledge, how it could determine which queries would be most useful, and how it could test whether its current program is complete (whether it is able to solve all problems in the domain).

4.2.4 "Programmable" Domains

I would like to investigate existing domains to determine which are *programmable*, or in which there exist compact algorithms that can solve all problems in the domain. I am interested in what separates programmable from non-programmable domains. I may also present new domains to more clearly illustrate the features of a programmable domain.

4.3 Evaluation

The evaluation of my work will be based on evaluating the three main features of the system. The first is the efficacy and robustness of the program learning algorithm. The primary test of the learning algorithm will be to determine whether it can reconstruct hand-written programs when presented with random traces from them. I will also investigate how many examples are needed to reconstruct the program and whether more are necessary with online versus offline learning. I will test the robustness of the algorithm by investigating whether programs are learnable from example traces not taken from a pre-written program, but provided by humans or ordinary planners. These examples will not exhibit the regularity of those taken from a pre-written program and it will thus be more challenging to extract a program from them.

I will also compare planning with learned programs to planning with other forms of learned information and to planning from scratch. I will compare the speed of problem solving using the different methods, the maximum problem sizes each of the methods can handle, and the efficiency of the solutions generated by each of the different methods.

Finally, I will evaluate the ability of the learned program to predict the behavior or human or computer agents. Accurately predicting agent behavior requires a much more accurate model of behavior than learning to solve problems based on observed behavior. It is unclear whether a system built to extract information about problem solving will be accurate enough to be useful for agent modelling.

References

[1] Charles W. Anderson. Strategy learning with multilayer connectionist representations. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 103–114, Irvine, California, 1987. Morgan Kaufmann.

[2] Corin R. Anderson, David E. Smith, and Daniel S. Weld. Conditional effects in graphplan. In Reid Simmons, Manuela Veloso, and Steven Smith, editors, *Proceedings of the fourth international conference on Artificial Intelligence Planning Systems (AIPS-98)*, pages 44–53, Pittsburgh, PA, June 1998. AAAI Press.

[3] Christer Bäckström. Finding least constrained plans and optimal parallel executions is harder than we thought. In Christer Bäckström and Erik Sandewall, editors, *Current Trends in AI Planning: Second European Workshop on Planning (EWSP-93)*, Frontiers in AI and Applications, pages 46–59, Vadstena, Sweden, Dec 1993. IOS Press.

[4] M. Bauer. Programming by examples. *Artificial Intelligence*, 12:1–21, 1979.

[5] Ralph Bergmann. Knowledge acquisition by generating skeletal plans from real world cases. In F. Schmalhofer, G. Strube, and T. Wetter, editors, *Contemporary Knowledge Engineering and Cognition*, pages 125–133, 1992.

[6] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[7] Jaime G. Carbonell and Yolanda Gil. Learning by experimentation: The operator refinement method. In R. S. Michalski and Y. Kodratoff, editors, *Machine Learning: An Artificial Intelligence Approach, Volume III*, pages 191–213. Morgan Kaufmann, Palo Alto, CA, 1990.

[8] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 875–881. AAAI Press, 1998.

[9] A. Cimatti, M. Roveri, and P. Traverso. Strong planning in non-deterministic domains via model checking. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning System (AIPS'98)*, pages 36–43. AAAI Press, 1998.

[10] Kurt Driessens. Relational reinforcement learning. In Michael Luck, Vladimír Marík, Olga Stepánková, and Robert Trappl, editors, *Multi-Agent Systems and Applications, 9th ECCAI Advanced Course ACAI 2001 and Agent Link's 3rd European Agent Systems Summer School (EASSS 2001)*, volume 2086 of *Lecture Notes in Computer Science*, pages 271–280, Prague, Czech Republic, July 2001. Springer.

[11] Sašo Džeroski, Luc De Raedt, and Hendrik Blockeel. Relational reinforcement learning. In *Proceedings of the International Workshop on Inductive Logic Programming*, pages 11–22, 1998.

[12] Oren Etzioni. Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–302, August 1993.

[13] Richard Fikes and Nils J. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[14] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[15] Peter E. Friedland and Yumi Iwasaki. The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, 1(2):161–208, 1985.

[16] Kristian J. Hammond. Chef: A model of case-based planning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 261–271. American Association for Artificial Intelligence, 1996.

[17] Judith Rich Harris. Where is the child's environment? a group socialization theory of development. *Psychological Review*, 102(3):458–489, July 1995.

[18] R. M. Jensen and M. M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research*, 13:189–226, 2000.

[19] Subbarao Kambhampati. *Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach.* PhD thesis, University of Maryland, College Park, MD, October 1989.

[20] Subbarao Kambhampati and James A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2-3):193–258, June 1992.

[21] Subbarao Kambhampati and Smadar Kedar. A unified framework for explanation-based generalization of partially ordered and partially instantiated plans. *Artificial Intelligence*, 67(1):29–70, 1994.

[22] Suresh Katukam and Subbarao Kambhampati. Learning explanation-based search control rules for partial order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-94)*, volume 1, pages 582–587, 1994.

[23] Henry A. Kautz and James F. Allen. Generalized plan recognition. In *Proceedings of the fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 32–37, Philadelphia, PA, August 1986. AAAI press, Menlo Park, CA.

[24] Roni Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1-2):125–148, 1999.

[25] Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In Thomas Dietterich and William Swartout, editors, *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, Menlo Park, California, 1990. AAAI Press.

[26] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[27] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–78, April 1985.

[28] Tessa Lau. *Programming by Demonstration: a Machine Learning Approach.* PhD thesis, University of Washington, Seattle, 2001.

[29] David B. Leake, editor. *Case-Based Reasoning: experiences, lessons, and future directions.* AAAI Press/The MIT Press, May 1996.

[30] D. Long. The AIPS-98 planning competition. *AI Magazine*, 21(2):13–34, 2000.

[31] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engeineering*, 18(8):674–704, August 1992.

[32] Steven Minton. Selectively generalizing plans for problem-solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 596–599, Los Angeles, CA, 1985. Morgan Kaufmann.

[33] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* Kluwer Academic Publishers, Boston, MA, 1988.

[34] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, March 1988.

[35] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[36] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8:295–318, 1991.

[37] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[38] Edwin Pednault. Formulating multiagent, dynamic-world problems in the classical planning framework. In Michael Georgeoff and Amy Lansky, editors, *Reasoning about actions and plans: Proceedings of the 1986 workshop*, pages 47–82, Los Altos, California, 1986. Morgan Kaufmann.

[39] J. Scott Penberthy and Daniel Weld. UCPOP: A sound, complete, partial-order planner for adl. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *proceedings of the third international conference on knowledge representation and reasoning (KR-92)*, pages 103–114, Cambridge, MA, October 1992. Morgan Kaufmann.

[40] Larry D. Pyeatt and Adele E. Howe. Decision tree function approximation in reinforcement learning. Technical Report CS-98-112, Colorado State University, Fort Collins, Colorado, 1998.

[41] Pierre Regnier and Bernard Fade. Complete determination of parallel actions and temporal optimization in linear plans of action. In Joachim Hertzberg, editor, *European Workshop on Planning*, volume 522 of *Lecture Notes in Artificial Intelligence*, pages 100–111. Springer-Verlag, Sankt Augustin, Germany, March 1991.

[42] Charles Rich and Richard C. Waters. Approaches to automatic programming. Technical Report 92-04, Mitsubishi Electric Research Laboratories Cambridge Research Center, Cambridge, Massachusetts, July 1992.

[43] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[44] Ute Schmid. *Inductive Synthesis of Functional Programs*. PhD thesis, Technische Universität Berlin, Berlin, Germany, May 2001.

[45] Ute Schmid and Fritz Wysotzki. Applying inductive program synthesis to macro learning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, pages 371–378, Breckenridge, Colorado, April 2000.

[46] Marcel J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-1987)*, pages 1039–1046, Milan, Italy, 1987.

[47] Jude W. Shavlik. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5:39–50, 1990.

[48] P. Shell and Jaime Carbonell. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, 1989.

[49] D. R. Smith. KIDS: A knowledge-based software development system. In M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*. AAAI press, 1991.

[50] David E. Smith and Mark A. Peot. Postponing threats in partial-order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 500–507, Washington, D.C., 1993. AAAI Press/MIT Press.

[51] David E. Smith and Mark A. Peot. Suspending recursion in causal-link planning. In B. Drabble, editor, *Proceedings of the third international conference on Artificial Intelligence Planning Systems (AIPS-96*, pages 182–190, Edinburgh, Scotland, May 1996.

[52] Mark Stefik. Planning and metaplanning. In Nils J. Nilsson and Bonnie Lynn Webber, editors, *Readings in Artificial Intelligence*, pages 272–286. Tioga Publishing, Palo Alto, CA, 1981.

[53] William T. B. Uther and Manuela Veloso. The lumberjack algorithm for learning linked decision forests. In *Symposium on Abstraction, Reformulation and Approximation (SARA-2000)*, Lecture Notes on Artificial Intelligence. Springer Verlag, 2000.

[54] Manuela Veloso, Alicia Pérez, and Jaime Carbonell. Nonlinear planning with parallel resource allocation. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, pages 207–212, San Diego, CA, November 1990. Morgan Kaufmann.

[55] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, December 1994.

[56] Manuela M. Veloso. Prodigy/analogy: Analogical reasoning in general problem solving. In S. Wess, K.-D. Althoff, and M. Richter, editors, *Topics on Case-Based Reasoning*, pages 33–50. Springer Verlag, 1994.

[57] Xuemei Wang. Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, pages 335–340, Chicago, IL, June 1994.

[58] Daniel Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, Winter 1994.

[59] Robert Wilensky. A model for planning in complex situations. *Cognition and Brain Theory*, IV(4), Fall 1981.

[60] Robert S. Williams. Learning to program by examining and modifying cases. In John Laird, editor, *Proceedings of the fifth international conference on machine learning (ICML-88)*. Morgan Kaufmann, 1988.

[61] Elly Winner and Manuela Veloso. Analyzing plans with conditional effects. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*, Toulouse, France, April 2002.

[62] Elly Winner and Manuela Veloso. Automatically acquiring planning templates from example plans. In *Proceedings of the AIPS-2002 Workshop on Exploring Real-World Plans*, Toulouse, France, April 2002.