Learning Planning Operators by Observation and Practice

Xuemei Wang

June 1996
CMU-CS-96-154

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Jaime G. Carbonell, Chair
Herbert A. Simon
Manuela M. Veloso
Jill Fain Lehman
Douglas H. Fisher, Vanderbilt University

Copyright © 1996 Xuemei Wang

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government.

Keywords: Artificial intelligence, planning, problem solving, PRODIGY, knowledge acquisition, machine learning, observation, practice.

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

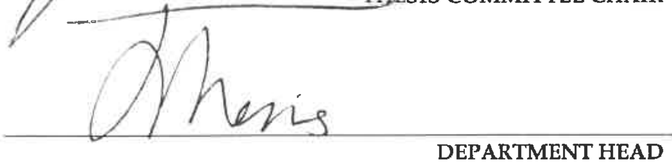
Learning Planning Operation by Observation and Practice

XUEMEI WANG

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

THESIS COMMITTEE CHAIR — 6/24/96 DATE



DEPARTMENT HEAD — 8/21/96 DATE

APPROVED:

DEAN — 8-26-96 DATE

Abstract

Acquiring and maintaining domain knowledge is a key bottleneck in applications of planning systems. This thesis describes a machine learning approach to automatic acquisition of planning operators. Our approach is to learn planning operators by observing expert solution traces and to refine operators through practice in a learning-by-doing paradigm. During observation, our system uses the knowledge that is observable when experts solve problems, without the need of explicit instruction or interrogation. During practice, our system generates its own learning opportunities by solving practice problems.

The inputs to our learning system are: the description language for the domain, experts' problem solving traces, and practice problems to allow learning-by-doing operator refinement. The output is a set of operators, each described by a list of variables, preconditions, and effects. The operators are learned incrementally using an inductive algorithm. During practice, our system effectively generates plans using incomplete and incorrect operators, repairs plans upon execution failures, and integrates planning, learning, and execution.

Our approach has been fully implemented and tested in a system called `OBSERVER`, which is built in the context of the PRODIGY4.0 nonlinear planner. Empirical results in a process planning domain and a version of the 34 meter Deep Space Network antenna operation domain demonstrate the validity of our approach. These results show that our system learns operators in these domains well enough to solve problems as effectively as expert human-coded operators, and that learning by observation and learning by practice both contribute significantly to the learning process.

Acknowledgements

It gives me pleasure to thank those who helped me through graduate school ....

This thesis would have not been possible without my advisor, Jaime Carbonell. Seven years ago, when I came to CMU with a degree in Applied Mathematics from Beijing, I knew little about Artificial Intelligence, or Computer Science. The transition was difficult, but Jaime made it easy for me. He patiently initiated me into Artificial Intelligence; he wisely guided me through the maze of graduate studies and research; he ceaselessly gave me confidence when I doubted myself. How many times have I gone into his office frustrated and discouraged about my research and come out smiling!

I thank my other committee members, Herb Simon, Manuela Veloso, Jill Fain, and Doug Fisher, all of whom gave me much help in my research and in writing my thesis. Herb, whose wisdom spans Artificial Intelligence, Economics, Psychology and many other areas, helped me to look at my work from a global perspective. As a line from a Chinese poem says, "To encompass the vista of thousand miles, climb up another flight of stairs." Manuela, with keen insights in the field of planning and machine learning, has never let the smallest detail of my work pass her scrutiny. Jill, whose enthusiam never ceases to amaze me, has push me much further in my research. Doug has always given me prompt feedback on research ideas and papers, and has taught me that there is a world outside CMU interested in my research.

I am also grateful to all the present and past members of the PRODIGY project. The stimulating research environment of the PRODIGY group was one of the main reasons that drew me to the project, and has never disappointed me! Many people in the PRODIGY group provided invaluable technical support and substantial feedback on my many presentations and papers: Manuelo Veloso, Alicia Pérez, Eugene Fink, Jim Blythe, Scott Reilly, Rujith de Silva, Karen Haigh, and Peter Stone. And thanks to the past members who have influenced this work: Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn, Craig Knoblock, Dan Kuokka, and Steve Minton.

I thank many people, besides members of my thesis committee, who took the time to read the thesis and gave me helpful comments: Steve Chien, Eugene Fink, Jonathan Gratch,

Henry Rowley, and Brian Surkan. Special thanks go to Steve Chien who has helped me since the writing of my thesis proposal, the first document that led to this thesis.

I have enjoyed the School of Computer Science at Carnegie Mellon University since the first day I was here. My wonderful officemates, Alicia Perez, Puneet Kumar, and Henry Rowley made my office a pleasant place to be in and made my time at work more delightful. They have helped in almost every aspect of my daily needs in the department, from Unix commands to drawing Postscript pictures. I have also learned much from my former officemates Rich Caruana, Daniel Julin, Ken Lang, Roni Rosenfeld, and Peter Weyhrauch, and from my current officemates Po-Jen Yang and James Thomas. Other friends in the department also made my graduate studies pleasant: Vincent Cate, Richard Goodwin, Jaspal Subhlok, Sebastian Thrun, and many more. Special thanks go to Sharon Burks, who has always greeted me with a pleasant smile when I walked into her office. She has taken care of every administrative detail concerning me and much more.

Pittsburgh is the city where I have lived the longest except for my hometown Wuhan in China. Throughout the years, I have grown to love this city, to love the many people I have had the pleasure to share my time with. Thanks to Wei Zhang who offered me the warmth of her home in Pittsburgh and innumerable delicious Chinese meals during my visits. Thanks to my friends at the ballroom dance club and at Rosebud. Enrique Mu, Heather Fisher, John Cheng, Andrew Huang, Mike Kajane, Fei Wong, Chris McConnell, and Marlon Silva, all danced gracefully with me to inspiring music.

I want to thank my uncle, Monto Ho, who has become so very dear to me. He has been father, friend, and mentor since the very first day I arrived in Pittsburgh. His love and caring have carried me through many difficult times in this foreign land. He taught me how to appreciate western culture while keeping my Chinese identity. He helped me to grow confident of myself. His influence pervades all my days in Pittsburgh and beyond.

Finally, I want to thank my parents, who planted the idea of pursuing a Ph.D since my childhood, and who taught me the value of hard work by their own example. Thanks to Yanling for the simple fact of her being my sister. I have always felt their love and support despite the fact that they are thousands of miles away.

Contents

1 Introduction **1**

1.1 Machine learning and planning 2

1.2 Methodology 3

1.3 Contributions 5

1.4 Thesis Organization 6

2 Learning by Observation and Practice—an Overview **7**

2.1 Initial Domain Knowledge 7

2.2 Input and Output 8

2.2.1 Input 9

2.2.2 Output 9

2.3 Assumptions 12

2.4 Architecture 13

2.5 Learning Operators as Concept Learning 15

2.5.1 Necessary properties of the learning algorithm 15

2.5.2 Learning operator preconditions 15

2.5.3 Learning operators effects 19

2.5.4 Learning the variable bindings—connecting the preconditions and effects 20

2.5.5 Summary of the learning method 21

2.6 Planning while Learning Operators 21

2.6.1 Interleaving planning and execution 22

2.6.2 Planning and plan repair 22

2.7 Summary 23

3 Learning Operators by Observation **25**

3.1 Input and Output . . . 25

3.2 Initializing the Operators . . . 27

3.3 Learning Operators Incrementally . . . 27

3.3.1 Learning variable bindings of the operators . . . 27

3.3.2 Updating the **S-rep** of operator preconditions . . . 29

3.3.3 Updating the operator effects . . . 33

3.4 Complexity Analysis . . . 38

3.5 Discussion . . . 39

3.5.1 Implication of Occam's Razor . . . 39

3.5.2 Learning the **S-rep** . . . 41

3.6 Summary . . . 41

4 Planning while Learning Operators **43**

4.1 Domain Knowledge Imperfections . . . 43

4.2 Issues in Planning while Learning . . . 45

4.3 Planning with Incorrect Operators and Plan Repair . . . 45

4.3.1 The Plan Repair Algorithm . . . 46

4.3.2 Integrating planning, execution and plan repair . . . 48

4.4 Notes on Implementation . . . 50

4.4.1 Resource bounds . . . 50

4.4.2 Binding generation . . . 51

4.5 Discussion . . . 51

4.5.1 Applicability to other operator-based planners . . . 51

4.5.2 Strategies for interleaving planning and execution . . . 52

4.5.3 Other plan repair strategies . . . 53

4.6 Summary . . . 54

5 Refining Operators by Practice **57**
5.1 Input and Output 57
5.2 Refining Operators 60
5.2.1 Updating the **S-rep** of operator preconditions 60
5.2.2 Learning the **G-rep** of operator preconditions 62
5.2.3 Refining operator effects 65
5.3 Complexity Analysis 66
5.4 Convergence Proof 69
5.5 Discussions 70
5.5.1 Trade-off between learning efficiency and learning rate 70
5.5.2 Negated preconditions 71
5.6 Summary 71

6 Empirical Results **73**
6.1 Application Domains 74
6.1.1 A process planning domain 74
6.1.2 DSN antenna operation domain 74
6.2 Design of Experimentation 76
6.2.1 Phases for learning and testing 76
6.2.2 Randomly generated problems 76
6.2.3 Base-level planner 77
6.3 Overall Effectiveness of OBSERVER 78
6.3.1 Criteria for evaluation 78
6.3.2 Results 80
6.3.3 Summary of the effectiveness evaluation 88
6.4 Role of Learning by Observation 90
6.5 Role of Practice 90
6.5.1 Method for demonstrating the role of practice 92
6.5.2 Results 92
6.5.3 Summary of the role of practice 95
6.6 Summary 95

7 Related Work **97**

7.1 Rule Learning in Structural Domains 97

7.1.1 Algorithms for learning in structural domains 97

7.1.2 Systems for learning in structural domains 101

7.2 Related Work in Planning 107

7.2.1 Machine learning in planning 107

7.2.2 Planning with incomplete information 107

7.2.3 Plan repair 108

7.2.4 Interleaving planning and execution 109

7.3 Knowledge Acquisition 109

8 Conclusions **111**

8.1 Summary of the Thesis 111

8.2 Contributions 112

8.3 Future Work 113

8.3.1 Handling uncertainty 113

8.3.2 Learning with unobservable features 114

8.3.3 Control knowledge learning 114

A The PRODIGY Problem Solver **117**

B A Trace From Practice **121**

Chapter 1

Introduction

Consider the problem of learning to repair an aircraft engine, or to configure a local computer network, or to produce a process plan to manufacture a metal part in a machine shop. Human learning of such tasks combines book learning, observation/instruction and repeated practice. First, the learner must know the vocabulary of the task domain. For example, in the process planning domain, one must learn the names of machines and tools, the properties that describe the part being created (e.g., geometrical attributes such as shape, orientation, features, and surface properties such as polished, rough or painted), as well as the relationships between machines, tools, and parts (e.g., the part is being held by a holding device). Second, the learner must acquire knowledge of the permissible actions and their preconditions and consequences, such as drilling, polishing, turning, clamping, cutting, milling, and painting. For instance, the action model for drilling requires that the part be securely clamped and oriented with the face pointed upward and orthogonal to the perforation, and the drillbit of appropriate diameter affixed to the drill press. Finally, methods for assembling effective and efficient plans out of these building blocks need to be learned: plans that achieve their desired effect, that consume minimal resources, and that can be generated as quickly as possible on demand.

This dissertation focuses on the second step of the complex-skill learning process, namely *how to acquire action models useful for planning and execution, given the basic domain vocabulary.* The third step, composition of effective plans in an efficient manner, is done automatically by the PRODIGY planner [Carbonell *et al.*, 1992, Veloso *et al.*, 1995].

The problem of formulating the domain knowledge for planning (also known as "planning operators", "rules" or "action models"), which occurs primarily at the second step, is a key bottleneck in real applications of planning systems [Chien *et al.*, 1995]. This is because it is difficult to write complete, consistent, and accurate operators—much iterative refinement is required for human experts and, as we shall see, also for our

automated approach in the learning by practice phase. Traditional approaches to domain-knowledge acquisition usually require the AI expert to learn about the domain, or the domain expert to learn the syntax and semantics of AI programs. To facilitate knowledge acquisition in this fashion, considerable research has focused on *knowledge acquisition tools* for rule-based systems (see [Boose and Gaines, 1989] for a summary) and, to a lesser extent, on using such tools for specialized planning systems [Chien, 1996]. These systems all require considerable interaction with domain experts, and it usually takes months or years to build the domain knowledge for an application domain. In particular, significant effort has been devoted to writing and debugging planning operators for applications of planning systems [Hayes, 1990, Gil, 1991, Chien, 1994].

To address this knowledge-acquisition bottleneck, this thesis focuses on automated acquires planning operators using machine learning methods. Our approach is to learn planning operators by observing expert solution traces and to refine operators through practice in a learning-by-doing paradigm. During observation, our system uses the knowledge that is observable when experts solve problems, without the need of explicit instruction or interrogation. During practice, our system generates its own learning opportunities by solving practice problems. Our approach for automatic acquisition of planning operators can significantly facilitate and speed-up applications of planning systems.

This chapter is divided into four sections. In the first section, we situate our work in the context of machine learning and planning. The second section introduces the thesis by briefly describing the methodology for operator learning. The third section discusses contributions. Finally, we provide the reader's guide to this dissertation.

1.1 Machine learning and planning

There are two primary reasons that learning is essential to transform planning systems from research tools to useful applications in real world. First, the world we live in is constantly changing. It is impossible to hand-code all the knowledge that will ever be required by the system. To adapt to such changes, the application system must learn from its experience and from the environment. Second, acquiring even the original base-level knowledge for complex domains through knowledge engineering is a costly and laborious process. Any significant automation that produces knowledge comparable in quality to human-coded knowledge renders significant benefits. Accordingly, much previous work has concentrated on machine learning in planning systems. These approaches can be divided into three categories.

The first category, also known as "speed-up" learning, is on acquiring knowledge that improves the efficiency of planning. Several techniques have been used in this framework, including learning macro-operators [Fikes *et al.*, 1972, Korf, 1985, Cheng and Carbonell,

1986, Segre, 1988], learning control rules [Minton, 1988, Tadepalli, 1989, Etzioni, 1990, Katukam and Kambhampati, 1994, Borrajo and Veloso, 1994a, Estlin and Mooney, 1996], learning by analogy [Veloso, 1994], chunking [Laird *et al.*, 1986], and learning abstraction hierarchies [Knoblock, 1994, Christensen, 1990].

The second category is on acquiring heuristics to guide the planner to automatically produce plans of high quality. Relatively little research has addressed this issue [Ruby and Kibler, 1990, Iwamoto, 1994, Pérez, 1995].

The third category is on learning domain knowledge for planning. The goal of this type of learning is to improve initially incomplete or inaccurate descriptions of operators, or to learn operators from scratch. Several research efforts have focused on automatic acquisition and refinement of planning operators using inductive learning techniques [Porter and Kibler, 1986, Shen, 1994, Gil, 1992, Wang, 1995, Oates and Cohen, 1996]. Such is the learning problem this thesis addresses.

1.2 Methodology

We are interested in fully automated acquisition of planning operators. We have broken down the process of learning new operators into two phases: *observation* and *practice*. Learning by observation, whether by human or by machine, is advantageous because it is usually much easier for a domain expert to solve a problem than to explain the rules he/she uses. Expert problem-solving traces are observable when experts solve problems and, hence, are easily available to the learner.[1] Human learning usually involves practice as well, because the knowledge obtained solely by observation is frequently incomplete and inaccurate. Similarly, our learning system first acquires approximate operator definitions by *observation* and then refines these operators during *practice* by solving problems. In this practice phase, inconsistent and incomplete operators are diagnosed and repaired incrementally.

In the first phase of learning, the system observes the sequence of steps employed by domain experts in solving typical problems, noting all observable results of the actions. This observation sequence is then generalized and converted into an initial set of planning operators, using an incremental inductive machine learning algorithm. The learning algorithm identifies the common features of each action in the observations to form the operator descriptions. The advantage of learning by observation is that the learning system does not require explicit instruction from the domain expert. Therefore, it minimizes the burden on the expert.

[1] The use of expert solution traces has also been a central motivation for case-based reasoning [Shank, 1982].

Initially learned operators are usually incomplete and incorrect (e.g., overly-specific preconditions, overly-general preconditions, missing effects). These operators are refined in a learning-by-doing fashion, using the training examples automatically generated by the learning system. During practice, the learning system first tries to execute an action (operator) regardless of the state conditions. This may lead to an execution failure, since certain constraints that must be satisfied in order to execute the operator successfully may not hold. Upon an execution failure, the learning system finds the differences between its current state and the generalization of the states in which this action has been previously successfully executed. The system then tries to reduce some of these differences by applying other relevant actions, until the original operator can be executed successfully. Operators are refined using both the successful and unsuccessful executions.

For example, suppose the learning system executes an operator to spot-drill a part when the part is not held securely. This leads to an execution failure—the expected result that the part has a spot hole does not occur. The learning system notices that when the expert executed this operator, the part was always held securely. Therefore, it executes another action to hold the part securely, and re-executes the action to spot-drill the part. This time the part is successfully spot-drilled. If the only difference between the states in which the action was executed successfully and the state in which it fails is that the part is held securely in the former, then the operator is refined: the system learns that the part must be held securely in order to execute the operator spot-drill successfully. If there are more differences between the failure and successful states, additional plan repair and learning may be required.

This dissertation addresses the following issues for this type of learning by observation and practice:

- How to incrementally generalize operator preconditions and effects from a large amount of data efficiently
- How to solve practice problems effectively, given incomplete and incorrect initial operators, and to generate good training examples
- How to repair a failed plan, given that execution failures are inevitable when planning with incomplete or incorrect domain knowledge
- How to incrementally refine incomplete and incorrect operator description by practice

We have designed algorithms that automatically learn operators by observation and practice. Our algorithms have been fully implemented in a system called `OBSERVER`, which is

built on top of the PRODIGY4.0 nonlinear planner [Carbonell *et al.*, 1992]. The operators that are learned are exactly what are required by most operator-based planners such as STRIPS [Fikes and Nilsson, 1971], TWEAK [Chapman, 1987], PRODIGY [Carbonell *et al.*, 1992, Veloso *et al.*, 1995], SNLP [McAllester and Rosenblitt, 1991], and UCPOP [Penberthy and Weld, 1992]. The validity of our approach has been demonstrated in two application domains: a process plan domain [Gil, 1991, Gil and Pérez, 1994] and a DSN antenna operations domain [Hill *et al.*, 1995, Chien *et al.*, 1996b], which are described in further detail in Chapter 6. These two domains were chosen because they represent real world planning problems.

Our approach is different from other research on learning planning operators—it requires minimal domain knowledge, while some work requires initial approximate planning operators [Gil, 1992], or considerable background knowledge provided by the user to identify relevant properties of the world state that should be considered during learning [Porter and Kibler, 1986]. OBSERVER learns in complex *structural domains* [Haussler, 1989] while some other work applies to smaller domains with much smaller number of states [Shen, 1994], or assumes simpler *attribute-based* domains [Oates and Cohen, 1996],

1.3 Contributions

The contributions of this dissertation can be summarized as follows:

- A fully automated approach for learning operators from observable expert solutions traces, without requiring direct interaction with domain experts
- A novel method for refining incomplete and inaccurate operators using the system's own execution traces during practice, exploiting intentionally generated learning opportunities
- The design of domain-independent algorithms for planning with incomplete and incorrect operators, for plan repair upon execution failures, as well as for integrating planning, learning, and execution
- Full implementation of all the algorithms designed in a system called OBSERVER and their integration with a nonlinear planner (PRODIGY4.0)
- Empirical demonstration of the validity of the approach in two realistic domains, i.e., a process planning domain and an antenna operation domain for the Deep Space Network (DSN).

1.4 Thesis Organization

The rest of the dissertation is organized as follows.

In Chapter 2, we present the architecture of the overall learning system. We describe the initial domain knowledge given to OBSERVER, the input and output of OBSERVER, and the assumptions of the learning system. We discuss the necessary properties of the learning algorithms and provide a high-level description of the algorithms for operator learning, as well as for planning with incomplete and incorrect operators and plan repair.

Chapters 3, 4, and 5 give detailed description of the learning system. In particular, Chapter 3 addresses the process of learning operators by observation. We describe how OBSERVER generalizes the observations to learn the variables, preconditions, and effects of the operators.

Chapter 4 focuses on the process of practice. We discuss how initially learned operators may be incomplete and incorrect and how planning is thus affected. We describe OBSERVER's algorithm for planning with incomplete and incorrect operators, as well as for integrating planning, learning, and execution in a simulated environment. A trace obtained from OBSERVER's practice is shown in Appendix B.

Chapter 5 focuses on refining incomplete and inaccurate operators during practice. We show how OBSERVER uses the positive and negative examples generated during practice to update the preconditions and effects of the operators.

Chapter 6 introduces the two application domains, (i.e., the process planning and the DSN antenna operation), used in our experiments, and presents detailed empirical results demonstrating the effectiveness of the learning system in two domains. First, we demonstrate the effectiveness of the overall learning system. We show that the learned operators are effective as human expert-coded operators in solving test problems. Second, we demonstrate the role of observation. We show that the total number of solvable test problems increases with the number of observation problems. Finally, we demonstrate the role of practice. We show that the operators learned from initial observation and then refined through practice are significantly more effective in problem solving than the operators learned only by observation.

Chapter 7 discusses related work on rule learning in structural domains, on planning, and on knowledge acquisition.

Finally, Chapter 8 summarizes the results, highlights the contributions of the thesis, and discusses future research directions.

Chapter 2

Learning by Observation and Practice—an Overview

This thesis is about automatic learning of planning operators without any initial knowledge of the operators. Our approach is to learn operators inductively by observing expert solution traces and by practicing in a learning-by-doing fashion [Anzai and Simon, 1979], where each operator is described by a list of variables, preconditions, and effects.

This chapter gives an overview of this thesis. We first describe the initial domain knowledge given to OBSERVER, the input and output of OBSERVER, and the assumptions we make for learning. We then give an overview of OBSERVER's architecture, which has three main components: learning operators by observation, solving practice problems, and refining operators by practice. We give high-level descriptions of our algorithms for learning operators, and for planning with inaccurate operators. Details of these algorithms are further described in Chapters 3, 4, and 5. Throughout the rest of this dissertation, we will illustrate our algorithms with examples from the process planning domain.

2.1 Initial Domain Knowledge

Our learning system requires very little initial domain knowledge for learning—it does not have any knowledge about the preconditions or effects of the operators. The system is given at the outset only the domain vocabulary, which includes the following:

The domain object ontology. This includes the classes of objects that are used in the domain. For example, Figure 2.1 shows the partial type hierarchy in the process planning domain. In this domain, the types are fluid, tool, machine, holding-device,

etc. Each type may have subtypes. For example, there can be different types of holding-devices, such as toe-clamp, v-block, or vise. The type hierarchy guides OBSERVER in determining the types of the variables in the operators.

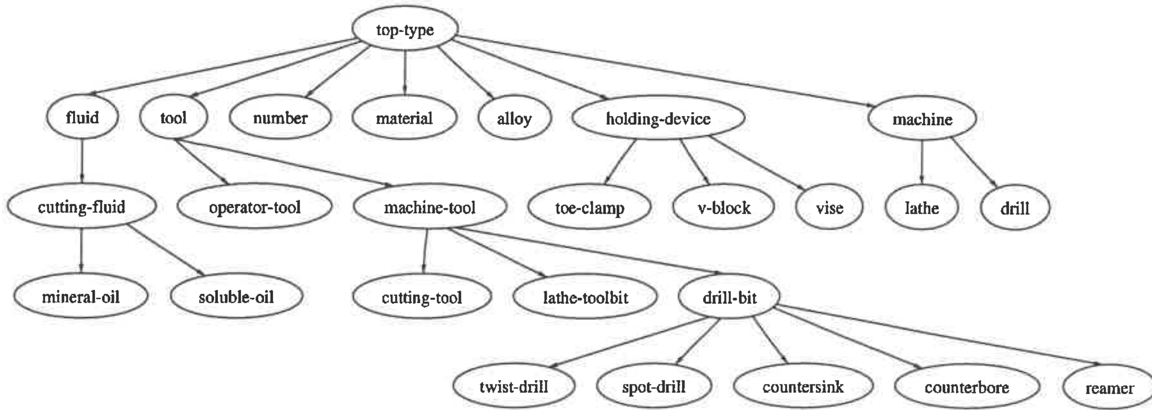


Figure 2.1: A partial type hierarchy of the process planning domain.

The description language. This includes the predicates that describe the states and the operators: `is-clean, shape-of, holding` are examples of predicates used in the process planning domain.

Static domain knowledge. This includes domain constants, i.e., the objects that are common in all the problems in a given domain. For example, the shape `rectangular` and the six sides (`side1, side2`, etc) of a rectangular block are domain constants in the process planning domain. Domain constants may occur in the preconditions or effects of operators.

In many domains, such as the process planning domain and the DSN antenna operation domain, the domain vocabulary has been long studied and agreed upon by domain experts, thus it is not difficult to acquire. In contrast, acquiring the preconditions and effects of the operators is usually a time- and labor-consuming task, if at all feasible. Thus, this thesis focuses on learning the preconditions and effects of the operators given the description language of the domain.

2.2 Input and Output

OBSERVER accepts as input the observations of expert solution traces and the practice problems to solve, and learns operator descriptions (i.e., variables, preconditions, and effects). The input and output are described here.

2.2.1 Input

The input to OBSERVER is the following:

1. Observations of an expert agent. An observation consists of:
   - The name of each operator in the expert's execution sequence
   - The collection of literals that describes the state in which each action is applied, called pre-state
   - The collection of literals that describes the state resulting from the execution of each action, called the post-state. The delta-state is the collection of literals that are added to or deleted from the pre-state as a result of the operator application. The delta-state can be determined from the difference between the post-state and pre-state

   Figure 2.2 gives an example of an observation of the operator HOLD-WITH-VISE. The pre-state and post-state are illustrated in Figure 2.3 and Figure 2.4, respectively. Note that many literals in the pre-state are irrelevant to the successful application of the operator.

2. Practice problems. The operators learned from observation are usually incomplete or incorrect, because observations alone do not provide negative examples and will not reveal the real relevant structure of the operator (more discussion on this can be found in Chapter 4). OBSERVER uses the initial operators to solve practice problems and further refines them based on its own execution traces.

2.2.2 Output

The output of learning is a set of planning operators. Each operator has a set of variables whose types are specified using the type hierarchy of the domain. An operator also has a precondition expression that must be satisfied in the state in order for the action to successfully execute in the state. Finally, An operator also has a set of regular effects (i.e., primary effects) that indicate unconditional changes, i.e., the formulas that are added to or deleted from the state when the operator is applied. An operator can also have conditional effects, which describe changes to the world also in terms of formulas to be added or deleted to/from the state, but as a function of a particular state configuration. A conditional effect is composed by a condition expression (i.e., conditional preconditions) and a regular effect. The effect should be added or deleted to the current state only if the conditional preconditions are true when the operator is applied.

op-name: `hold-with-vise`

Pre-state:

```
(has-device drill0 vise0)
(on-table drill0 part0)
(is-clean part0)
(is-empty-holding-device vise0 drill0)
(is-available-table drill0 vise0)
(holding-tool drill0 spot-drill0)
(is-available-part part0)
(hardness-of part0 hard)
(material-of part0 iron)
(size-of part0 width 2.75)
(size-of part0 height 4.25)
(size-of part0 length 5.5)
(shape-of part0 rectangular)
```

Delta-state:

adds:

```
(holding drill0 vise0 part0 side5)
```

dels:

```
(is-empty-holding-device vise0 drill0)
(on-table drill0 part0)
(is-available-part part0)
```

Figure 2.2: An observation of the state before and after the application of the operator `HOLD-WITH-VISE`. This figure illustrates the delta-state, which is the difference between the post-state and the pre-state of the observation. Note that many literals in the pre-states are irrelevant to the successful application of the operator `HOLD-WITH-VISE`, e.g., the sizes of the part and the material of the part.

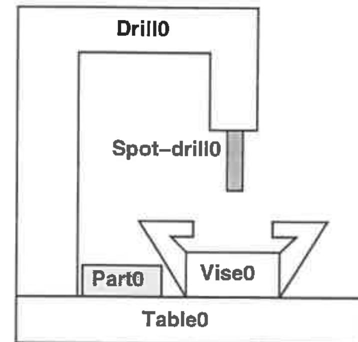


Figure 2.3: Pre-state of the observation in Figure 2.2.

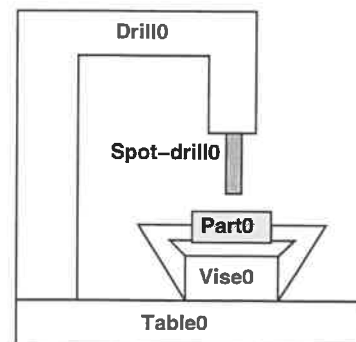


Figure 2.4: Post-state of observation in Figure 2.2.

operator	=	(**OPERATOR** *rule-name* (**preconds** *var-descriptor** *pdl-expression*) (**effects** *var-descriptor** *effects*))
var-descriptor	=	(**Or** *type-names*$^+$) | *type-name*
pdl-expression	=	(**AND** *pdl-expression*$^+$) | **not** *predicate* | *predicate*
effects	=	(*effect*$^+$)
effect	=	(**ADD** *predicate*) | (**DEL** *predicate*) | *conditional-effect*
conditional-effect	=	(**IF** *pdl-expression effects*)
rule-name	=	**id-name**
type-name	=	**id-name**
var-name	=	< **id-name** >

Table 2.1: The syntax for operators learned by OBSERVER.

Table 2.1 gives rigid definitions of the description language for operators that OBSERVER learns. This is a restricted form of the description language PRODIGY4.0 uses, but an extended version of the simple STRIPS operators—it has negated preconditions, and conditional effects with their corresponding conditional preconditions. Details of PRODIGY4.0's complete description language and search algorithm can be found in [Carbonell *et al.*, 1992] and in Appendix A.

As an example, Figure 2.5 shows the operator HOLD-WITH-VISE learned by OBSERVER by observation and practice. This operator represents the action of holding a vise. Note that in Figure 2.5, (not (has-burrs <part>)) is a negated precondition, meaning that HOLD-WITH-VISE can only be applied when (has-burrs <part>) is absent from the state. Also note that (add (holding-weakly <machine> <holding-device> <part> <side>)) and (add (holding <machine> <holding-device> <part> <side>)) are conditional effects, with their corresponding conditional preconditions being (shape-of <part> CYLINDRICAL) and (shape-of <part> RECTANGULAR). A conditional effect occurs *only* when its corresponding conditional preconditions are satisfied. To apply an operator successfully, the regular operator preconditions must be satisfied, whereas the conditional preconditions do not need to be satisfied.

```
(Operator HOLD-WITH-VISE
 (preconds ((<hd> Vise) (<side> Side) (<machine> Machine) (<part> Part))
  (and (has-device <machine> <hd>)
       (not (has-burrs <part>))
       (is-clean <part>)
       (on-table <machine> <part>)
       (is-empty-holding-device <hd> <machine>)
       (is-available-part <part>)))
  (effects
   (del (on-table <machine> <part>))
   (del (is-available-part <part>))
   (del (is-empty-holding-device <hd> <machine>))
   (if (shape-of <part> CYLINDRICAL)
       (add (holding-weakly <machine> <hd> <part> <side>))))
   (if (shape-of <part> RECTANGULAR)
       (add (holding <machine> <hd> <part> <side>))))
```

Figure 2.5: Operator HOLD-WITH-VISE from the process planning domain. This operator specifies the preconditions and effects of holding a part with a vise.

2.3 Assumptions

This thesis makes the following assumptions:

Deterministic operators. Since OBSERVER is operating within the framework of classical planners, it assumes that the operators and the states are deterministic, i.e., if the same operator with the same bindings are applied in the same states, the resulting states are the same.

Noise-free sensors. OBSERVER assumes noise-free sensors, i.e., there are no errors in the states. In other words, OBSERVER relies on the expert to give correct and complete descriptions pre-states and post-states. Although this may not always be easy, when experts miss part of the state descriptions, they tend to miss the parts that are irrelevant to the operator. This only enables OBSERVER to converge faster.

Conjunctive precondition expression. We notice that in most application domains, the majority of the operators have conjunctive preconditions expression only. For example, in more than 30 domains currently implemented in PRODIGY, more than 90% of the operators have only existential conjunctive preconditions. Furthermore, the domain expert can always split an operator with disjunctive preconditions into several operators with conjunctive preconditions. Therefore, OBSERVER assumes

that the operators have existential conjunctive preconditions. This assumption greatly reduces the search space for operator preconditions without sacrificing much of the generality of our learning approach.

Many issues complicate the task of learning operators by observation and practice, under the above assumptions:

- There are many literals in the pre-states that are irrelevant to the successful application of the operators. For example, in the process planning domain, there are on average about fifty literals in the pre-state or post-state of an observation, and about five to seven literals in the operator preconditions and three to five literals in the operators effects
- The operators may have conditional effects
- The operators may have negated preconditions, which significantly increases the space of possible precondition expression

2.4 Architecture

OBSERVER's overall learning architecture is shown in Figure 2.6. There are three components in the system: *(i)* learning operators by observation, *(ii)* solving practice problems, by planning and plan repair using learned operators and by executing plan steps in the environment, and *(iii)* refining operators during practice.

Learning operators by observation: Given very little initial knowledge, OBSERVER inductively learns an initial set of operators from the observations of expert solutions. Details of the learning algorithm are described in Chapter 3.

Planning, plan repair, and execution: The set of operators learned from observation can be incomplete and incorrect in many ways. They are further refined when OBSERVER uses them to solve practice problems. Given a practice problem, OBSERVER first generates an initial plan to solve the problem. The initial plan is executed in the environment, resulting in both successful and unsuccessful executions of operators. OBSERVER uses these executions as positive or negative training examples for further operator refinement. The planner also repairs the failed plans upon unsuccessful executions. The repaired plans are then executed and this process is repeated until the problem is solved, or until a resource bound is exceeded. Details of the integration of planning, execution, and learning are given in Chapter 4.

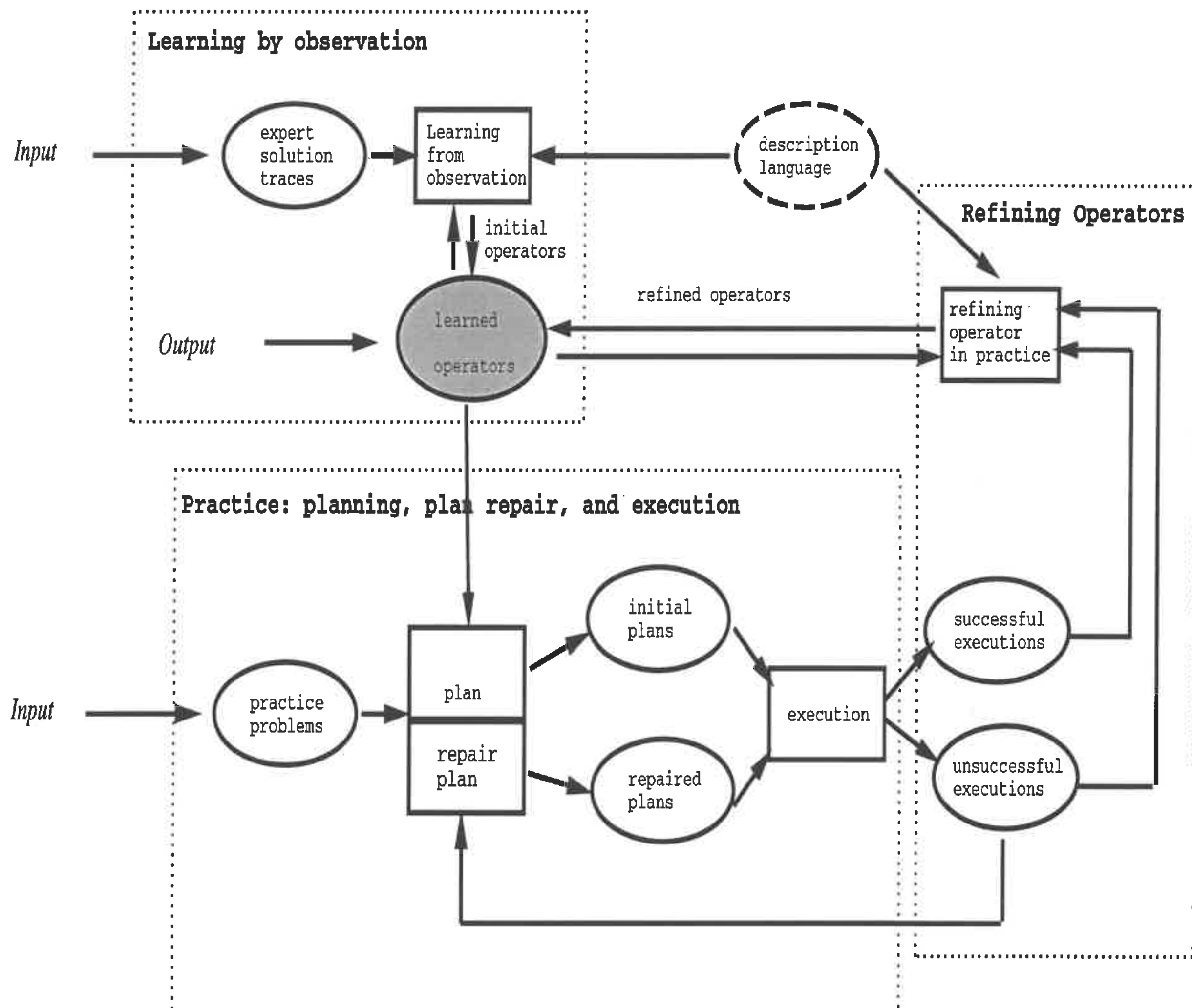


Figure 2.6: Overview of OBSERVER's learning architecture. There are three major components in this architecture: *(i)* learning operators by observation, *(ii)* solving practice problems, by planning and plan repair using learned operators and by executing plan steps in the environment, and *(iii)* refining operators during practice.

Refining operators during practice: The successful and unsuccessful executions generated during practice are effective training examples that OBSERVER uses to further refine the initial imperfect operators. The refinement method is described in Chapter 5.

2.5 Learning Operators as Concept Learning

Learning operators is a form of concept learning—given a set of positive and negative examples, the algorithm identifies a generalized description that is consistent with the positive examples but excludes the negative examples. The positive examples for learning are observed expert solution steps and OBSERVER's successful executions during practice. The negative examples are OBSERVER's unsuccessful executions during practice. This section motivates OBSERVER's learning approach and gives a high-level description of the learning algorithm.

2.5.1 Necessary properties of the learning algorithm

An algorithm for learning operators from observation and practice should have the following properties:

- The algorithm must handle new positive and negative examples efficiently. An incremental algorithm is more advantageous than a non-incremental algorithm, because it does not need to save all the previous examples and because it can be more efficient in processing a new example
- The algorithm should learn operators that facilitate planning for solving practice problems, and for plan repair upon possible execution failure. Refining operators from practice is a crucial component introduced in this thesis, thus generating effective training examples is important

OBSERVER's learning algorithm is designed according to the above criteria—it is incremental and learned operators can be used effectively for planning and plan repair. Figure 2.7 illustrates the incremental flavor of operator learning. Because of the incremental nature of the learning algorithm, it is necessary to match the preconditions and effects of the learned operators against the pre-states and the delta-states of new positive examples.

2.5.2 Learning operator preconditions

To learn operator preconditions, we need to identify the commonalities of the pre-states of positive examples that exclude negative examples. In general, there are two directions for learning—specific-to-general and general-to-specific.

Algorithms such as GOLEM [Muggleton and Feng, 1990] and the interference matching algorithm [Hayes-Roth and McDermott, 1978] use specific-to-general learning based on

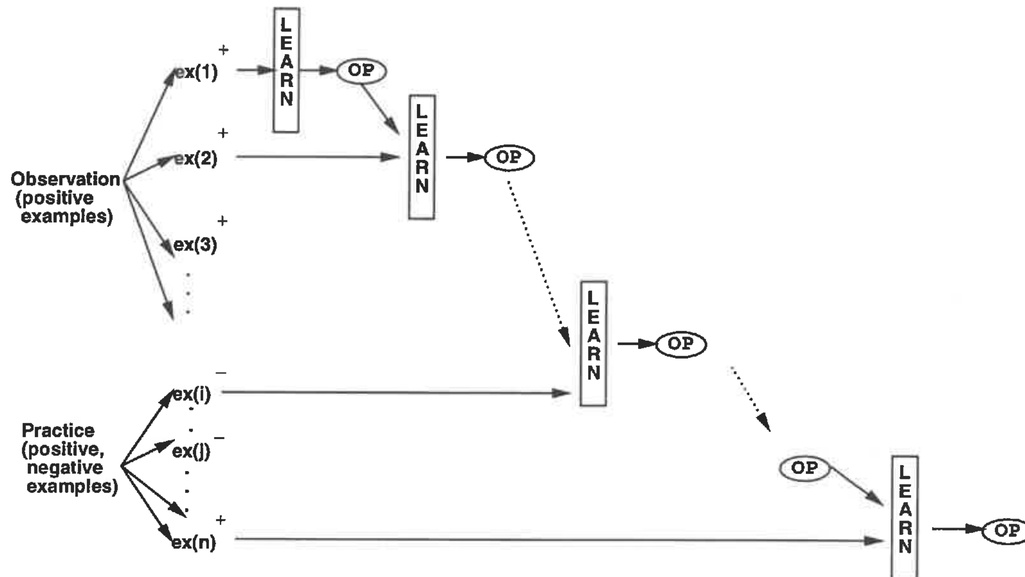


Figure 2.7: Incremental learning of planning operators. The observations of expert solution traces provide positive examples, while OBSERVER's own execution traces provide both positive and negative examples. The positive examples are annotated with "+," and the negative examples with "-."

the literals held in common in positive examples. The operators thus learned usually have extraneous preconditions and are overly-specific. If OBSERVER uses these operators for planning during practice, then the associated behavior for planning and execution is conservative in the sense that it usually generates correct plans, but these plans usually contain unnecessary operators. The planner often misses plans that could be generated if the operators were correct.

Algorithms such as FOIL [Quinlan, 1990] and SAGE [Langley, 1985] learn a concept in a general-to-specific fashion by identifying literals that separate the positive examples from the negative ones. The operators thus learned usually miss some true preconditions. If OBSERVER uses these operators for planning, it usually does not miss correct plans, but may generate incorrect plans with missing steps.

The version-spaces approach [Mitchell, 1978] incorporates aspects of both methods by learning a most specific hypothesis S-set and a most general hypothesis G-set. The concept being learned is delimited by the two boundaries. The S-set is generalized whenever a positive example is given. The G-set is specialized upon every negative example, so that the learned concept does not include any negative examples. Using version-spaces like approaches has the following advantages:

- It is incremental

- It gives the system a self-knowledge of what it knows about the operators and where the correct concept is (i.e., the true preconditions lie within the boundaries delimited by the S-set and G-set). This is especially important during practice in order to generate useful learning examples—an example is only useful for learning if it lies in the boundaries delimited by the S-set and G-set

- The G-set can be used as operator preconditions for planning to generate an initial plan quickly for practice problems. Executing the plan in the environment generates effective positive and negative learning examples

- The S-set can be used during plan repair to identify what additional literals must be achieved in order to make the failed operator applicable

The major problem with version-spaces like approaches is the complexity of computing the S-set and G-set when learning in structural domain. [Haussler, 1989] has shown that the size of the S-set and G-set can both grow exponentially and, hence, updating them is exponentially hard. To address this, OBSERVER uses an incremental algorithm to learn the operator preconditions by building a general representation, the **G-rep**, and a specific representation, the **S-rep**, in a manner similar to the version-spaces method [Mitchell, 1978]. The **G-rep** is a conjunct of literals that represents a general boundary of the concept to be learned, and the **S-rep** is a conjunct of literals that represents a specific boundary. The **G-rep** is different from the most general boundary, the G-set, in version-spaces in that it is not guaranteed to be the most general representation for the negative examples. The **G-rep** contains concepts that are more general than would be included in the S-set, but it can be updated in polynomial time. Similarly, the **S-rep** differs from the S-set in that it is not guaranteed to be the most specific representation for the set of positive examples. The **S-rep** contains concepts that are more specific than would be included in the S-set, but can be updated in polynomial time.

The key ideas of OBSERVER's algorithm for learning operator preconditions include the following, as illustrated in Figure 2.8:

- OBSERVER keeps a *single* representation in the **S-rep** and **G-rep** instead of maintaining a *set* of plausible hypotheses in the S-set and G-set in the version-spaces approach. Keeping a single representation reduces the planning search space, because each representation corresponds to a different operator.

- The **S-rep** is initialized by parameterization of the pre-state of the first observation, and updated using both observations and successful executions from practice. The **G-rep** is initialized to the empty set and updated using unsuccessful executions during practice

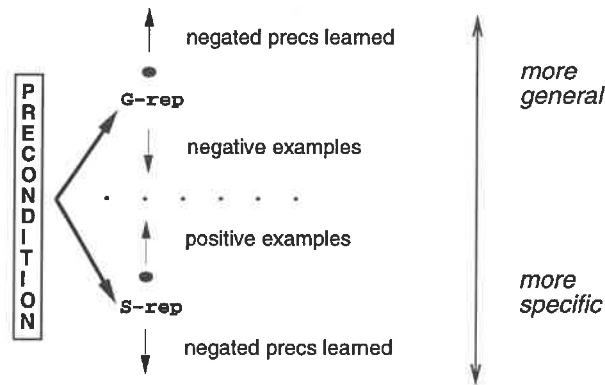


Figure 2.8: Learning operator preconditions. The operator preconditions are represented by the **G-rep** and the **S-rep**, which represent a general and specific boundary of the preconditions, respectively. The **S-rep** is generalized given positive examples, but can be made more specific upon learning negated preconditions. The **G-rep** is specialized given negative examples, but can be made more general upon learning negated preconditions.

- The **S-rep** is a specific generalization of the pre-states of the positive examples, although it is *not* always the *most* specific generalization. When learning by observation, OBSERVER conservatively generalizes the **S-rep** by removing from it those literals that cannot be in any generalizations, without finding the most specific common generalizations of the operator precondition and the pre-state of the new observation. We use a heuristic that conservatively generalizes a constant to a variable, similar to learn-one-disjunct-at-a-time [Vanlehn, 1987].

  There is a tradeoff between the degree of generalization and the efficiency of generalization—the usual version-spaces approach learns the *most specific* boundary of the concept, but the time complexity for updating the boundary is exponential. OBSERVER's approach updates the specific boundary in polynomial time, but the boundary is not guaranteed to be *most specific.* OBSERVER trades off the degree of generalization for the computational efficiency during updating.

  When refining the **S-rep** during practice, since the bindings for the variables in the operator are known, the **S-rep** can be generalized by removing from it those preconditions, substituted with the bindings, that are not members of the pre-state

- The **G-rep** is a general boundary of the concept learned from near miss negative examples. A near miss is an example for which only one literal is not covered by the target concept. Thus the **G-rep** is *not* always the *most* general boundary as the G-set in the version-spaces method.

  There is also a tradeoff between the degree of specialization and the efficiency of

specialization—the usual version-spaces approach learns the *most general* boundary of the concept, but the time complexity for specialization the boundary is exponential. `OBSERVER`'s approach updates the general boundary in polynomial time, but the boundary is not guaranteed to be *most general*. `OBSERVER`'s approach trades off the degree of specialization for the computational efficiency of specialization

- `OBSERVER` can learn negated preconditions. Sometimes `OBSERVER` over-generalizes the **S-rep** because it first assumes there are no negated preconditions. But it compensates for this simplifying assumption by learning negated preconditions when they are detected during practice
- Because of the presence of negated preconditions, the **S-rep** can also be specialized and **G-rep** can be generalized when learning negated preconditions

2.5.3 Learning operators effects

Learning operator effects is to learn the regular effects of an operator, as well as conditional effects with their corresponding conditional preconditions. The regular effects are learned by identifying the commonalities of the delta-states of the positive examples. The conditional effects are inferred from the subset of the delta-states that are *not* common to all the delta-states of the examples. Each conditional effect has its own conditional preconditions that determine whether this conditional effect occurs when the operator is applied. The conditional precondition for each conditional effect is also learned by keeping its own **G-rep** and **S-rep**.

`OBSERVER` learns operator effects incrementally as they occur in the delta-states of the observations or practice episodes. It does not try to conjecture all the possible conditional effects before they are observed. While learning operator effects, `OBSERVER` initializes the effects to the parameterized delta-state of the first observation. Given a new observation, `OBSERVER` incorporates Occam's Razor [Blumer *et al.*, 1987] to minimize the number of conditional effects learned. For each operator effect, `OBSERVER` finds all the literals in the delta-state of a new observation or successful execution that are unifiable with it. If the effect does not have any possible unifications, then it is learned as a conditional effect. If an effect has exactly one possible unification, then the effect can be updated by generalizing constants to variables, if appropriate. Otherwise, the effect is kept as it is. Every literal in the delta-state that is not unifiable with any effect is parameterized and learned as a conditional effect.

Refining operator effects during practice is similar to learning operator effects by observation, except that the variable mappings between the effects and delta-states are already determined by the planner.

OBSERVER learns operator effects in polynomial time, as proven in Chapters 3 and 5.

2.5.4 Learning the variable bindings—connecting the preconditions and effects

The pre-state and delta-state always share some common objects. For example, in the observation shown in Figure 2.2, the objects that appear in the delta-state of the observation, such as part0, drill0, and vise0, also appear in the pre-state of the observation. Similarly, the preconditions and effects of an operator share common variables. For example, in the operator shown in Figure 2.5, the variables <hd>, <part>, and <machine> are shared by the preconditions and effects.

Knowing the bindings for the variables reduces the ambiguity in matching the preconditions of an operator against the pre-states, as well as in matching the effects against the delta-states. An extreme case of this is updating operators from OBSERVER's own executions, where bindings for all the variables are determined by the planner. In this case, it is trivial to generalize the **S-rep** of the operator: all that is required is to check whether a precondition in the **S-rep**, substituted using the variable bindings, is a member in the pre-state; however, when updating operators from observations, the variable bindings are not known. How can we learn the variable bindings?

In the observations given to OBSERVER, the delta-states are usually much smaller than the pre-states of observations, because delta-states do not have any irrelevant information, whereas irrelevant information abounds in pre-states. Therefore, matching the effects against the delta-states usually have much less ambiguity than matching the preconditions against the pre-states. Thus, OBSERVER first matches the effects against the delta-state to learn bindings for some variables in the operator. These bindings are then employed in the refinement of preconditions.

We assume that different variables in an operator are never bound to the same value. This is reasonable in almost all the domains, because operators that need to bind distinct variables with the same values are rare. For example, this assumption holds in our two application domains. The variables in an operator are initialized by parameterizing the objects in the first observation of the operator. Every object is uniquely parameterized to one variable. The domain constants are not not parameterized given the first observation; however, they may be further generalized when a different constant is observed later. The type of each variable is determined using the domain object ontology (i.e., type hierarchy) given to OBSERVER as initial knowledge. Variables are removed from the operator if they are no longer needed when the preconditions and the effects are generalized.

2.5.5 Summary of the learning method

Operators are learned in two phases: first, they are learned from the observations of expert solutions, then they are refined through practice. During learning by observation, OBSERVER initializes the operator by parameterizing the first observation. Given other observations, OBSERVER first learns variable bindings by matching the effects against delta-state. It then updates the effects and generalizes the **S-rep** of the operator. During operator refinement by practice, OBSERVER generalizes the **S-rep** and updates the effects with positive examples. It learns negated preconditions (hence, specializes the **S-rep**) and specializes the **G-rep** with negative examples. Table 2.2 shows what steps are involved in learning and refining operators.

	observation (positive examples) (Chapter 3)	successful execution (positive examples) (Chapter 5)	unsuccessful execution (negative examples) (Chapter 5)
learning variables	yes	yes	no
constant-to-variable	yes	no	no
initialize **S-rep**	first observation	no	no
initialize the **G-rep**	the empty set	no	no
generalize the **S-rep**	yes	yes	no
specialize the **G-rep**	no	no	yes
learn negated preconds	no	no	yes
update regular and conditional effects	yes	yes	no

Table 2.2: This table summarizes what components (variables, preconditions, and effects) of the operators are being learned by observation and practice. The inputs for learning consist of the positive examples from observations of expert solution traces, as well as from positive and negative examples collected during practices.

2.6 Planning while Learning Operators

The operators learned by observation are usually incomplete and incorrect. This incorrectness and incompleteness present difficulties for planning and execution, as classical planners presume a correct domain model. To solve this problem, OBSERVER interleaves planning and execution. It refines the operators based on its own execution traces, including the successful and unsuccessful executions, and repairs the plan upon execution failure. This section gives a high-level description of how OBSERVER integrates planning, execution, and learning.

We have built a simulator of the external environment. The simulator uses a complete and correct set of operators to model the available actions (note that OBSERVER does not have access to this complete domain), similar to the simulators used in [Shen, 1994, Gil, 1992]. The simulator also has complete and correct knowledge of the state. When OBSERVER executes an action, the simulator checks if the preconditions of the corresponding operator used in the simulator are satisfied in the current state. If so, the simulator updates its current state and provides the complete descriptions of the pre-state and post-state of the execution to the learning system. Otherwise, the state is not changed, and the simulator signals execution failure.

2.6.1 Interleaving planning and execution

When solving a problem using a set of imperfect operators, OBSERVER first generates an initial plan that achieves the preconditions in the **G-rep** of each operator, but does not require achieving preconditions in the **S-reps** of the operators. Then, OBSERVER executes the plan in the simulator. During execution, if an operator executes successfully when some preconditions in the **S-rep** are not satisfied in the state, the unsatisfied preconditions are removed from the **S-rep**, resulting in a more accurate operator description. An operator may fail to achieve its intended effects due to unmet preconditions, necessitating plan repair and operator refinement.

2.6.2 Planning and plan repair

OBSERVER repairs a plan after an execution failure in order to bring the system to a state in which the failed operator can be successfully executed, and in order to generate learning opportunities for operator refinement. During plan repair, OBSERVER uses the **S-rep** of the failed operator to determine which additional preconditions to achieve to make the failed operator applicable. The plan repair proceeds as follows:

1. OBSERVER chooses a precondition randomly from unsatisfied **S-rep** preconditions, and calls the planner to achieve this additional precondition, as well as all the **G-rep** preconditions.

2. If the planner successfully generates a subplan to achieve the chosen precondition, this subplan is concatenated with the rest of the original plan to form the repaired plan. The repaired plan is immediately executed, without achieving other preconditions in the **S-rep**. Further plan repair may be necessary if the failed operator has more than one unsatisfied **S-rep** precondition, or if the subplans generated during plan repair fail during execution.

3. If OBSERVER cannot find any plan segments to achieve any additional preconditions in the **S-rep**, it conjectures negated preconditions. It adds the negated preconditions to the **S-rep** and generates plan segments to achieve the conjectured negated precondition for plan repair.

4. If all of the above fails, OBSERVER revises the original plan by abandoning the failed operator and by proposing a different operator to achieve the goal that the failed operator achieves.

The process of planning, execution, and plan repair continues until the problem is solved, i.e., the goals are satisfied in the environment, or until a preset resource bound is exceeded, or until the algorithm can no longer generate repaired plans. Some alternative methods for plan repair and for integrating planning and execution, as well as their advantages and disadvantages, are discussed in Chapter 4.

2.7 Summary

This chapter defined the scope of the thesis, i.e., learning planning operators, including their preconditions and effects, without any initial knowledge of the operators. Our approach is to learn planning operators by observing expert solution traces and by practicing in a learning-by-doing fashion [Anzai and Simon, 1979].

The initial knowledge given to OBSERVER includes only the domain object ontology, the description language of the domain, and static domain knowledge. Given observations of expert solution traces and practice problems, OBSERVER learns operators from observations as well as from OBSERVER's own execution traces. Thus, the learning system consists of three components: *(i)* learning operators from observation, *(ii)* solving practice problems, by planning and plan repair using learned operators and by executing plan steps in the environment, and *(iii)* refining operators during practice. OBSERVER learns a specific representation, the **S-rep**, and a general representation, the **G-rep**, of operator preconditions, in a manner similar to the version-spaces method. Both the **G-rep** and **S-rep** are updated in polynomial time. OBSERVER also incrementally learns operator effects, including conditional effects and conditional preconditions, in polynomial time. The following chapters elaborate the main ideas discussed in this chapter.

Chapter 3

Learning Operators by Observation

A primary contribution of this dissertation is a method of learning planning operators from observations of expert's solution traces. Our approach is motivated by three factors: *(i)* humans learn by observing others, *(ii)* it is usually easier for a domain expert to solve a problem than to explain the rules he/she uses, and *(iii)* the traces used for learning are observable when experts solve problems.

The task of learning by observation is essentially learning the preconditions and effects of the operators by generalizing the pre-states and post-states of the observations. In Section 3.1, we describe the input and output of our learning algorithm. Section 3.2 describes how the algorithm generate an initial operator description from the first observation. Section 3.3 describes the details of learning operators incrementally by observation, which includes learning the variable bindings by matching the effects against the delta-states, updating the **S-rep** of the operator preconditions, and updating the operator effects. Section 3.4 analyzes the complexity of the learning method and proves that it is polynomial. Section 3.5 discusses issues related to learning by observation. Finally, Section 3.6 summarizes the chapter.

3.1 Input and Output

Formally, an observation is a triple `<N, P, D>`, where `N` is the name of the operator being applied, `P` is the pre-state of the observation, and `D` is the delta-state of the observation. The pre-state of an observation is a conjunct of ground literals that describes the state in which an operator is applied. The delta-state is a pair `<ADD, DEL>`, where `ADD` and `DEL` are both conjuncts of ground literals describing the literals that are added to and deleted from the pre-state after the operator is applied. The delta-state is the difference between the post-state and pre-state. An example of an observation is given in Figure 3.1.

op-name: `hold-with-vise`

Pre-state:

```
(has-device drill0 vise0)
(on-table drill0 part0)
(is-clean part0)
(is-empty-holding-device vise0 drill0)
(is-available-table drill0 vise0)
(holding-tool drill0 spot-drill0)
(is-available-part part0)
(hardness-of part0 hard)
(material-of part0 iron)
(size-of part0 width 2.75)
(size-of part0 height 4.25)
(size-of part0 length 5.5)
(shape-of part0 rectangular)
```

Delta-state:

adds:

```
(holding drill0 vise0 part0 side5)
```

dels:

```
(is-empty-holding-device vise0 drill0)
(on-table drill0 part0)
(is-available-part part0)
```

Figure 3.1: An observation of the state before and after the application of the operator **HOLD-WITH-VISE**. This examples is a copy of the example shown in Figure 2.2. This figure illustrates the delta-state, which is the difference between the post-state and the pre-state of the observation. Many literals in the pre-states are irrelevant for the successful application of the operator **HOLD-WITH-VISE**, e.g., the sizes of the part and the material of the part.

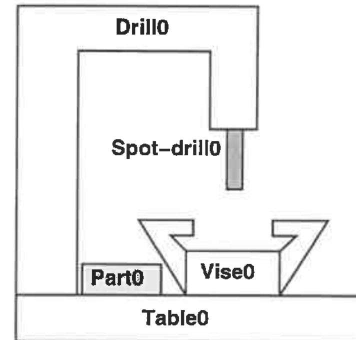


Figure 3.2: Pre-state of observation in Figure 3.1.

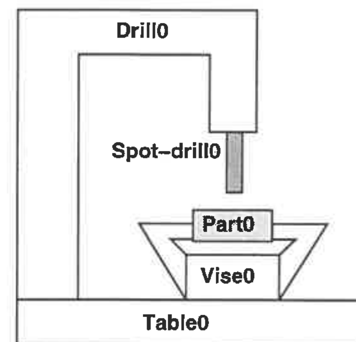


Figure 3.3: Post-state of observation in Figure 3.1.

Learning is triggered upon observations of expert solution traces. The pre-states of these observations are generalized incrementally in a specific-to-general manner to form the **S-rep** of the operator preconditions. The delta-states are generalized to form the effects of the operator. Since the **G-rep** is only updated using negative examples, it is an empty conjunct at the end of learning by observation.

3.2 Initializing the Operators

Given the first observation of an operator, OBSERVER initializes the specific representation, the **S-rep**, to the parameterized pre-state of the observation, and initializes the effects to the parameterized delta-states. During parameterization, each object (except for *domain constants*) is uniquely generalized to a typed variable. Domain constants are given to OBSERVER as part of the input; they are only generalized to variables if OBSERVER notices a different constant in a later observation. The type for each variable in the operator is the most specific type in the type hierarchy for the corresponding object. For example, given the first observation of the operator HOLD-WITH-VISE shown in Figure 3.1, OBSERVER generates an initial operator as shown in Figure 3.4. Note that constants such as **width**, **rectangular**, and **iron** are not generalized to variables. Also note that the **S-rep** learned from this one observation has many extraneous preconditions such as (size-of <v1> height 2.75). They will later be generalized or removed with more observations and/or practice.

3.3 Learning Operators Incrementally

These initial operators are generalized incrementally with more observations. First, OBSERVER matches the effects of the operator against the delta-state of the new observation (Section 3.3.1). The matching produces partial bindings for variables in the operator. Then, OBSERVER uses the partial bindings to update the specific representation **S-rep** of the operator preconditions (Section 3.3.2). And, finally, the effects of the operators are updated (Section 3.3.3).

3.3.1 Learning variable bindings of the operators

This section describes in detail the procedure **learn_variable_bindings**, which learns bindings by matching the effects of the operator against the delta-state of the observation.

Our algorithm for learning variable bindings incorporates Occam's razor, which prefers a simpler explanation given two explanations of the data and all other things being equal.

```
(operator hold-with-vise
  (preconds ((<v3> Drill) (<v2> Vise) (<v1> Part) (<v5> Spot-drill))
   (and (has-device <v3> <v2>)
        (is-available-table <v3> <v2>)
        (is-empty-holding-device <v2> <v3>)
        (holding-tool <v3> <v5>)
        (size-of <v1> width 2.75)
        (size-of <v1> height 4.25)
        (size-of <v1> length 5.5)
        (shape-of <v1> rectangular)
        (on-table <v3> <v1>)
        (hardness-of <v1> soft)
        (is-clean <v1>)
        (material-of <v1> copper)
        (is-available-part <v1>)
  (effects
   (add (holding <v3> <v2> <v1> side5))
   (del (on-table <v3> <v1>))
   (del (is-available-part <v1>))
   (del (is-empty-holding-device <v2> <v3>))))))
```

Figure 3.4: Learned operator HOLD-WITH-VISE when the observation in Figure 3.1 is given to OBSERVER. The preconditions shown here is the **S-rep** of the operator preconditions.

Thus, in the presence of ambiguity during matching, the algorithm chooses the mapping that leads to fewer conditional effects being learned. This heuristic implies that any effect that is unifiable with a unique element in the delta-state should be unified; such pairs of effect and delta-state are called unambiguous_pairs. The formal definition of unambiguous_pair is given in Figure 3.5. In essence, a pair *(e d)*, where *e* is an effect and *d* is a member of a delta-state, is an unambiguous_pair if there is no element in the delta-state other than *d* that is unifiable with *e*, and if there is no effect other than *e* that is unifiable with *d*. Occam's Razor enforces that the elements in the unambiguous_pair be unified with each other so they are not learned as conditional effects.

Figure 3.6 gives the algorithm for learning the variable bindings. In steps 4-9, this procedure repeatedly finds unambiguous_pairs. For each unambiguous pair *(this-effect, this-delta-state)*, where *this-effect* is an effect of the operator and *this-delta-state* is a member of the delta-state of the observation, OBSERVER learns new *bindings* by unifying the unambiguous_pair *(this-effect, this-delta-state)*. This process continues until all the unambiguous_pairs are unified. Since the process of finding unambiguous_pairs is incremental, different orderings of effects may, in theory, result in different updates of effects;

definition: unambiguous_pair

Given: *bindings, effects(op), delta-state(obs)*
Define: unambiguous_pair*(e, d)*, where $e \in$ *effects(op)*, $d \in$ *delta-state(obs)* iff:

- for every $e' \in$*effects(op)*, $e' \neq e$, $\textbf{unify}(\textbf{substitute}(e', bindings), d) = \text{NIL}$, and
- for every $d' \in$*delta-state(op)*, $d' \neq d$, $\textbf{unify}(\textbf{substitute}(e, bindings), d') = \text{NIL}$

Figure 3.5: Definition of unambiguous_pair. A pair (*e,d*), where *e* is an effect and *d* is an element in the delta-state, is an unambiguous_pair if *d* is the only element in the delta-state that is unifiable with *e*, and *e* is the only element in the effects that is unifiable with *d*.

however, this has not posed any problems in our experiments.

In addition to learning the bindings of the variables in the effects, `OBSERVER` generalizes a constant to a variable if this constant is unified with a different constant in the bindings (steps 10-13). It also generalizes the type of a variable to the least common ancestor type of the type of the variable in the operator and the type of the corresponding object in the bindings (steps 14 and 15).

For example, given the initial operator shown in Figure 3.4 and a new observation shown in Figure 3.7, `OBSERVER` matches the effects of the operator against the delta-state of the observation. `(holding <v3> <v2> <v1> side5)` and `(holding milling-machine1 vise1 part1 side4)` is an unambiguous_pair. Unifying them results in learning *bindings* = {part1/`<v1>`, vise1/`<v2>`, milling-machine1/`<v3>`, side4/side5}. Other unambiguous_pairs are unified similarly. The type of the variable `<v3>` is generalized to the least common type of milling-machine and drill, i.e., machine, and the constant `side5` is generalized to a new variable `<v4>` whose type is `Side`.

3.3.2 Updating the S-rep of operator preconditions

Figure 3.10 shows the procedure for updating the **S-rep** of the operator preconditions from an observation. `OBSERVER` updates the **S-rep** by removing those literals that are not present in the pre-state of the observation, and by generalizing domain constants to variables if different constants are used in the pre-state. Determining which preconditions in the **S-rep** are not present in the pre-state of the observation is nontrivial when bindings for the variables are unknown. Different mappings of variables to objects result in different generalizations of the **S-rep**, and the number of possible mappings is exponential. To avoid the computational complexity, `OBSERVER` only removes **S-rep** preconditions that are definitely not present in the pre-state (i.e., is not unifiable with any literal in the

procedure: learn_variable_bindings

Given: observation *obs*, learned operator *op* corresponding to *obs*
Learn: *bindings*

1. *effects-to-match* ← **effects**(*op*)
2. *state-to-match* ← **delta-state**(*obs*)
3. *bindings* ← NIL
4. *(this-effect, this-delta-state)*
   ← **find_unambiguous_pair**(*effects-to-match, state-to-match, bindings*)
5. Repeat until there are no more unambiguous_pair
6. *(new-effect, bindings)*
   ← **unify**(*this-delta-state*, **substitute**(*this-effect,bindings*))
7. *effects-to-match* ← *effects-to-match* \ {*this-effect*}
8. *state-to-match* ← *state-to-match* \ {*this-delta-state*}
9. *(this-effect, this-delta-state)*
   ← **find_unambiguous_pair**(*effects-to-match, state-to-match, bindings*)
10. For every element (x/y) in *bindings*
    (x is an object in the delta-state, and y is a variable or constant in the effect)
11. If y is a constant
    then
12. *new-variable* ← **parameterize**(x,y)
13. **type_of**(*new-variable*) ← **least_common_ancestor_type**(x, y)
    otherwise
14. **type_of**(y) ← **least_common_ancestor_type**(x, y)
15. Return *bindings*

Figure 3.6: Learning variable bindings for operator by matching the effects of the operator against the delta-state of the observation. Occam's Razor is incorporated to minimize the number of learned conditional effects in the operator.

pre-state) under all the possible bindings for the variables in the operator, provided that they are consistent with the bindings returned by **learn_variable_bindings**.

To determine if a **S-rep** precondition, *prec*, is definitely not in the pre-state of an observation, OBSERVER unifies *prec*, substituted with the bindings returned by **learn_variable_bindings**, with each literal in the pre-state of the observation. During unification, every term in *prec*, including domain constants, is treated as a variable. We say that a literal *lit* in the pre-state is a *potential-match* of the precondition *prec*, iff: *(i)* they are unifiable, and *(ii)* let *new-bindings* ($\{p_1/l_1, p_2/l_2, ..., p_n/l_n\}$) = **unify**(substitute(*prec, bindings*), the number of P_i that are domain constants (as op-

op-name: `hold-with-vise`

Pre-state:

```
(has-device milling-machine1 vise1)
(on-table milling-machine1 part1)
(is-available-table milling-machine1 vise1)
(is-empty-holding-device vise1 milling-machine1)
(is-available-tool-holder milling-machine1)
(is-available-part part1)
(is-clean part1)
(size-of part1 length 4)
(size-of part1 width 3)
(size-of part1 height 2.25)
(shape-of part1 rectangular)
(hardness-of part1 soft)
(material-of part1 bronze)
```

Delta-state:

adds:

```
(holding milling-machine1 vise1 part1 side4)
```

dels:

```
(is-empty-holding-device vise1
milling-machine1)
(on-table milling-machine1 part1)
(is-available-part part1)
```

Figure 3.7: The second observation of the operator HOLD-WITH-VISE. The main differences from the first observation shown in Figure 3.1 are that *(i)* the machine in use is a milling-machine instead of a drill, *(ii)* there is no drill-tool being held by the machine, and *(iii)* the part is made of a different material and has a different size.

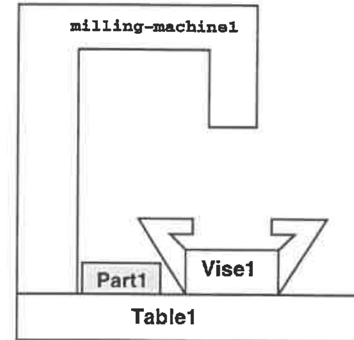


Figure 3.8: Pre-state of observation in Figure 3.7.

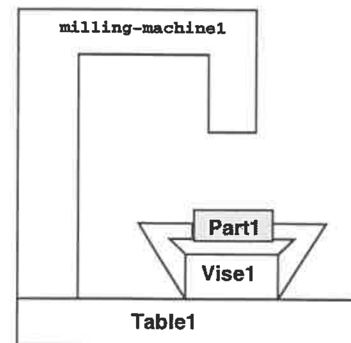


Figure 3.9: Post-state of observation in Figure 3.7.

posed to variables) is at most one. That is, *lit* has at most one constant that is different from *prec*.

In steps 3 and 4, if *prec* does not have any potential matches, then *prec* can not be present in the pre-state no matter what variable bindings are given, thus it is removed from the **S-rep**. Frequently this is because no literal in the pre-state has the same predicate name as *prec*. In steps 5 and 6, if *prec* has exactly one potential match, then *prec* is generalized

---

procedure: update_S_rep_from_observation

Input: *op, obs, bindings*
Output: the **S-rep**

1. for each *prec* $\in$ the **S-rep**
2. *potential-matches(prec)* $\leftarrow$ **find_potential_matches***(prec,pre-state(obs),bindings)*
3. if *potential-matches(prec)= NIL*
4. then **S-rep** $\leftarrow$ **S-rep** $\backslash\{prec\}$
5. if *potential-matches(prec)* has 1 element
6. then **S-rep** $\leftarrow$ (**S-rep** $\backslash\{prec\}$) $\cup$ **generalize**(*prec, bindings*)
7. if *potential-matches(prec)* has more than 1 element
8. then do not modify the **S-rep**

Figure 3.10: Updating the **S-rep** of the operator preconditions given a new observation. The algorithm is polynomial in the size of the pre-state.

as a result of the unification of *prec* with the unique potential match, *lit*, in the pre-state. If one constant in *prec* is unified with another constant in the pre-state, the constant in the **S-rep** is generalized to a variable. Otherwise, in steps 7 and 8, *prec* has several potential matches, thus `OBSERVER` avoids this ambiguity by keeping *prec* as it is in the **S-rep**.

In essence, `OBSERVER` allows at most one constant-to-variable replacement for each **S-rep** precondition. It generalizes a constant to a variable only when there is one difference between the precondition and the corresponding literal in the pre-state of the observation. This is similar to the concept of learning-one-disjunct-at-a-time [Vanlehn, 1987]. A careful ordering of traces by expert may, in theory, speed up the generalization. `OBSERVER`'s algorithm is still effective without that, as demonstrated in our empirical results, because the number of domain constants is relatively small compared with the number of objects.

For example, consider the operator shown in Figure 3.4 and a new observation shown in Figure 3.7. Procedure **learn_variable_bindings** learns a set of bindings *bindings* = {`<v1>/part1, <v2>/vise1, <v3>/milling-machine1,side5/side4`} by matching the effects of the operator against the delta-state of the new observation. Then, `OBSERVER` computes the *potential-matches* for each precondition in the **S-rep** of the operator giving *bindings*. We see that there are no *potential-matches* for `(holding-tool <v3> <v5>)` because no element in the pre-state has the predicate name `holding-tool`, i.e., for every *lit* in the pre-state **unify**(*lit*, `(holding-tool <v3> <v5>)`) = NIL. Note that `(size-of part1 width 3)` is a *potential-match* of `(size-of <v1> width 2.75)` because **unify**(`(size-of part1 width 3), (size-of <v1> width 2.75)`) = {(3/2.75)}, which has exactly one con-

stant substitution; however, (size-of part1 length 5.5) is not a *potential-match* of (size-of <v1> width 2.75), because **unify**((size-of part1 width 3), (size-of <v1> length 5.5)) = {(width/length,3/5.5)}, which requires two constant substitutions. Here are some examples of what **find_potential_matches** returns:

potential-match((holding-tool <v3> <v5>)) = NIL;
potential-match((size-of <v1> width 2.75)) = {(size-of part1 width 3)};

Therefore, ((holding-tool <v3> <v5>) is removed from the **S-rep**. (size-of <v1> width 2.75) is generalized to (size-of <v1> width <anything>) by unifying it with (size-of part1 width 3). Since <anything> is not constrained by any other preconditions in the **S-rep**, OBSERVER learns that the width of a part can be any value and is irrelevant to the operator and thus is removed from the **S-rep**. Similarly, the preconditions (size-of <v1> height 2.75), (size-of <v1> length 5.5), (material-of <v1> copper) are removed from the **S-rep**. The modified operator is shown in Figure 3.11, along with the extraneous preconditions that are removed from the initial operator.

3.3.3 Updating the operator effects

Figure 3.12 describes the procedure **update_effects_from_observation**, for updating the effects of an operator from an observation. OBSERVER first generalizes the effects that have unambiguous matches in the delta-state (steps 4-9). Steps 10-18 show how OBSERVER learns conditional effects. If an effect of the operator does not unify with any element in the delta-state of the observation, then it is learned as a conditional effect. The specific representation of the preconditions of this conditional effect is initialized to the **S-rep** of the preconditions of the operator (steps 10-13). Every element in the delta-state that does not unify with any effect is also learned as a conditional effect (steps 14-18), and its specific precondition representation is the parameterized pre-state of the current observation. OBSERVER does not update an effect that can be unified with more than one literal in the delta-states.

To illustrate how OBSERVER generalizes an effect, consider the operator HOLD-WITH-VISE as shown in Figure 3.4 and a new observation shown in Figure 3.7. Matching the effects of the operator against the delta-state of the observation generates the following bindings: {part1/<v1>, vise1/<v2>, milling-machine1/<v3>, side4/side5}. The effect (add (holding <v3> <v2> <v1> side5)) is generalized to (add (holding <v3> <v2> <v1> <v4>)) because the constant side5 in the effect is substituted with a different constant side4 in the delta-state. Other effects are kept as they are. Also, since the mapping between the effects and the *delta-state* is unambiguous, no conditional effects are learned. The updated operator is shown in Figure 3.11.

```
(operator hold-with-vise
  (preconds ((<v3> Machine) (<v2> Vise) (<v1> Part) (<v4> Side))
   (and (has-device <v3> <v2>)
        (is-available-table <v3> <v2>)
        (is-empty-holding-device <v2> <v3>)
        (shape-of <v1> rectangular)
        (on-table <v3> <v1>)
        (is-clean <v1>)
        (hardness-of <v1> soft)
        (is-available-part <v1>)
  (effects
   (add (holding <v3> <v2> <v1> <v4>))
   (del (on-table <v3> <v1>))
   (del (is-available-part <v1>))
   (del (is-empty-holding-device <v2> <v3>))))
```

extraneous preconditions:

```
(holding-tool <v3> <v5>)
(size-of <v1> width <v6>)
(size-of <v1> height <v7>)
(size-of <v1> length <v8>)
(material-of <v1> <v10>)
```

Figure 3.11: Learned operator `HOLD-WITH-VISE` when the observations in Figure 3.1 and 3.7 are both given to `OBSERVER`. The preconditions listed here are the literals in the **S-rep** of the operator. Note that the type of variable `<v3>` has been generalized to Machine from the initial operator shown in Figure 3.4.

The next example illustrates how conditional effects are learned. Given a new observation shown in Figure 3.13 and the operator that was shown in Figure 3.11, `OBSERVER` matches the effects of the operator against the delta-states of the observation. Bindings {drill2/`<v3>`, part2/`<v1>`, vise2/`<v2>`} are learned. The effects `(del (on-table <v3> <v1>))`, `(del (is-available-part <v1>))`, and `(del (is-empty-holding-device <v2> <v3>))` each has an unambiguous match in the delta-states, i.e., `(del (on-table drill2 part2))`, `(del (is-available-part part2))`, and `(del (is-empty-holding-device vise2 drill2))`, respectively. The effect `(add (holding <v3> <v2> <v1> <v4>))` is not unifiable with any literal in the delta-state and therefore is learned as a conditional effect of the operator. Delta-state `(add (holding-weakly drill2 vise2 part2 side0))` is not unifiable with any effect of the operator, and thus it is parameterized and learned as a conditional effect. The resulting operator with conditional effect is shown in Figure 3.16.

---

procedure: update_effects_from_observation

Input: *bindings*, learned operator *op*, new observation *obs*
Output: modified effects(op)

1. *effects* ← *Effects*(*op*)
2. *deltas* ← *delta-state*(*obs*)
3. for every $e \in$ **Effects**(*op*)
4. if there $\exists\ d \in$ *deltas* s.t. **is_unambiguous_pair***(e,d)* ;; Figure 3.5
   then
5. $e' \leftarrow$ **generalize***(e, bindings)*
6. replace e with e'
7. if e is a conditional effect, update its conditional preconditions
8. *effects* ← *effects* \ {*e*}
9. *deltas* ← *deltas* \ {*d*}
10. for every $e \in$ *effects*
11. if $\forall\ d \in$ *deltas*, **unify***(e, d1)* = NIL
    then ;;; learning a conditional effect
12. **conditional_effects***(op)* ← **conditional_effects***(op)* ∪ *e*
13. **S-rep**(e) ← **S-rep**(op)
14. for every $d \in$ *deltas*
15. if $\forall\ e1 \in$ *effects*, **unify***(e1, d)* = NIL
    then ;;; learning a conditional effect
16. *new-effect* = **parameterize***(d)*
17. **conditional_effects***(op)* ← **conditional_effects***(op)* ∪ {*new-effect*}
18. **S-rep**(*new-effect*) ← **parameterize**(*pre-state(obs)*)

Figure 3.12: Updating the effects of an operator from a new observation. Every effect that has a unique match with the delta-state is generalized according to the match. Every effect that is not unifiable with any element in the delta-state, and every element in the delta-state that is not unifiable with any effect, are learned as conditional effects.

op-name: `hold-with-vise`

Pre-state:

```
(has-device drill2 vise2)
(on-table drill2 part2)
(is-available-table drill2 vise2)
(is-clean part2)
(is-empty-holding-device vise2 drill2)
(holding-tool drill2 spot-drill2)
(is-available-part part2)
(size-of part2 diameter 4.75)
(size-of part2 length 5.5)
(material-of part2 bronze)
(hardness-of part2 soft)
(shape-of part2 cylindrical)
```

Delta-state:

adds:

```
(holding-weakly drill2 vise2 part2 side0)
```

dels:

```
(is-empty-holding-device vise2 drill2)
(on-table drill2 part2)
(is-available-part part2)
```

Figure 3.13: The third observation of the operator HOLD-WITH-VISE. Note that the effect has "holding-weakly" instead of "holding." OBSERVER thus learns conditional effects.

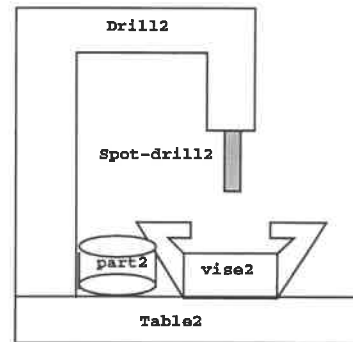


Figure 3.14: Pre-state of observation in Figure 3.13

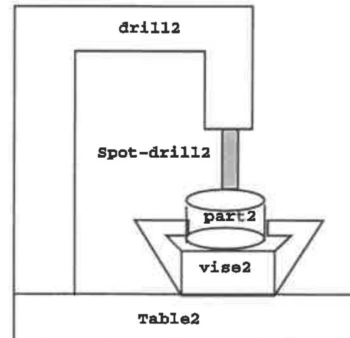


Figure 3.15: Post-state of observation in Figure 3.13

```
(operator hold-with-vise
  (preconds ((<v3> Machine) (<v2> Vise) (<v1> Part) (<v4> Side))
   (and (has-device <v3> <v2>)
        (is-available-table <v3> <v2>)
        (is-empty-holding-device <v2> <v3>)
        (on-table <v3> <v1>)
        (is-clean <v1>)
        (hardness-of <v1> soft)
        (is-available-part <v1>)
  (effects (<v6> spot-drill))
   (if (and (size-of <v1> diameter 4.75)
            (size-of <v1> length 5.5)
            (shape-of <v1> cylindrical)
            (hardness-of <v1> soft)
            (material-of <v1> bronze)
            (holding-tool <v3> <v6>))
       (add (holding-weakly <v3> <v2> <v1> side0)))
   (if (and (shape-of <v1> rectangular)
            (has-device <v3> <v2>)
            (is-available-table <v3> <v2>)
            (is-empty-holding-device <v2> <v3>)
            (on-table <v3> <v1>)
            (is-clean <v1>)
            (is-available-part <v1>)
       (add (holding <v3> <v2> <v1> <v4>)))
   (del (on-table <v3> <v1>))
   (del (is-available-part <v1>))
   (del (is-empty-holding-device <v2> <v3>)))))
```

extraneous preconditions:

```
(holding-tool <v3> <v5>)
(size-of <v1> width <v6>)
(size-of <v1> height <v7>)
(size-of <v1> length <v8>)
(material-of <v1> <v10>)
(shape-of <v1> <v11>)
```

Figure 3.16: Operator HOLD-WITH-VISE after learning from observations shown in Figures 3.1, 3.7, and 3.13. Note the conditional effects and conditional preconditions.

3.4 Complexity Analysis

This section analyzes the complexity of learning by observation.

Let:

n = maximum number of literals in a state description (either pre-state or post-state)
m = maximum number of arguments in a literal
l = maximum number of conditional effects in an operator

Thus:

the number of literals in the delta-states is $O(n)$
unifying two literals has time complexity $O(m)$

Initializing the operators

Parameterizing each literal in the pre-state and delta-state has time complexity $O(m)$. There are $O(n)$ literals in each pre-state or delta-state. Therefore, initializing an operator has time complexity $O(mn)$.

Learning the variable bindings

Learning variable bindings by matching the effect of an operator with the delta-state of an observation requires that every effect be unified with every literal in the delta-state. Since there are $O(n)$ literals in the effects of an operator and $O(n)$ literals in each delta-state, there are $O(n^2)$ pairs of literals to unify. Therefore the time complexity for learning the variable bindings is $O(mn^2)$.

Updating the S-rep

The computation for updating the **S-rep** comes from unifying every literal in the **S-rep** with every literal in the pre-state in **find_potential_matches**. There are $O(n^2)$ such pairs to unify, because there are $O(n)$ literals in each pre-state and $O(n)$ literals in each each **S-rep**. Since each unification has time complexity $O(m)$, the time complexity for updating the **S-rep** is $O(mn^2)$.

Updating the effects

Given variable bindings that are already learned, the effects of an operator is updated in time $O(mn)$, because there are $O(n)$ effects in an operator and each effect is updated in time $O(m)$. Each conditional precondition is updated in time $O(mn^2)$. Since the total number of conditional effects is bounded by l, the time complexity of updating conditional preconditions is $O(lmn^2)$.

3.5 Discussion

We now discuss some issues related to the two heuristics that are used in learning by observation.

3.5.1 Implication of Occam's Razor

Our algorithm prefers simpler effects, i.e., it learns operators with as few conditional effects as possible; however, if different alternatives have the same complexity, Occam's Razor does not favor one over another. For example, given an observation shown in Figure 3.17, the operator learned from it shown in Figure 3.19, and a new observation shown in Figure 3.18. There are two ways to update the operator effect using this new observation, as shown in Figure 3.20 and Figure 3.21. Both updates result in the same number of conditional effects being learned.

```
op-name: op-2
Pre-state:
     (H a)
     (H b)
     (F a b)
Delta-state:
adds: (add (P a b))
      (add (Q a b))
```

Figure 3.17: The first observation of the operator op-2.

```
op-name: op-2
Pre-state:
     (H c)
     (H d)
     (G c d)
Delta-state:
adds: (add (P c d))
      (add (Q d c))
```

Figure 3.18: The second observation of the operator op-2.

In theory, the above situation may occur; however, we have not yet encountered it in our experiments. To resolve this ambiguity, one could pick one update at random, or ask the expert to identify how the effects should be updated. Another way is for the expert to separate the ambiguous predicate into two distinct predicates so that OBSERVER can learn the correct effect automatically. For instance, if the effect (add (P <x> <y>)) should be learned as a regular effect and (add (Q <x> <y>)) should be learned as a conditional effect, then the predicate Q can be represented using two distinct predicates: Q1 for the first observation and Q2 for the second. The ambiguity is thus resolved.

op-name: op-2

```
(operator op-1
  (preconds ((<x> TYPE_a)
             (<y> TYPE_b))
    (and (H <x>)
         (H <y>)
         (F <x> <y>))
  (effects
    ((add (P <x> <y>))
     (add (Q <x> <y>))
```

Figure 3.19: The learned operator from observation in Figure 3.17.

bindings = {c/<x>, d/<y>}

```
(operator op2
   (preconds ((<x> Type))
     (and (H <x>)
          (H <y>)))
   (effects
    ((add (P <x> <y>)))
     (if (F <x> <y>)
         (add (Q <x> <y>)))
     (if (G <x> <y>)
         (add (Q <y> <x>)))))
```

Figure 3.20: **scenario 1:** Learned operator op-2. (Q <x> <y>) is learned as a conditional effect.

bindings = {d/<x>, c/<y>}

```
(operator op2
   (preconds ((<x> Type))
     (and (H <x>)
          (H <y>)))
   (effects
    ((add (Q <x> <y>)))
     (if (F <x> <y>)
         (add (P <x> <y>)))
     (if (G <y> <x>)
         (add (P <y> <x>)))))
```

Figure 3.21: **scenario 2:** Learned operator op-2. (P <y> <x>) is learned as a conditional effect.

3.5.2 Learning the S-rep

One way to learn the **S-rep** is to find the maximal specific common generalization of the **S-rep** and the pre-state of the new observation. In our application domains, the number of facts in the pre-state and post-state are typically much larger than the number of preconditions and effects of the corresponding operators, because many facts in the state are not relevant to the operator. For example, in the process planning domain, the pre-state and post-state typically include fifty to seventy assertions, whereas an operator usually has two to six preconditions or effects. In the absence of background knowledge for identifying the portion of the world state relevant to the planning operator, it is computationally expensive to find the maximally specific common generalization of the pre-states. In fact, Haussler [Haussler, 1989] has proven that finding such generalizations is NP-complete. OBSERVER does not learn the *most* specific generalization. Instead, it trades off the amount of generalization for the efficiency of generalization. OBSERVER generalizes the **S-rep** by a simple unification of each **S-rep** precondition with every literal in the pre-state of the observation. A precondition is removed iff it is not unifiable with any literal in the pre-state and the **S-rep** is not generalized in the presence of ambiguity.

3.6 Summary

This chapter presented details of algorithms for learning operators by observation. The input for learning are observations of expert's solution traces, {*obs: (pre-state(obs), delta-state(obs), op-name)*}. Each observation consists of the operator name, the pre-state, and post-state.

Given these observations, OBSERVER learns the **S-rep** of the operator preconditions by generalizing the pre-states of the observations in an efficient specific-to-general manner that does not find the most specific common generalization, and therefore avoiding the computational intractability. The generalization process first matches the effects of the operator against the delta-state of the observation to learn a set of partial bindings for the variables in the operators. It then removes extraneous preconditions based on learning the *potential-matches* for each precondition. Effects are learned by generalizing every element in the delta-states of the observations. The literals that occur in some delta-states of the observations but not in other delta-states are learned as conditional effects. OBSERVER processes observations in polynomial time.

Chapter 4

Planning while Learning Operators

The previous chapter presented an algorithm for learning tentative operators from observation. These initial operators often include overly-specific preconditions, overly-general preconditions, and incomplete effects. To further refine operators, OBSERVER uses them to solve practice problems and refines them based on plan execution traces. Refinement requires that planning, execution, and learning be integrated. In our approach, when solving a problem using incomplete and incorrect operators, OBSERVER first generates an initial plan that achieves the preconditions in the general representation (**G-rep**) of each operator but does not require achieving preconditions in the specific representation (**S-rep**) of the operator. The planner repairs the plan when execution failures arise, by using the **S-rep** to determine which additional preconditions to achieve to make the failed operator applicable.

This chapter describes how OBSERVER practices with initially incomplete and incorrect operators. Section 4.1 describes ways in which operators can be incomplete and incorrect and the impact of this incorrectness on the effectiveness of classical planning systems. Section 4.2 discusses issues relevant to planning with imperfect operators. Section 4.3 presents a framework for integrating planning, execution, and learning, and describes algorithms for planning with imperfect operators and for plan repair upon execution failure. Section 4.4 describes some implementation details. Section 4.5 delimits the generality of our approach. Finally, Section 4.6 summarizes this chapter.

4.1 Domain Knowledge Imperfections

Operators can be incomplete and incorrect in the following ways [Huffman *et al.*, 1993]: *overly-general preconditions, overly-specific preconditions, incomplete effects, extraneous*

effects, and missing operators. In this section, we explain why these types of domain knowledge imperfections occur in OBSERVER and how they affect planning.

1. *Overly-specific preconditions:* The **S-rep** of the operator preconditions can have unnecessary preconditions and thus be too specific. Planning with *overly-specific preconditions* forces the planner to do unnecessary search while achieving the extraneous preconditions of the operators. In cases where the extraneous preconditions are not achievable, the planner cannot find solutions to problems that are solvable with correct operators.

   The **G-rep** may also be overly-specific due to OBSERVER's initial assumption that there are no negated preconditions. If OBSERVER adds a precondition to the **G-rep** in response to a near miss negative example, and if the true reason for the unsuccessful execution of the operator is due to a missing negated precondition (instead of the unsatisfied precondition in the **S-rep**), then the **G-rep** is over-specialized.

2. *Overly-general preconditions:* The **S-rep** of the operator preconditions can be *overly-general*—sometimes OBSERVER over generalizes the **S-rep**, because it first assumes there are no negated preconditions. But it compensates for this simplifying assumption by learning negated preconditions when they are detected during practice. The **G-rep** of the operator precondition is general by definition. While OBSERVER plans with *overly-general preconditions*, some true preconditions of the operators in the plan are not considered. The plan may therefore miss some steps and thus, an operator in the plan may fail to execute in the environment because of unmet preconditions.

3. *Incomplete effects:* Some effects of an operator may not have been learned before the operator is used during planning. For example, the effects of an operator can also be masked by the pre-state—a literal that would otherwise be added in the effects of an operator will not be detected by OBSERVER if the literal is already present in the pre-state of the corresponding observation. These masked effects are not learned as effects of the operators from one observation. They may be learned from future observations of expert solutions and/or OBSERVER's own successful executions where the effects are not masked. Incomplete effects may cause the planner to have an incorrect internal model of the state and, hence, the planner may generate incorrect plans.

4. *Extraneous effects:* Since OBSERVER records only effects that it sees, and because the sensors are noise-free, no extraneous effects are learned.

5. *Missing operators:* An operator is missing from the domain knowledge if it is never observed by our learning system. The only way to find the missing operator is by observing an expert using it. The planner cannot solve a problem if the solution requires a *missing operators.*

4.2 Issues in Planning while Learning

Learning operators automatically and incrementally during practice complicates the planning system in the following ways:

1. Classical planners presume a correct domain model. In our learning system however, the newly acquired operators are possibly incomplete and incorrect. How can the planner generate plans to solve practice problems effectively and in the meantime generate learning opportunities for refining operators using incomplete and incorrect domain knowledge?

2. Because of incomplete and incorrect operators used during practice, the plans generated by the planner may be incorrect, which in turn may lead to execution failures. Thus, plan repair upon execution failure is necessary. How can the planner effectively repair the incorrect plan using incomplete and incorrect operators?

3. How can planning and execution be interleaved so that the system can solve practice problems effectively and concurrently generate learning opportunities for refining operators using incomplete and incorrect domain knowledge?

4.3 Planning with Incorrect Operators and Plan Repair

OBSERVER learns both the **S-rep** and **G-rep** of the operator preconditions. Either the **S-rep** or **G-rep** can be used for planning, each with different implications for the performance of learning and planning. Using the **S-rep** for planning leads to conservative system behavior in that generated plans usually have all the necessary steps but also have unnecessary extra steps. Therefore, the plan steps can be executed successfully to achieve their intended effects (except in the presence of incomplete operator effects); however, the system would not have the opportunity to learn which preconditions are extraneous. Using the **G-rep** for planning leads to more radical system behaviors in that the plans may miss some steps because some unknown true preconditions may not

be achieved by the planner and, hence, execution failures may occur. The system can, however, learn extraneous preconditions if a plan step is executed successfully.

Since OBSERVER's goals include both solving problems and learning operators, our approach uses the **G-rep** of the operator preconditions for planning. In this approach, the individual plans generated for achieving the top-level goals achieve the preconditions in the **G-rep** of each operator, but do not require achieving preconditions in the **S-rep** of the operators. This has the advantage of generating an initial plan quickly and of generating opportunities for operator refinement. During plan repair, the preconditions in the **S-rep** are used to identify which additional preconditions should be achieved to make the failed operator applicable.

4.3.1 The Plan Repair Algorithm

The plan repair algorithm is invoked upon OBSERVER's execution failures. The algorithm is given the following inputs:

- *op*, the operator that fails in the environment
- *top-goals*, the initial goals of the problem being solved
- *current-state*, the state in which the operator fails to execute, and
- *plan*, the remaining plan steps to be executed in order to achieve *top-goals*

In a nutshell, the plan repair algorithm first tries to generate a plan segment to achieve one of the unmet preconditions in the **S-rep** of the failed operator. If this fails, it then conjectures negated preconditions and tries to generate a plan segment to achieve one of the negated preconditions. If this also fails, it generates a plan that does not use the failed operator, but achieves the same goals as the failed operator.

Figure 4.1 gives a detailed description of the plan repair algorithm. In step 1, OBSERVER computes *unmet-preconds*, the preconditions in the **S-rep** of the failed operator *op* that are not satisfied in the *current-state*. There are usually many such *unmet-preconds*. Since some *unmet-preconds* may be extraneous preconditions of *op*, achieving all of them during plan repair not only results in added search, but also prevents the system from generating training examples for removing extraneous preconditions. Therefore, OBSERVER generates a plan segment to achieve only *one* unmet precondition, as described in steps 2-8. First, OBSERVER chooses one precondition from *unmet-preconds* to achieve. Then, it calls the basic planner to generate a plan segment to achieve *goals-to-achieve* (i.e., the union of the chosen precondition and all the preconditions in the **G-rep**). If a plan segment is found, a repaired plan is returned by concatenating the plan segment with *plan*, the

remaining plan steps to achieve *top-goals*; otherwise, the plan repair algorithm randomly chooses another precondition (which may be the same as the previously chosen one) from *unmet-preconds* to achieve by repeating steps 3-8. If *unmet-preconds* is the empty set, or if OBSERVER cannot find a plan segment to achieve any unmet preconditions in the **S-rep**, then OBSERVER conjectures negated preconditions and adds them to *unmet-preconds*, as described in step 9. Then, the plan repair algorithm repeats steps 3-8 to generate a plan segment to achieve *one* conjectured negated precondition. If the planner still cannot generate a plan segment to achieve any unmet precondition, then it generates a plan to achieve the effect that the failed operator *op* intended to achieve that does not use *op* (step 12).

---

procedure: plan_repair_for_failed_op

Inputs: *op*, the operator that fails to execute in the environment,
top-goals, goals of the problem to solve,
current-state, a list of assertions that are true in the state,
plan, the remaining plan steps to be executed
Output: *repaired-plan*, a repaired plan for the failure

1. *unmet-preconds* ← S-rep(*op*) \ *current-state*
2. Repeat until a *repaired-plan* is found:
3. *precond* ← **Choose_a_precond_to_achieve**(*unmet-preconds*)
4. *goals-to-achieve* ← G-rep*(op)* ∪ {*precond*}
5. *plan-segment* ← **Plan**(*goals-to-achieve, current-state*)
6. If (*plan-segment* ≠ FAIL)
   then ;; found a plan segment to achieve the precondition, return
7. *repaired-plan* ← **Concatenate**(*plan-segment, op, plan*)
8. Return *repaired-plan*
9. *unmet-preconds* ← **Conjecture_negated_preconds**
10. Repeat steps 3-8 for the new *unmet-preconds*
11. If (*plan-segment* = FAIL)
    then ;; abandon the original plan, replan from scratch
12. *new-plan* ← **Plan**(**Goals-of**(*op*) ,*current-state*)
13. Return **Concatenate**(*new-plan,plan*)

Figure 4.1: OBSERVER's plan repair algorithm. The algorithm generates a plan segment to achieve an additional precondition in the **S-rep** of the operator or a conjectured negated precondition, or it generates a plan that does not use the failed operator.

Note that it is possible that more than one true precondition of a failed operator is not satisfied in the state when execution failure occurs. Since OBSERVER needs to generate near miss negative examples and that many preconditions in the **S-rep** may be unnecessary preconditions, the plan repair algorithm generates a plan segment to achieve only *one* unmet precondition at a time. Therefore, it is often necessary to repair plan for a failed operator several times before this operator can be executed successfully in the environment.

4.3.2 Integrating planning, execution and plan repair

OBSERVER's algorithm for integrating planning, execution, and learning is presented in Figure 4.2. Given a problem to solve, OBSERVER first generates an initial plan (step 1) using the learned operators. The **G-rep** of each operator in the initial plan is achieved by the planner, but some preconditions in the **S-rep** may not be achieved. The *current-state* is initialized to the *initial-state* of the practice problem. Whenever the top-level goals of the problem are satisfied in the current state, OBSERVER returns SUCCESS in solving the problem. Each operator in the initial plan is executed in the environment, if the preceding operator is executed successfully and if the goal this operator achieves is not already satisfied in the state. The *current-state* is updated according to each execution. When an operator is executed in the environment, there are two possible outcomes, which generates either positive or negative training examples for refining operators:

1. The state does not change after executing *op*. In this case, we say that *op* is executed *unsuccessfully*.

   An operator executes unsuccessfully because OBSERVER achieves the preconditions in the **G-rep** of each operator during planning without necessarily achieving all the preconditions in the **S-rep**. This introduces the possibility of incomplete or incorrect plans in the sense that a real precondition is unsatisfied. Unsuccessful executions form the negative examples that OBSERVER uses for refining operators as discussed in Chapter 5. Upon each unsuccessful execution, OBSERVER updates the **G-rep** by learning necessary preconditions of the operator (step 10). OBSERVER also attempts to generate a *repaired-plan* (step 11). If such a plan is found, OBSERVER continues execution using the repaired plan; otherwise, OBSERVER removes the failed operator and continues execution with the remaining plan (steps 12-15).

2. The state changes after executing *op*. In this case, we say that *op* is executed *successfully*.

   Successful executions form the positive examples that OBSERVER uses for refining operator as described in Chapter 5. Note that a successful execution may still have

procedure: practice_and_learn

Inputs: *top-goals*, goals of the problem to solve
initial-state, initial state of the problem,
Output: SUCCESS or FAILURE in solving the problem

```
1.  plan ← Plan(top-goals, initial-state)
2.  current-state ← initial-state
    Repeat
3.        if top-goals are satisfied in current-state then return(SUCCESS)
4.        if plan = NULL then plan ← Plan(top-goals, current-state)
5.        if plan = FAIL
6         then return(FAILURE)
          else
7.            op ← First_step_of(plan)
8.            new-state ← Execute_in_environment(op)
9.            if (new-state = current-state)
              then                   ;; unsuccessful execution, refine operator, repair plan
10.                Update the G-rep
11.                repaired-plan ← plan_repair_for_failed_op
                                   (op, top-goals, current-state)
12.                if (repaired-plan = FAIL)
13.                then plan ← plan \ (op)
14.                else  plan ← repaired-plan
15.                goto 3
16.           if (new-state ≠ current-state)
              then                                  ;; successful execution, refine operator
17.                Update the S-rep and operator effects
18.                current-state ← new-state
19.                plan ← plan \ (op)
20.                goto 3
```

Figure 4.2: OBSERVER's algorithm for integrating planning, execution, and learning. When solving a problem, OBSERVER starts execution when the planner has achieved all the preconditions in the **G-rep** of the operators in the plan. OBSERVER refines operators based on executions. Plan repair is invoked upon each execution failure.

incorrect predictions of how the state changes due to incomplete operator effects. After each successful execution, OBSERVER generalizes the **S-rep** by removing the preconditions that are not satisfied in *current-state*, and generalizes the effects of *op* if missing effects are observed in the *delta-state* of the execution (step 17). OBSERVER then updates the current state and continues execution with the remaining plan (steps 18-20).

Note that our notion of success or failure of execution is based on the correctness of the operator preconditions, rather than on the correctness of the predictions by operator effects.

4.4 Notes on Implementation

This section describes some implementation details in OBSERVER.

4.4.1 Resource bounds

Interleaving planning and execution is undecidable because even classical planning is undecidable. One common technique used for search in the space of an undecidable problem is to use resource bound. The following resource bounds are used in OBSERVER during practice: [1]

Maximum plan-repair depth

A parameter `*max-plan-repair*` is used as a limit for the maximum number of recursion in repair one failed operator.

Maximum number of executions allowed to solve a problem

Another resource bound is to limit the maximum number of trials of operator executions in the environment. In OBSERVER, two parameters `*max-num-of-unsuc-exe*` and `*max-num-of-suc-exe*` are used to upper-bound the total number of unsuccessful and successful executions that OBSERVER tries before it stops working on the problem.

[1] We have experimented with different heuristics (such as goal loop detection, heuristics for choosing which operator to apply next) for planning and plan repair during practice. None of the heuristics improves OBSERVER's performance.

Maximum number of plan repairs allowed for each operator

When an operator fails to execute in the environment, OBSERVER repairs the plan by achieving one condition in the **S-rep** of the operator preconditions at a time and re-executes the failed operator when the condition in the **S-rep** is achieved. When the operator fails in the first place, there may be several necessary preconditions of this operator that are not yet learned as conditions in the **G-rep**. Therefore, it may take a number of plan repairs before this operator becomes applicable, since the plan repair mechanism achieves only one condition in the **S-rep** at a time. But if this failed operator contains true preconditions that are not achievable under the current circumstance, OBSERVER may repair this operator endlessly as it may not know that some true preconditions are not achievable. To prevent this from happening, a parameter `*max-num-plan-repair-per-op*` is used to limit the number of plan repairs for each failed operator.

4.4.2 Binding generation

OBSERVER's underlying planner, PRODIGY4.0, generates bindings for every variable in an operator. Since OBSERVER uses the **G-rep** for planning, it generates bindings for every variable in the **G-rep**. The variables in the **S-rep**, but not in the **G-rep**, may be extraneous variables, because the **S-rep** often contains extraneous preconditions. To learn whether such variables are extraneous, OBSERVER may execute the operator even though there are no objects of the right type for these variables. If the operator executes successfully under this circumstance, the preconditions using these variables are learned as extraneous preconditions, and these variables are learned as extraneous variables.

4.5 Discussion

OBSERVER implements a framework for integrating planning, execution, and learning that incrementally refines imperfect operators from practice. This section describes the scope of this framework.

4.5.1 Applicability to other operator-based planners

Although OBSERVER's framework for integrating planning, execution, and learning is implemented in the context of the PRODIGY4.0 planner, we argue that this framework can be applied to most operator-based planners for the following reasons:

- Most operator-based planners use a similar representation of operators. These operators all have preconditions, and effects that have the same semantics (i.e., preconditions of an operator denote the conditions under which the operator can be applied successfully, the effects of the operator denote changes to the state of the world when the operator is executed). The preconditions and the effects are exactly what OBSERVER is learning.

- The planning and plan repair strategy used in OBSERVER is solely dependent upon the representation of the learned operators. OBSERVER does not depend on PRODIGY4.0's search principles of combining backward chaining and forward projection of the states. OBSERVER does not modify PRODIGY4.0's search algorithm either. Plan-space operator-based planners can generate plans for practice problems using the **G-rep**, and can repair a plan using the **S-rep** of the operator preconditions, in the same way as PRODIGY4.0.

4.5.2 Strategies for interleaving planning and execution

When interleaving planning and execution, there are many strategies for determining when to plan and when to execute. We discuss what the advantages and disadvantages are of several strategies.

1. Achieve all the preconditions in the **G-rep** of the operator preconditions before execution starts. This is OBSERVER's strategy, which has the following advantages:

   - OBSERVER does not need to achieve all the preconditions in the **S-rep**, thus avoiding unnecessary search and facilitating the rapid generation of an initial plan to achieve the top-level goals
   - The system generates useful learning opportunities: if an operator executes successfully when some preconditions in the **S-rep** are not satisfied in the state, then the unsatisfied preconditions are learned to be extraneous preconditions and are thus removed from the **S-rep**, resulting in more complete operators

   One potential problem of this approach is that initially, when the **G-rep** of operator preconditions is overly-general, it may take a long time for the system to solve a problem; however, empirical results, as shown in Chapter 6, demonstrate that this strategy is effective for generating learning opportunities during practice and for solving problems using incomplete and incorrect operators.

2. Achieve all the preconditions in the **S-rep** of the operator preconditions before execution. The advantage of this approach is that the plans thus generated are correct in the sense that they do not miss operators; however, such plans often contain unnecessary extra operators. The disadvantage is that it may take a long time to generate a plan when the **S-rep** is overly-specific and that training examples for removing extraneous preconditions from the **S-rep** cannot be generated.

3. Start execution as soon as the first operator with satisfied **G-rep** is found by the planner to achieve a top-level goal or a subgoal. This approach has the advantage that it can quickly generate a plan step to execute. On the other hand, this strategy does not use the knowledge the planner already has to plan ahead before execution starts. Thus it may lead to executions of operators that are not needed for solving the given problem. This is especially dangerous if many steps are irreversible.

4. Execute the plan when all the preconditions in the **G-rep** are achieved as well as a subset of the preconditions in the **S-rep** are achieved. This can be a very good strategy, if there are good heuristics that choose which subset of the preconditions in the **S-rep** are more likely to be the true operator preconditions; however, there is no obviously good domain-independent heuristic. Furthermore, to generate near miss negative examples for specializing the **G-rep**, it is necessary to generate a negative example where a true operator precondition is not satisfied in the pre-state of the execution.

4.5.3 Other plan repair strategies

There are also several methods of plan repair that can be considered by OBSERVER. In this section, we discuss the following three types of plan repair that are relevant to OBSERVER: safe repair, unsafe repair, and inadmissible repair.

- A *safe repair* method ensures that the postconditions of the plan segment generated during plan repair do not undo the preconditions that are needed for executions of the rest of the initial plan

- An *unsafe repair* method may generate a plan segment whose postconditions undo some preconditions that are needed for the execution of the rest of the initial plan; however, the undone states can be re-achieved by other operators in the domain (e.g., domains where every action is reversible)

- An *inadmissible repair* method may generate a plan segment whose postconditions undo some preconditions that are needed for executions of the initial plan, and

that the undone states cannot be re-achieved by any operators in the domain (e.g., domains where some actions are irreversible)

Although a *safe repair* may seem more desirable than an *unsafe repair*, because the rest of the operators in the initial plan can be executed without additional plan repair, it is not so for the following reasons:

- The preconditions of an operator that are violated by earlier plan repair can be easily achieved immediately before OBSERVER executes this operator
- It takes much longer to repair a plan upon each execution failure using a *safe repair* method, because the planner must consider the preconditions of the rest of the initial plan as goals during plan repair
- A precondition of a later operator may be achieved and deleted several times before this operator is executed, especially when the initial plan misses many steps and much plan repair is required. Achieving such a precondition as soon as it is violated may be unnecessary

An inadmissible repair is in general not desirable as it causes an operator in the rest of the initial plan to be invalid, because its preconditions are violated and unachievable. OBSERVER may nonetheless generate inadmissible plan repairs because it may not know all the effects of the operators. This is not a serious problem, because OBSERVER is able to choose a different operator instead of using the original operator, when the original operator fails to execute. Furthermore, most planning domains are *"almost-reversible"* in that most literals can be added and deleted by some operators.

4.6 Summary

This chapter described how OBSERVER practices for operator refinement. First, we discussed how planning operators can be imperfect due to *overly-general preconditions, overly-specific preconditions, incomplete effects, extraneous effects,* and *missing operators*, and how each type of imperfection complicates classical planning.

OBSERVER uses the **G-rep** of operator preconditions for initial planning. Since the **G-rep** may miss some true preconditions of the operator, operators in the initial plan may fail to execute in the environment, necessitating plan repair. During plan repair, OBSERVER uses the **S-rep** of the failed operator to determine which additional preconditions to achieve to make the failed operator applicable.

We argue that OBSERVER algorithm is applicable to operator-based planners in general. Some issues related to different strategies for plan repair and for integrating planning, learning, and execution were discussed.

Chapter 5

Refining Operators by Practice

So far we have described how OBSERVER learns operators by observation, and how OBSERVER plans and repairs plans during practice. This chapter describes our algorithms for refining operators, using both positive and negative examples of operator executions generated during practice. As described in Chapter 4, there are two possible outcomes of operator execution during practice: *(i)* if an operator causes the state to change, then the execution is successful. Such executions are used as positive examples for operator refinement; *(ii)* if the operator does not cause any changes in the state, then the execution is unsuccessful. Such executions are used as negative examples for operator refinement. In Section 5.1, we formalize the input and output for operator refinement. In Section 5.2, we give detailed descriptions of the learning algorithm. In Section 5.3, we analyze the computational complexity of the learning algorithm described and prove that the algorithm runs in polynomial time. We also prove convergence of the learning algorithm in Section 5.4. In Section 5.5, we discuss some issues related to operator refinement in our approach. And finally we summarize this chapter in Section 5.6.

5.1 Input and Output

During practice, OBSERVER executes plans to solve problems, using the initial operators learned by observation. Each execution is a quadruple `<Op, P, D, B>`, where `Op` is the operator being executed, `P` is the pre-state of the execution, `D` is the delta-state of the execution, and `B` is the variable binding of the operator. We say that an execution is successful if it causes the state to change (i.e., the delta-state is nonempty). We say that an execution is unsuccessful if it does not cause any state changes (i.e., the delta-state is the empty set). Since OBSERVER only executes an operator if its effect is not already true

```
(operator hold-with-vise
  (preconds ((<v3> Machine) (<v2> Vise) (<v1> Part) (<v4> Side) (<v7> Side))
   (and (has-device <v3> <v2>)
        (is-available-table <v3> <v2>)
        (is-empty-holding-device <v2> <v3>)
        (on-table <v3> <v1>)
        (is-clean <v1>)
        (hardness-of <v1> soft)
        (is-available-part <v1>)
  (effects (<v6> spot-drill))
   (if (and (size-of <v1> diameter 4.75)
            (size-of <v1> length 5.5)
            (shape-of <v1> cylindrical)
            (hardness-of <v1> soft)
            (material-of <v1> bronze)
            (holding-tool <v3> <v6>))
       (add (holding-weakly <v3> <v2> <v1> <v7>)))
   (if (and (shape-of <v1> rectangular)
            (has-device <v3> <v2>)
            (is-available-table <v3> <v2>)
            (is-empty-holding-device <v2> <v3>)
            (on-table <v3> <v1>)
            (is-clean <v1>)
            (is-available-part <v1>)
       (add (holding <v3> <v2> <v1> <v4>)))
   (del (on-table <v3> <v1>))
   (del (is-available-part <v1>))
   (del (is-empty-holding-device <v2> <v3>)))))
```

extraneous preconditions:

```
(holding-tool <v3> <v5>)
(size-of <v1> width <v6>)
(size-of <v1> height <v7>)
(size-of <v1> length <v8>)
(material-of <v1> <v10>)
(shape-of <v1> <v11>)
```

Figure 5.1: Operator HOLD-WITH-VISE learned from observations of expert solutions. Note that the operator preconditions and the conditional preconditions have extraneous literals. This operator will be further refined during practice.

in the state, an unsuccessful execution occurs only when some true preconditions are not satisfied in the state.

Figure 5.1 is an example of operator HOLD-WITH-VISE that is learned by observation. Note that both the regular and conditional preconditions have extraneous literals. Figure 5.2 is an example of a successful execution. Figure 5.5 is an example of an unsuccessful execution. Note that operator refinement by practice differs from the initial observation stage in that the variable bindings are determined by the planner and, hence, are known to OBSERVER for learning.

op-name: hold-with-vise

bindings: {drill4/<v3>, part4/<v1>, vise4/<v2>, side4/<v7>}

Pre-state:

```
(has-device drill4 vise4)
(on-table drill4 part4)
(is-available-table drill4 vise4)
(is-clean part4)
(is-available-part part4)
(size-of part4 diameter 3.0)
(size-of part4 length 4.5)
(shape-of part4 cylindrical)
(material-of part4 iron)
(hardness-of part4 hard)
(is-empty-holding-device vise4 drill4)
```

Delta-state:

adds:

```
(holding-weakly drill4 vise4 part4 side4)
```

dels:

```
(is-empty-holding-device vise4 drill4)
(on-table drill4 part4)
(is-available-part part4)
```

Figure 5.2: A *successful* execution of the operator HOLD-WITH-VISE shown in Figure 5.1.

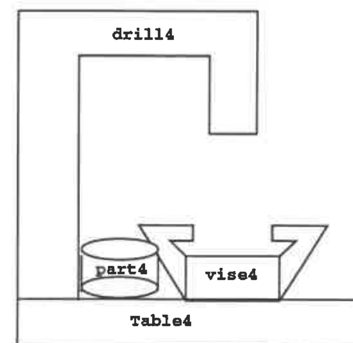


Figure 5.3: Pre-state of execution in Figure 5.2.

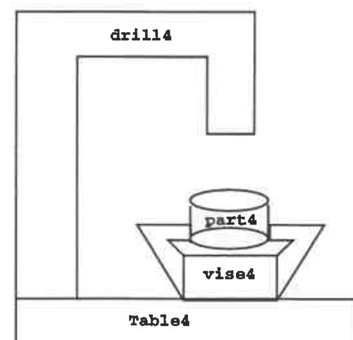


Figure 5.4: Post-state of execution in Figure 5.2.

op-name: `hold-with-vise` (Figure 5.1)

bindings: {drill5/`<v3>`, part5/`<v1>`, vise5/`<v2>`}

Pre-state:

```
(has-device drill5 vise5)
(on-table drill5 part5)
(is-available-table drill5 vise5)
(is-clean part5)
(is-available-part part5)
(size-of part5 diameter 3.0)
(size-of part5 length 5.5)
(material-of part5 bronze)
(hardness-of part5 soft)
(shape-of part5 cylindrical)
(size-of part6 length 4.75)
(size-of part6 width 3.5)
(size-of part6 height 4.5)
(shape-of part6 rectangular)
(holding drill5 vise5 part6 side1)
```

Delta-state: NIL

Figure 5.5: An *unsuccessful* execution of the operator `HOLD-WITH-VISE` shown in Figure 5.1. This operator fails to execute because a true precondition of the operator (i.e., `(is-empty-holding-device <v2> <v3>)`) is not satisfied in the pre-state.

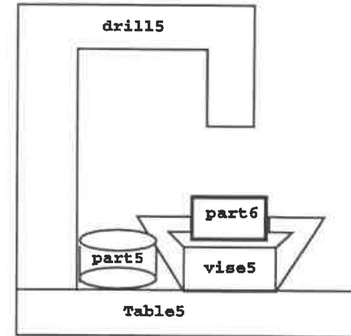


Figure 5.6: Pre-state of the execution in Figure 5.5. Holding devise `vise5` is not empty and, hence, it cannot hold a second part (i.e., part5)

5.2 Refining Operators

This section presents `OBSERVER`'s algorithm for refining operators during practice. The refinement involves: *(i)* updating the **S-rep** of operator preconditions, *(ii)* learning the **G-rep** of the operator preconditions, and *(iii)* updating the effects, including conditional effects and conditional preconditions.

5.2.1 Updating the S-rep of operator preconditions

Updating the **S-rep** of operator preconditions by practice is similar to learning the **S-rep** by observation, except for two differences. The first difference lies in the variable bindings of the operators. When learning by practice, the bindings are known, because

they are uniquely determined by the planner. But when learning by observation, most bindings are unknown, even though partial bindings can be determined by matching the operator effects with the delta-state. As a result, the **S-rep** can be generalized more effectively by practice than by observation. The second difference is that the **S-rep** may also be specialized by practice—negated preconditions are conjectured and added to the **S-rep** (thus specializing the **G-rep**) from unsuccessful executions where all the **S-rep** preconditions are satisfied in the pre-states.

Figure 5.7 describes the procedure **update_S_rep_by_practice** for updating the **S-rep** of operator preconditions by practice. Steps 1-2 concern learning from positive examples (i.e., successful executions). The **S-rep** is generalized by removing all the preconditions whose instantiations, given the variable bindings, are not satisfied in the pre-state. For example, given the initial operator shown in Figure 5.1 and the successful execution shown in Figure 5.2, OBSERVER first instantiates the preconditions in the **S-rep** using the bindings of the execution ({`drill4/<v3>, part4/<v1>, vise4/<v2>, side4/<v7>`}). OBSERVER then checks if each instantiated precondition is in the pre-state of the execution. In this example, the precondition `(hardness-of <v1> soft)` is not satisfied in the pre-state and, hence, is removed from the **S-rep**.

---

procedure: update_S_rep_by_practice

Input: learned operator *op*, execution *exe*
Output: the **S-rep**

1. If *exe* is a successful execution
   then ;; removing extraneous preconditions
2. for each $s \in$ **S-rep**, if $s \notin$ *pre-state(exe)*, then **S-rep** $\leftarrow$ **S-rep** $\setminus\{s\}$
3. If *exe* is an unsuccessful execution
4. and if $\forall p \in$ **S-rep**, $p \in$*pre-state(exe)*
   then ;; learning negated preconditions
5. for each $s \in$ *pre-state(exe)*, if $s \notin$ **S-rep** and $s \notin$ *extraneous_preconds(op)*
6. then **S-rep**$\leftarrow$ **S-rep** $\cup\{(not(s))\}$

Figure 5.7: Update the **S-rep** of operator preconditions by practice. This includes removing extraneous preconds from successful executions and adding conjectured negated preconditions from unsuccessful executions.

Steps 3-6 concern learning from negative examples (i.e., unsuccessful executions). Negated preconditions are learned from unsuccessful executions where all the preconditions in the **S-rep** are satisfied in the pre-state. The potentially negated preconditions are the literals that are true in the pre-state of the unsuccessful execution, but are not specified in

the **S-rep** of the operator preconditions, nor in the extraneous preconditions previously removed from the operator. In other words, a literal *(not p)* is a potential negated precondition if: *(i)* Substitute(p, bindings) is in the pre-state of the unsuccessful execution, *(ii)* p is not in the **S-rep** of the operator preconditions, and *(iii)* p is not an extraneous precondition. `OBSERVER` adds potential negated preconditions to the **S-rep**.[1] Once negated preconditions are added to the **S-rep**, they are subject to refinement (i.e., removal or permanent retainment), and are used for plan repair just as other preconditions in the **S-rep**. Since many potential negated preconditions may be conjectured and added to the **S-rep** from one unsuccessful example, the negated preconditions are removed from the **S-rep** at the end of each practice problem, so as not to increase the size of the **S-rep**.

For example, consider an unsuccessful execution of operator `HOLD-WITH-VISE`, shown in Figure 5.8. `(has-burrs part7)` is the only literal in the pre-state of the execution that is not in the **S-rep**, nor in the extraneous preconditions of the operator. Therefore, `(not (has-burrs <v1>))` is conjectured as a negated precondition and added to the **S-rep**. Note that, in general, more than one negated precondition may be conjectured from one unsuccessful execution.

5.2.2 Learning the G-rep of operator preconditions

The **G-rep** of an operator is initialized to the empty set. It is specialized using negative examples by adding elements from the **S-rep** that `OBSERVER` confirms to be true preconditions of the operator. Traditional methods for updating the G-set of a version space use every negative example; however, there are usually multiple ways to specialize the G-set given one negative example. In fact, Haussler [Haussler, 1988] proved that specializing the G-set is an exponential process and the size of the G-set may grow exponentially with the number of examples. When the number of negative examples is large, the size of the G-set becomes unmanageable.

To avoid problems stemming from the exponential growth of the G-set, `OBSERVER` specializes the **G-rep** only when there is a unique specialization of the **G-rep** given the negative example. We assume that after learning by observation, the **S-rep** is general enough such that *(i)* all the constants are generalized to variables, if they are to be in the true preconditions of the operators, and *(ii)* the types of the variables in the operators are correctly learned. Thus, the **G-rep** can be uniquely specified given a negative example where exactly one precondition in the **S-rep** is not satisfied in the pre-state. Such

[1] In our implementation, negated preconditions may also be conjectured when there are unsatisfied preconditions in the **S-rep**. This occurs when learning from an unsuccessful execution for which `OBSERVER` has exhausted resource bound in plan repair. This provides `OBSERVER` with more opportunities for learning negated preconditions.

op-name: `hold-with-vise` (Figure 5.1)

bindings: {drill7/<v3>, part7/<v1>, vise7/<v2>}

Pre-state:

```
(has-device drill7 vise7)
(on-table drill7 part7)
(is-clean part7)
(is-empty-holding-device vise7 drill7)
(is-available-table drill7 vise7)
(is-available-part part7)
(hardness-of part7 soft)
(material-of part7 brass)
(size-of part7 width 2.5)
(size-of part7 height 4.0)
(size-of part7 length 5.75)
(shape-of part7 rectangular)
(has-burrs part7)
```

Delta-state: NIL

Figure 5.8: An execution of the operator `HOLD-WITH-VISE`. This execution is unsuccessful, since the application of this operator does not lead to any state changes (delta-state is the empty set). `(has-burrs part7)` is a literal in the pre-state that is not in the **S-rep**, nor in the extraneous preconditions of the operator, and is learned as a negated precondition.

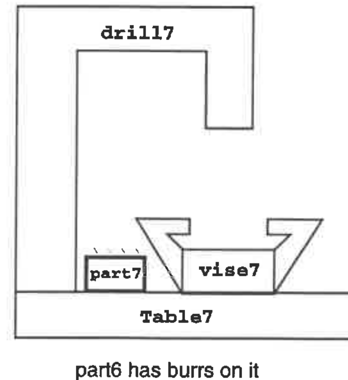


Figure 5.9: Pre-state of execution in Figure 5.8.

negative examples are called *near misses*. When there are multiple ways to specialize the **G-rep**, `OBSERVER` does not specialize the **G-rep**, but rather, stores the unsuccessful executions to learn from them later, when the **S-rep** is further generalized.

Figure 5.10 shows `OBSERVER`'s algorithm for learning the **G-rep**. In step 1, the **G-rep** is initialized to the empty set. `OBSERVER` incrementally learns from a training instance (either a successful or unsuccessful execution) by practice (steps 3-10). If the execution is unsuccessful, `OBSERVER` checks whether it is a near miss. If not, `OBSERVER` stores this failure execution in the list of *failed-states* to use it if it becomes a near miss when the **S-rep** is further generalized. If the unsuccessful execution is a near miss, (i.e., there is exactly one precondition, say p, in the **S-rep** not satisfied in the pre-state), then p is learned to be a true precondition of the operator and the **G-rep** is specialized by

procedure: learn_G_rep_by_practice

Given: a list of executions$\{exe\}$
Learn: the **G-rep**

```
1.  G-rep = ∅
2.  failed-states ← ∅
3.  for every execution exe
4.      if exe is an unsuccessful execution
        then
5.          if exe is a near miss
            i.e., ∃ p ∈ S-rep s.t. p is the only S-rep precondition not satisfied in pre-state
6.          then
7.              if p is a negated precondition
                then                ;; reset the G-rep when adding a negated precondition
8.                  G-rep = ∅
9.              G-rep = G-rep ∪ {p}
10.         else failed-states ← failed-states ∪ {exe}
11.     if exe is a successful execution
        then
12.         S-rep = update_S_rep_by_practice(exe)
13.         for each fail ∈ failed-states
14.             if fail is a near miss
15.             then repeat steps 5-9
16.                 failed-states ← failed-states \ {fail}
```

Figure 5.10: Learning the **G-rep** of operator preconditions incrementally by practice. The **G-rep** is updated only when the negative example is a *near miss*, i.e., there is a unique specialization of the **G-rep** based on the unsuccessful execution. The negative examples that are not near misses are stored and considered later, when further **S-rep** generalizations render them near misses.

adding p. Note that p may be incorrectly learned as a true precondition, if the reason for the unsuccessful execution is a missing negated precondition. This potential error is corrected later, when a negated precondition is added to the **G-rep**, by removing all the non-negated literals from the **G-rep**. Should a non-negated literal thus removed be a true operator precondition, it will be added back to the **G-rep** with future near misses.[2]

[2]Another method for correcting this potential error is to save all the previous near misses. When a negated precondition is added to the **G-rep** (and therefore to the **S-rep**), previous near misses are re-verified, so that the incorrectly learned true preconditions are removed from the **G-rep** while other preconditions are retained. Since negated preconditions are rare, both methods are efficient.

Steps 11-16 describe the case when the input is a successful execution. After generalizing the **S-rep**, OBSERVER re-examines previous unsuccessful executions in *failed-states* for near misses. The **G-rep** is specialized accordingly, if any previous unsuccessful executions become near misses due to the generalization of the **S-rep**.

For example, consider the operator shown in Figure 5.1 and an unsuccessful execution shown in Figure 5.5. OBSERVER notices that there is only one unmet precondition in the **S-rep**, (is-empty-holding-device <v2> <v3>). This precondition is thus learned as a true precondition and the **G-rep** is specialize by adding (is-empty-holding-device <v2> <v3>); however, if a second precondition (e.g., (is-clean part5)) in the **S-rep** is not satisfied in the pre-state, then the execution is not a near miss and OBSERVER does not specialize the **G-rep**.

5.2.3 Refining operator effects

Figure 5.11 summarizes the algorithm for refining operator effects (including conditional effects and preconditions) by practice. This procedure is very similar to the procedure **update_effects_by_observation** for learning by observation, as described in Figure 3.12. The only difference is that the variable bindings of the operators are determined by the planner during execution, whereas but only partial bindings can be learned through matching when learning by observation.

For example, consider the operator shown in Figure 5.1 and its successful execution shown in Figure 5.2. The conditional effect (add (holding-weakly <v3> <v2> <v1> <v7>)) appears in the delta-state. Therefore, the **S-rep** of the conditional preconditions for this conditional effect is generalized by removing those not satisfied in the pre-state. The conditional effect (add (holding <v3> <v2> <v1> <v4>)) does not appear in the delta-state and, thus, is not updated. The other parts of the effects are not updated either, since they correctly predict the outcome of this execution. The resulting operator, given the successful execution shown in Figure 5.2 and unsuccessful executions shown in Figure 5.5 and 5.5, is shown in Figure 5.12.

procedure: update_effects_by_practice

Given: successful execution *exe*
Update: the effects of the operator

1. *effects* ← *Effects*(*op*)
2. *deltas* ← *delta-state*(*exe*)
3. for every $e \in$ **Effects**(*op*)
4.   if $e \in$ *delta-state(exe)*
     then
5.   if *e* is a conditional effect
6.   then **update_conditional_preconditions**(*e*, *exe*)
7.   *effects* ← *effects* $\setminus \{e\}$
8.   *deltas* ← *deltas* $\setminus \{d\}$
9. for every $e \in$ *effects*
10.   if *e* is not a conditional effect
     then ;;; *e* not in the delta-state, learn a conditional effect
11.   **conditional_effects**(*op*) ← **conditional_effects**(*op*) ∪ *e*
12.   **S-rep**(e) ← **S-rep**(op)
13. for every $d \in$ *deltas* ;;; *d* not predicted by the operator, learn a conditional effect
14.   *new-effect* ← **parameterize**(*d*)
15.   **conditional_effects**(*op*) ← **conditional_effects**(*op*) ∪ *new-effect*
16.   S(*new-effect*) ← **parameterize**(*pre-state*(*exe*))

Figure 5.11: Updating the operator effects by practice. This is similar to updating the operator effects by observation, as described in Figure 3.12, except that variable bindings are already known and need not be learned.

5.3 Complexity Analysis

This section analyzes the time complexity of refining an operator by practice.

Let:

n = maximum number of literals in a state (either pre-state or post-state)
m = maximum number of arguments in a literal
l = maximum number of conditional effects in an operator
k_1 = total number of successful executions (positive examples)

```
(operator hold-with-vise
  (preconds ((<v3> Machine) (<v2> Vise) (<v1> Part) (<v4> Side) (<v7> Side))
   (and (has-device <v3> <v2>)
        (is-available-table <v3> <v2>)
        (is-empty-holding-device <v2> <v3>)
        (on-table <v3> <v1>)
        (is-clean <v1>)
        (is-available-part <v1>)
        (not (has-burrs <v1>))))
  (effects
   (if (shape-of <v1> cylindrical)
       (add (holding-weakly <v3> <v2> <v1> <v7>)))
   (if (and (shape-of <v1> rectangular)
            (has-device <v3> <v2>)
            (is-available-table <v3> <v2>)
            (is-empty-holding-device <v2> <v3>)
            (on-table <v3> <v1>)
            (is-clean <v1>)
            (is-available-part <v1>)
       (add (holding <v3> <v2> <v1> <v4>))))
   (del (on-table <v3> <v1>))
   (del (is-available-part <v1>))
   (del (is-empty-holding-device <v2> <v3>))))))
```

G-rep: `(is-empty-holding-device <v2> <v3>)`

extraneous preconditions:

```
(holding-tool <v3> <v5>)
(size-of <v1> width <v6>)
(size-of <v1> height <v7>)
(size-of <v1> length <v8>)
(hardness-of <v1> soft)
(material-of <v1> <v10>)
(shape-of <v1> <v11>)
```

Figure 5.12: Refined operator `HOLD-WITH-VISE`, given the successful execution shown in Figure 5.2, and given the unsuccessful executions shown in Figure 5.2 and 5.5. Note that: *(i)* negated precondition `(not (has-burrs <v1>))` is added to the **S-rep** (using negative example), *(ii)* the previous precondition `(hardness-of <v1> soft)` is removed from the **S-rep** (using positive example), *(iii)* the conditional precondition for the effect `(add (holding-weakly <v3> <v2> <v1> <v7>))` is generalized, and *(iv)* `(is-empty-holding-device <v2> <v3>)` is added to the **G-rep**.

Updating the S-rep

Updating the **S-rep** has two parts: removing extraneous preconditions using positive examples and learning negated preconditions using negative examples.

Given a positive example (i.e., successful execution), **S-rep** is updated by removing every literal that is not satisfied in the pre-state of the execution. Thus, for every literal l in the **S-rep** (a total of $O(n)$ literals), the algorithm substitutes the variables in l using the variable bindings (the substitution has time complexity $O(m)$ for each literal), and checks if the substitution is in the pre-state (this can be done in constant time using a hash table to store the state, or in time $O(mn)$ when searching linearly.). Thus the time complexity for updating the **S-rep** is $O(mn)$ (or $O(m^2n^2)$ when not using a hash-table).

Given a negative example (i.e., unsuccessful execution), **S-rep** is updated by adding potential negated preconditions. Each literal in the pre-state (a total of $O(n)$ literals) of the unsuccessful execution is generalized using the variable bindings (this has time complexity $O(m)$ for each literal). Each generalization is tested for membership in the **S-rep** and the extraneous preconditions of the operator (the membership test can be done in constant time when using a hash-table to store the **S-rep**, or in time $O(mn)$ otherwise.) Thus, the complexity of learning negated preconditions from one negative example is $O(mn)$ (or $O(m^2n^2)$ when not using a hash-table).

Updating the G-rep

Learning the **G-rep** is *"semi-incremental,"* because the negative examples that were not near misses are re-examined whenever the **S-rep** is generalized. Thus, in the worst case, a negative example may be examined k_1 times after every generalization of the **S-rep** using a successful execution.

Given one negative example, OBSERVER identifies which literals in the **S-rep** are not satisfied in the pre-state of the execution. The time complexity is the same with updating the **S-rep**, i.e., $O(mn)$ (or $O(m^2n^2)$ when not using a hash-table).

Updating the effects

The number of conditional effects that can be learned from one successful execution is $O(l)$. Updating each **S-rep** and **G-rep** of the conditional preconditions takes $O(mn)$ (or $O(m^2n^2)$ when not using a hash-table) time. Therefore, the time complexity for updating the effects is $O(lm^2n^2)$ for each example.

5.4 Convergence Proof

Theorem: If the operator does not have negated preconditions and the **S-rep** *learned by observation is general enough such that: (i) all the constants are generalized to variables if they are in the true preconditions of the operators, and (ii) the types of the variables in the operators are correctly learned, then if the S-set and G-set in the version-spaces approach converge using all the negative examples, then using only the near miss negative examples suffices to converge.*

Let:
`P` = {e | e is a positive example from practice}
`F` = {e | e is a far miss negative example}
`M` = {e | e is a near miss negative example}
`N` = {e | e is a negative example} = F ∪ M
S-rep = $\{l_1, l_2, ..., l_n\}$

We need to prove that, if the set of examples (P ∪ F ∪ M) is sufficient for convergence using the version spaces algorithm by learning S-set and G-set (i.e., S-set = G-set = C), then the P ∪ M (i.e., using only the positive examples and near miss negative examples) is also sufficient for convergence of the **S-rep** and **G-rep**.

We assume that *(i)* all the constants are generalized to variables if they are in the true preconditions of the operators, and that *(ii)* the types of the variables in the operators are correctly learned. Thus, the G-set should be a conjunct of a subset of the literals in the **S-rep**. We now prove that if the negative examples (F ∪ M) are sufficient for converging the G-set, then the **G-rep** learned using only the near misses (M) will also converge.

Suppose the G-set converges to $C = \{l_{i_k} | 1 \leq i_k \leq n\} \subset$ **S-rep** using the candidate elimination [Mitchell, 1978] algorithm. To prove that learning the **G-rep** using `OBSERVER`'s algorithm will also cause the **G-rep** to converge to C, it suffices to show that, for every $l_j \in C$, there exists a near miss negative example `m` ∈ M, such that l_j is the only element in `m` that is not satisfied in the pre-state of `m`. Suppose this does not hold. Then $\exists\ l_a \in$ C such that there is no near miss where $l_a \in$ **S-rep** is the only unsatisfied precondition. This implies that for every negative example $e \in$ `N` , there is at least one other element $l_b \in C$, $l_b \neq l_a$, such that l_b is not satisfied in the pre-state of the negative example e. This is because in every negative example, there is at least one element in the **S-rep** that is not satisfied in the pre-state (by definition of unsuccessful execution) and that l_a cannot be the only literal in e not satisfied in the pre-state (by our assumption of l_b). Then, let $C' = C \setminus \{l_a\}$. We see that:

- C′ is more general than C
- C′ is a general boundary of the operator preconditions, since none of the negative examples is more specific than C′. This is because in every negative example, at least one element $l_b \neq l_a$ is not satisfied in C′

This is a contradiction with the fact that G-set is the most general boundary of the concept given all the positive and negative examples.

Q.E.D.

This theorem has two premises. First, the **S-rep** learned from observations must be general enough so that *(i)* all the constants are generalized to variables if they are in the true preconditions of the operators, and *(ii)* the types of the variables in the operators are correctly learned. These premises are important, because OBSERVER's algorithm for refining operators by practice cannot generalize constants to variables or generalize the types of the variables. Our algorithm can be extended if we allow OBSERVER to experiment more during practice; however, this requires more systematic experimentation methods that are not within the scope of this thesis. Nevertheless, as demonstrated by the empirical results in Chapter 6, OBSERVER is able to generalize the **S-rep** enough so that the overall operator learning is effective.

Second, the proof of the theorem assumes that there is no negated preconditions. Since negated preconditions are rare in most application domains, this theorem covers the majority of the situations for learning operator preconditions.

5.5 Discussions

This section discusses issues related to refining operators by practice.

5.5.1 Trade-off between learning efficiency and learning rate

OBSERVER learns a specific boundary **S-rep** of the operator preconditions incrementally in a way that does require finding the most specific common generalization of the **S-rep** and the pre-state. The **S-rep** contains concepts that are more specific than would be included in the S-set, but can be updated in polynomial time, but it is updated in polynomial time. The **G-rep** is only specialized using near miss negative examples. The **G-rep** contains concepts that are more general than would be included in the S-set, but it can be updated in polynomial time. OBSERVER's approach has the following advantages over the strict version-spaces approach, which learns the most specific and most general boundaries:

- It is much faster to compute the **S-rep** and **G-rep** (polynomial vs. exponential time)
- Both the **S-rep** and **G-rep** are *single* generalization, whereas there are usually many hypotheses in the S-set and G-set. Keeping a single generalization per boundary facilitates planning as it eliminates a choice point for planning, i.e., the choice of which operator hypothesis to use to achieve a goal during planning

5.5.2 Negated preconditions

The planning operators OBSERVER learns may have negated preconditions, whereas the S-set of a version space is represented with existential conjunctive concepts that do not have negations. We notice that negated preconditions are rare in most application domains. For example, in our process planning domain, there are only twenty-five negated preconditions among three-hundred and fifty preconditions in the human expert-coded operators. In many other domains implemented in PRODIGY and other classical planners, such as the extended-strips domain, there are no negated preconditions at all. Thus, OBSERVER first assumes there are no negated preconditions and corrects this simplifying assumption by learning negated preconditions when they are detected during practice.

5.6 Summary

This chapter presented the learning method for refining operators by practice. The input for learning are execution traces obtained during practice where executions = {*exe: (pre-state(exe), delta-state(exe), op, bindings(op))*}. If *delta-state(exe)* is not empty, then this execution is successful; otherwise, it is unsuccessful.

During operator refinement by practice, the **S-rep** is generalized by removing the literals not satisfied in the pre-states of the successful execution. **S-rep** can also be specialized by learning negated preconditions from an unsuccessful execution where all the preconditions in the **S-rep** are satisfied.

The **G-rep** is specialized using *near miss* negative examples where all but one literal in the **S-rep** are satisfied in the pre-state. The **G-rep** can also be generalized—when OBSERVER adds a negated precondition to the **G-rep**, it removes all the non-negated literals that may be previously learned incorrectly due to OBSERVER's initial assumption that there are no negated preconditions.

Effects, including conditional effects and conditional preconditions, are refined using positive examples.

Chapter 6

Empirical Results

The previous chapters described our methods for learning planning operators from observations of expert solutions and for refining operators during practice in a learning-by-doing fashion. The learning algorithms described in this thesis have been fully implemented and tested in the context of PRODIGY4.0 [Carbonell *et al.*, 1992]. In this chapter, we present empirical results that demonstrate the effectiveness of our learning system, including the significance of and the synergism between learning by observation and learning by practice. All the experiments are conducted in two domains: the process planning domain [Gil, 1991, Gil and Pérez, 1994] and a DSN antenna operations domain [Hill *et al.*, 1995, Chien *et al.*, 1996b].

We begin by introducing the two application domains used in our experiments (Section 6.1). We describe the design of our experiments (Section 6.2). The empirical evaluation consists of three parts. First, we evaluate the overall effectiveness of the learning system (Section 6.3). This includes a comparison of problem solving with learned operators and human expert-coded operators. We show that problems can be solved as effectively using learned operators as using expert-coded operators according to several criteria. Second, we evaluate the role of observation (Section 6.4). We show that the total number of solved test problems increases with the number of observation problems the system is initially given, and thus, learning by observation is a crucial component of the learning system. Finally, we evaluate the role of practice (Section 6.5). We show that the operators learned by observation and practice are more effective in problem solving than the operators learned by observation only, using the same training problems.

6.1 Application Domains

The methods described in this thesis were tested and evaluated in two domains: a process planning domain and an antenna operation domain for the Deep Space Network (DSN). These two domains were chosen because they are real world planning problems.

6.1.1 A process planning domain

Process planning is one of the crucial intermediate steps of production manufacturing [Doyle, 1985, Nau, 1987, Hayes, 1990]. The first stage in preparation for manufacturing is engineering, which entails building a model that satisfies a set of specifications, selecting the proper materials, ascertaining the proportions and desired physical properties, and configuring parts into larger assemblies. In the next step process plans are delineated. This includes listing the steps or operations and the dependencies among them, i.e., the process plans, and designating the machines, equipment, and tools needed and performance expected. A process plan, for instance, may require cutting metal stock, machining it into a desired shape, drilling holes for bolt-assembly, and polishing its surface. On the basis of the process plans, operation routines are planned in detail. This phase is called production planning. The last phase is one of scheduling multiple process plans on available machines and allocating time, resources, and human operators. The parts are then manufactured according to the production plans and the master schedule.

Of all the processes involved in process planning, we have concentrated on the machining, joining, and finishing operations. Machining refers to the art of creating parts, usually metal, by carving raw material with power tools such as bandsaws, lathes, milling machines, and drill presses, and using processes such as drilling, milling, and turning [Hayes, 1990]. Joining and assembly processes include soldering, welding and bolting. Finishing processes change the surface properties of a part, including cleaning it or removing burrs [Gil, 1991, Gil and Pérez, 1994]. In our implemented model, there are seventy-three distinct action schemas, where the number of preconditions ranges from one to six with an average of five, and the number of effects ranges from one to five with an average of three.

6.1.2 DSN antenna operation domain

The Deep Space Network (DSN) is a set of world-wide antenna networks which is maintained by the Jet Propulsion Laboratory (JPL). Through these antennae, JPL is responsible for providing the communications link for a multitude of spacecraft. Operations

personnel are responsible for creating and maintaining this link by configuring the required antenna subsystems and performing test and calibration procedures. The task of creating the communications link is a manual and time-consuming process which requires operator input of over a hundred control directives and the constant monitoring of several dozen displays to determine the exact execution status of the system. Recently, a system called the Link Monitor and Control Operator Assistant (LMCOA), has been developed to improve operations efficiency and reduce pre-calibration time. The LMCOA provides semi-automated monitor and control functions to support operating DSN antennae. One of the main inputs to the LMCOA is a Temporal Dependency Network (TDN). A TDN is a directed graph that incorporates temporal and behavioral knowledge. This graph represents the steps required to perform a communications link operation. In current operations, these TDNs are developed manually. DPLAN [Hill *et al.*, 1995, Chien *et al.*, 1996b, Estlin *et al.*, 1996] has been designed to automatically generate these TDNs based on input information describing the antenna track type and the necessary equipment configuration.

DPLAN has access to several information sources to determine track operations. First, it has access to information on spacecraft activities in the form of a Project Sequence of Events (SOE). The SOE specifies the spacecraft activities, such as downlinks and bit-rate changes. Second, the planner has access to a Project Profile which specifies information on how to interpret the Project SOE, such as default frequencies. Third, the planner is given an equipment configuration which details the exact pieces and types of equipment assigned to the track. Fourth, the planner has access to the Temporal Dependency Network (TDN) Knowledge Base which specifies all of the actions available to the planner and their preconditions and postconditions. Finally, DPLAN has access to the track request specification, which corresponds to the goal specification.

The actions in this domain include: moving an antenna to point, configuring subsystems (e.g., antenna controller, receiver, exciter, transmitter, microwave controller, telemetry string), calibrating subsystems, and turning test translators on or off. The resulting output plan configures and calibrates the antenna and its subsystems properly in order to perform required services such as telemetry, ranging, or commanding. In our implemented model, there are twenty-four distinct action schemas, where the number of preconditions ranges from one to seven with an average of three, and the number of effects ranges from one to three with an average of two.

The implemented portion of the DSN domain covers nominal operations for the 34-meter Beam Wave Guide antenna, and as such covers six of twenty-three antenna subsystems in the full DSN domain and three of seven service request types (goals). The operator-based representation of the DSN domain does not capture certain aspects of plan quality (such as the desire to constrain nominal operations [Chien *et al.*, 1996a]). Nevertheless, the implemented portion of the DSN domain is representative of the precondition and effect

relationships of the DSN domain as a whole and thus represents a real-world test for the OBSERVER operator learning system.

6.2 Design of Experimentation

This section describes the design of our experiments, including the different phases involved for learning and testing.

6.2.1 Phases for learning and testing

Our experiments include the following three phases:

1. *Learning operators by observation.* In this phase, OBSERVER is given the expert solution traces for each problem in the *observation training set.* OBSERVER learns an initial set of operators from observation, using the algorithms described in Chapter 3.

2. *Refining operators during practice.* In this phase, OBSERVER solves problems in the *practice training set*, using planning and plan repair mechanisms described in Chapter 4. It then refines operators based on its own execution traces using learning methods described in Chapter 5.

3. *Testing.* In this phase, we compare the effectiveness of problem solving with the learned operators and human expert-coded operators, according to several criteria. In our experiments, we assume that the expert-coded operators are correct. These correct operators are used to model the environment in the simulator.

6.2.2 Randomly generated problems

We have built a random problem generator for each testing domain to generate problems for observation, practice, and testing.

The random generator for the process planning domain is based on that used in [Pérez, 1995]. The problems generated have between one to three top-level goals. In our experiments, we have concentrated on goals of cutting parts to desired sizes according to their three dimensions, and on drilling holes of several different types (counterbored, countersink, tapped, and reamed) in any of the six part sides, because these are the most basic problems in this domain. The randomly generated problem initial state specify the

manufacturing environment (i.e., what and how many machines, tools, holding devices, and so on, are available).

For the DSN antenna operation domain, the problems used have between one to three top-level goals. The type of goals include telemetry, ranging, or commanding, or any combinations of them. The randomly generated problem initial state specify the initial availability of the antennae and their subsystems, that is, what type of subsystems (e.g., antenna controllers, receivers, exciter controllers, command processors, microwave controllers, telemetry strings) are available and how many of each type are available. These problems capture nominal operations for the 34-meter Beam Wave Guide antenna.

6.2.3 Base-level planner

OBSERVER is implemented in the context of PRODIGY4.0 [Carbonell *et al.*, 1992], a nonlinear, operator-based planner (see Appendix A for the overview of the planner). Many different kinds of search heuristics and modes exist for PRODIGY4.0. We use a version of PRODIGY4.0 that chooses to apply as long as there are any active applicable operators (called SAVTA) [Veloso and Stone, 1995]. There are many choice points during plannings: choice points for choosing a goal, for choosing an operator to achieve the goal, and for choosing a set of bindings for an operator, etc. In the absence of control knowledge, PRODIGY4.0's default strategy is to always choose the first alternative. This pre-determined order is arbitrary and introduces a systematic bias in the search time. To remove this bias, we introduced a random selection mechanism at each choice point in our experiments. We ran the planner multiple times to measure the average performance.

In the experiments we report in this chapter, the following resource bounds are used for the planner so that the experiments terminate in reasonable time:

Depth_bound: As search proceeds depth-first in our planner, a depth bound in terms of the number of choice points is used. In our experiments, **depth_bound** = 50.

Max_node_bound: This resource bound is used to limit the maximum number of planning nodes generated, where each node corresponds to a choice point. When planning using incomplete and incorrect operators, this bound limits the sum of the nodes generated during initial planning and the nodes generated during plan repair. In our experiments, during testing, **Max_node_bound = 1500**, while during practice, **Max_node_bound = 5000**.

We varied the above resource bounds in our experiments and the variation does not qualitatively affect the empirical results presented in this chapter.

6.3 Overall Effectiveness of OBSERVER

This section presents empirical results to demonstrate the overall effectiveness of OBSERVER. We first discuss the criteria for evaluating the quality of learned operators. We then present results in the two domains to demonstrate that the OBSERVER-learned operators are as effective in problem solving as human expert-coded operators according to these criteria.

In the process planning domain, the domain knowledge (i.e., a set of operators) was first acquired by an AI expert over several years when developing a special-purpose planner for this domain [Hayes, 1990]. The domain knowledge was then encoded in PRODIGY by another AI expert in six months as a half-time project [Gil, 1991]. In the DSN antenna operations domain, initial knowledge acquisition took ten experts six months each and then was encoded into PRODIGY operators by an AI expert in three months. In both cases, knowledge acquisition required significant direct interaction between AI experts and domain experts.

6.3.1 Criteria for evaluation

To evaluate the overall effectiveness of OBSERVER, we compare the performance of problem solving with OBSERVER-learned operators and human expert-coded operators, according to the following criteria:

1. *The total number of solved test problems.*

   When using expert-coded operators, test problems are either solvable without execution failure or unsolvable within the resource bound, because these operators are correct. When using the learned operators to solve test problems before the end of learning, the initial plan generated may fail to execute when applied in the environment, because these operators may be imperfect. Since OBSERVER has the ability to repair failed plans, the problem can be solved if all the failures can be repaired. Therefore, when using learned operators, a problem could be either solved without plan repair (i.e., without execution failure), or solved with plan repair, or not solved even with plan repair. Thus we measure:

   - Total number of solved test problems without plan repair
   - Total number of solved test problems, including the problems that required plan repair

   We expect that for both measurements, the total number of solved problems should increase with increased training, i.e., with more observation and more practice.

2. *The average number of failures that must be repaired to solve each test problem.*

   We expect that with more observation and practice, the learned operators should become more accurate, and therefore there should be fewer execution failures.

3. *The average number of operator executions required to solve each test problem.*

   The average number of operator executions measures the efficiency of the *solutions generated*, which is one measurement of plan quality. The smaller the number of executions, the better the plan. We expect that the average number of operator executions to solve each test problem should decrease with learning, and that at the end of learning, the average number of operator executions using learned operators should be comparable to that using human expert-coded operators.

   When using OBSERVER-learned operators to solve test problems, the operator executions include both the successful and the unsuccessful executions. Since we assume the human expert-coded operators are correct, the average number of operator executions to solve each test problem using human expert-coded operators is simply the solution length of the plan generated.

4. *The average number of planning nodes required to solve each test problem.*

   When our planner is solving a problem, it explores the search space by building a tree of nodes. Each node represents a decision made by the planner. Thus, the average number of planning nodes measures the efficiency of *planning* (as opposed to *plan execution*). We expect that the average number of planning nodes should decrease with learning, and that at the end of learning the average number of planning nodes using learned operators should be comparable to that using human expert-coded operators.

   When using OBSERVER-learned operators to solve test problems, the average number of planning nodes to solve each test problem is the sum of the number of nodes to generate the initial plan and the nodes for plan repair. It is compared with the average number of planning nodes to solve a test problem using expert-coded operators.

All the measurements according to these criteria vary for different runs due to the planner's random choice at decision points. Therefore, we measure the results five times during testing and compute the averages. We also compute the standard derivation of the performance of expert-coded operators from the different runs. We consider the performance of learned operators to be comparable to expert-coded operators if it is within one standard derivation from the average of the human level of performance.

6.3.2 Results

We now present the empirical results to evaluate the overall effectiveness of OBSERVER in the process planning and the DSN antenna operation domains.

In the process planning domain, thirty-three operators are learned from observation of expert solutions traces for one hundred problems. The test set consists of thirty-three randomly generated, previously unseen problems. The performance of the learned operators on the test set is measured after the initial operators are learned from observation, and then after OBSERVER refines the operators by practicing on additional batches of fifteen problems.

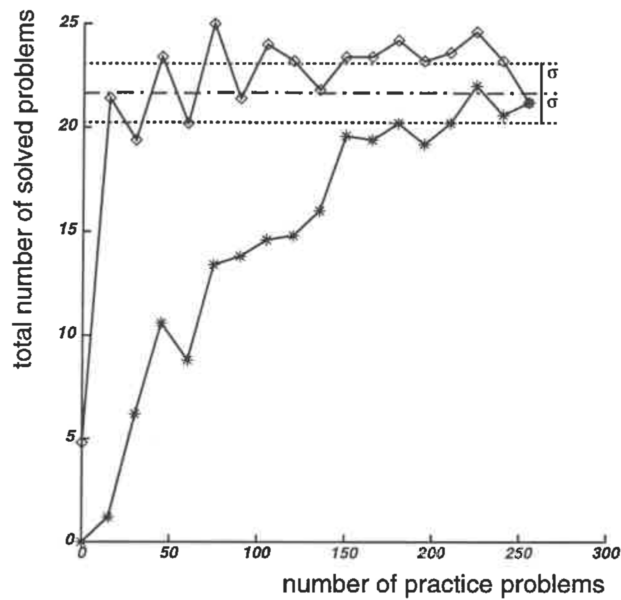
In the DSN antenna operations domain, twenty-four operators are learned from observation of expert solutions traces for twenty-four problems. The test set consists of thirty randomly generated, previously unseen problems. The performance of the learned operators on the test set is measured after the initial operators are learned from observation, and then after OBSERVER refines the operators during practice on additional batches of eight problems.

6.3.2.1 Total number of solved problems

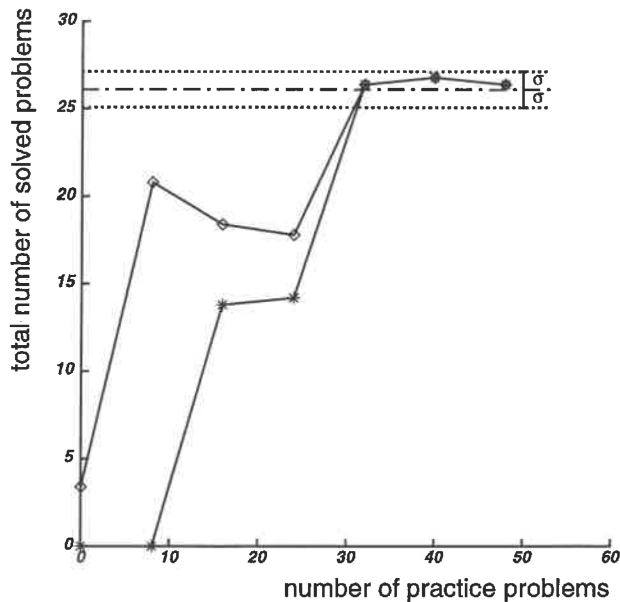
Figure 6.1 shows the total number of solved test problems using learned operators and expert-coded operators in the process planning domain and in the DSN antenna operation domain. The results in this graph show that:

1. Total number of solved test problems, with and without plan repair, increases with the number of practice problems.
2. At the end of learning, the total number of solved test problems using OBSERVER-learned operators is the same as using expert-coded operators.
3. The number of solved test problems when plan repair is used during testing increases much faster than if plan repair is not used. This indicates that our methods for planning with incomplete and incorrect operators and plan repair are effective for solving problems when domain knowledge is imperfect. It is important to be able to cope with imperfect domain knowledge, given gradual convergence for acquiring correct operators.
4. We see that in the process planning domain, before learning has completed, more test problems can be solved using learned operators with plan repair, than using human expert-coded operators. This is because the planner is given a resource bound **max_node_bound** during planning. Problems are considered not solvable

The Process Planning Domain:



The DSN Antenna Operations Domain:



—·— human expert-coded operators: average
········ human expert-coded operators: one standard derivation
✳——✳ learned operators: without failure
◇——◇ learned operators: with plan repair

Figure 6.1: Total number of solved test problems, as a function of the number of practice problems in training.

if no plan is generated within the node limit bound. Note that OBSERVER uses the **G-rep** (see Section 5) for planning, and that before learning is complete, the **G-rep** contains fewer preconditions than the human coded operators. Therefore OBSERVER needs to achieve fewer preconditions when using incomplete operators for initial planning. Thus, OBSERVER can sometimes generate plans for problems that are not solvable using human coded operators within the node limit. The plan repair mechanism then effectively repairs the plan by achieving additional preconditions in the **S-rep**. In this domain while learning is progressing, the **G-rep** happens to form an effective abstraction hierarchy for the operators, and therefore OBSERVER can solve more problems using incompletely learned operators than using human expert-coded operators.

This phenomenon (i.e., more test problems are solved using learning operators with plan repair than using human expert-coded operators), does not occur in the DSN antenna operation domain. Thus we do not make any general conclusions about the abstraction hierarchy OBSERVER naturally learns.

6.3.2.2 Average number of failures in solving a test problem

Figure 6.2 shows how the average number of failures that must be repaired to solve a test problem depends on the amount of practice. We see that:

1. The average number of failures before solving a test problem decreases as the number of practice problems increases.
2. At the end of learning, OBSERVER can solve problems without any failures. The average number of failures can be used by OBSERVER to determine when to stop learning.

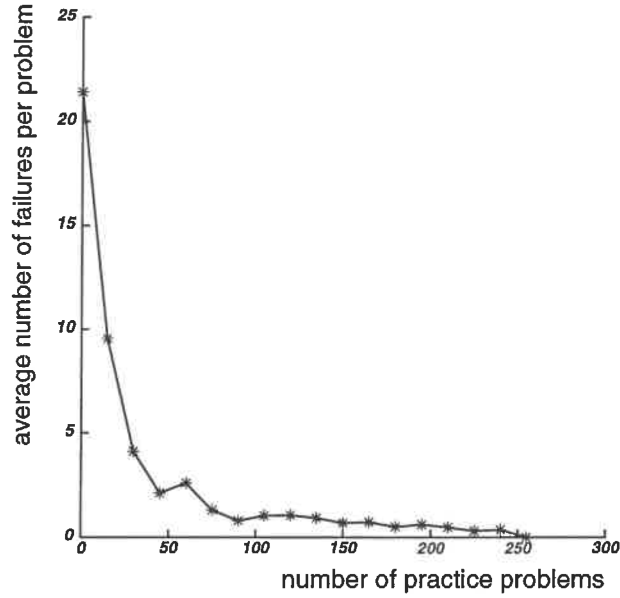
6.3.2.3 Average number of operator executions to solve each test problem

Figure 6.3 shows the average number of operator executions required to solve a test problem using learned operators and expert-coded operators.

For the learned operators, the average number of operator executions is the average of the sum of the number of operators that execute successfully and the number that execute unsuccessfully. For the human expert-coded operators, the average number of operator executions is simply the length of the plan, since we assume that the expert-coded operators are correct.

The following phenomena are worth noting from Figure 6.3:

The Process Planning Domain:



The DSN Antenna Operations Domain:

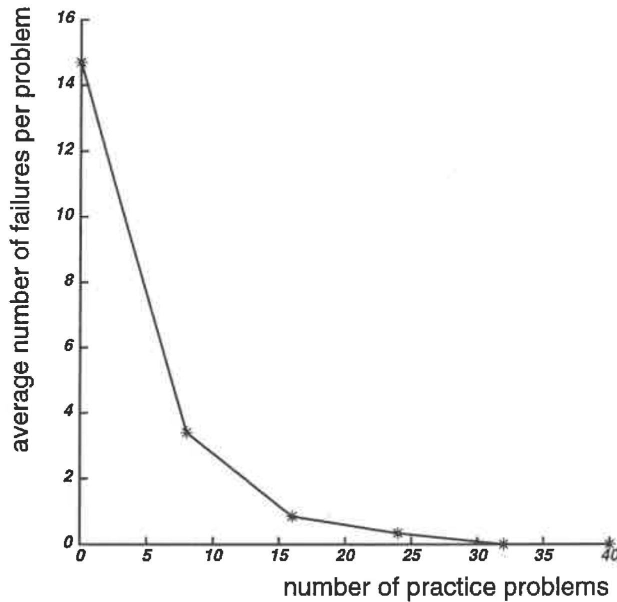
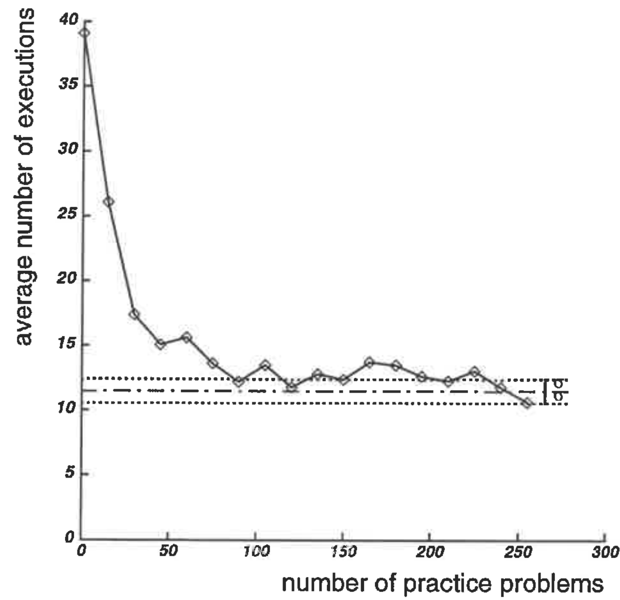


Figure 6.2: Average number of failures that must be repaired to solve each test problem, as a function of the number of practice problems in training.

The Process Planning Domain:



The DSN Antenna Operations Domain:

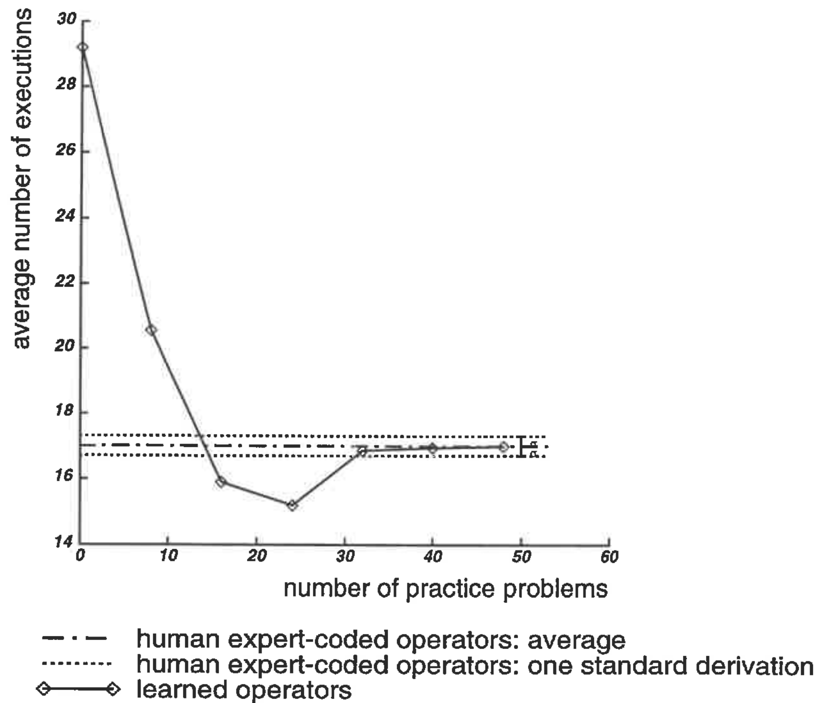


Figure 6.3: Average number of operator executions required to solve each test problem, as a function of the number of practice problems in training.

1. At the end of learning, the average number of operator executions to solve each test problem using the learned operators is comparable to that using expert-coded operators in both domains. This indicates that the efficiency of the solutions (i.e., the plan quality) using learned operators is comparable to the solutions using expert-coded operators.

2. In both domains, the average number of executions *decreases* (with minor exception in the DSN domain, as explained below) with more practice until equilibrium. The decrease stems from higher accuracy of the learned operators.

3. In the DSN domain, the average number of executions using learned operators dips significantly below the result using human expert-coded operators after learning from a number of practice problems, but before learning is complete. This is because before learning is complete, fewer test problems are solved using learned operators than using human expert coded operators (see Figure 6.1 for the number of problems solved). These solved problems are simpler than the problems that are solved using human expert-coded operators in that their solutions are shorter. To demonstrate this, Figure 6.4 shows the *ratio* of operator executions using learned operators to the operator executions using human coded operators, on the problems that are solved in *both* cases, where

   $operator_execution_ratio =$

   $$\frac{num_of_operatior_executions_using_learned_operators}{num_of_operatior_executions_using_human_coded_operators}$$

   on the same problems that are solved in both cases.

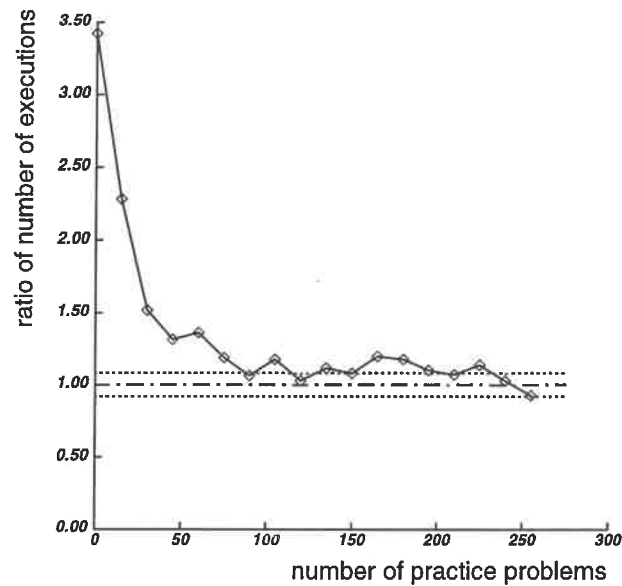
   We see that in both domains, this ratio decreases almost monotonically with more practice until it converges to one.

6.3.2.4 Average number of planning nodes to solve each test problem

Figure 6.5 compares the average number of planning nodes to solve each test problem using learned operators with that using human expert-coded operators. For the learned operators, the average number of planning nodes is the sum of the average number of nodes generated for initial planning and for plan repair. For the expert-coded operators, this number is simply the number of nodes generated during initial planning.

The following phenomena are worth noting:

The Process Planning Domain:



The DSN Antenna Operations Domain:

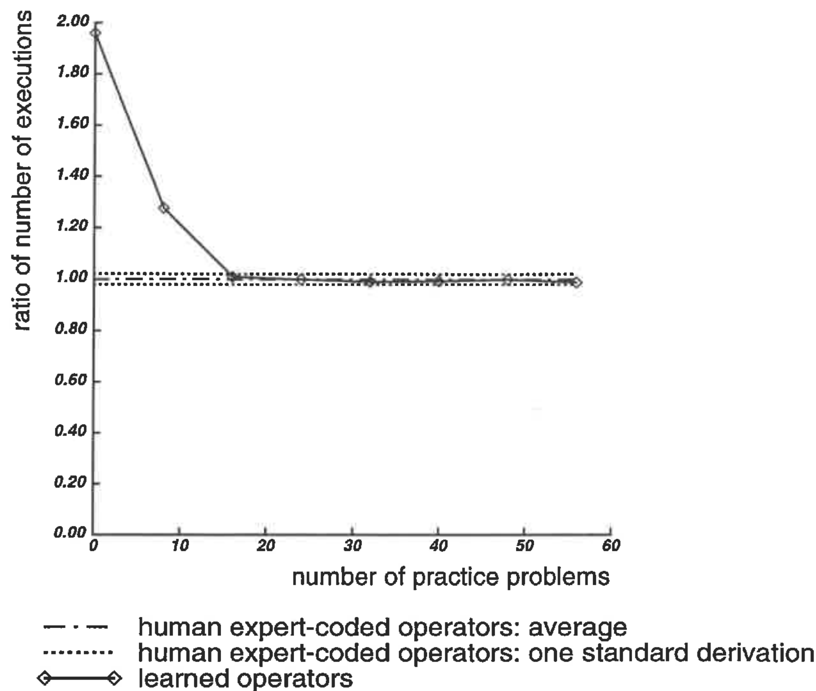
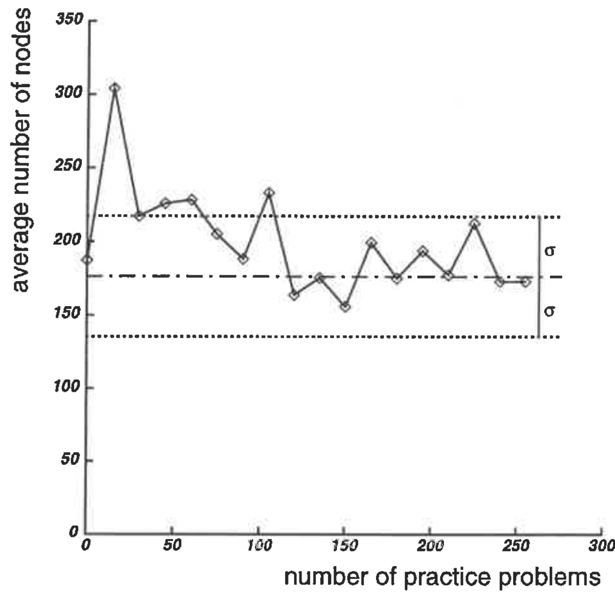
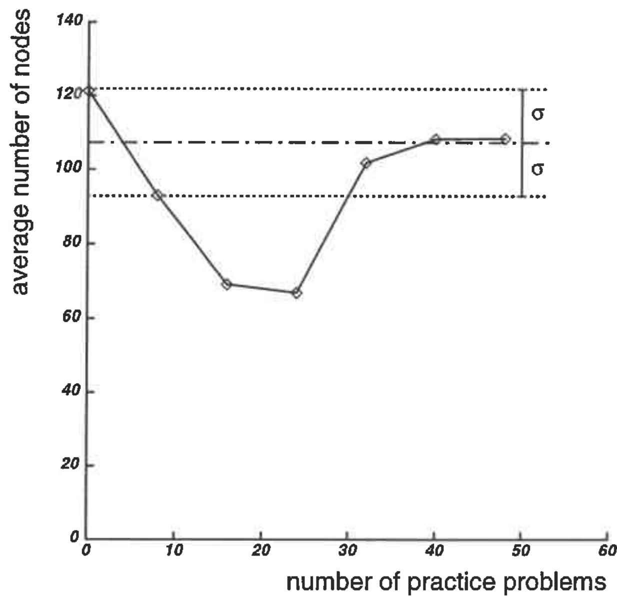


Figure 6.4: *Operator-execution ratio*, as a function of the number of practice problems in training.

The Process Planning Domain:



The DSN Antenna Operations Domain:



—·— human expert-coded operators: average
········ human expert-coded operators: one standard derivation
◇——◇ learned operators

Figure 6.5: Average number of planning nodes generated to solve each test problem, as a function of the number of practice problems in training.

1. At the end of learning, the average number of planning nodes generated to solve each test problem using the learned operators is comparable to that using human expert-coded operators in both domains.

2. The curves exhibit significant oscillation, because of the high variance in the number of planning nodes when the planner makes random choices at different decision points. If the planner makes a wrong choice, it may perform a great deal of unnecessary search before it backtracks to find the right decision. But if the planner makes the right choice the first time, it may find a plan very quickly.

3. In the result for the DSN domain, the average number of planning nodes using learned operators dips significantly below the result using human expert-coded operators after learning from a number of practice problems, but before learning is complete. As with the number of operator executions, this is because before learning is complete, fewer problems are solved using the learned operators than using human expert-coded operators (see Figure 6.1 for the number of problems solved). These solved problems are simpler in that the planning nodes for solving these problems are fewer. To demonstrate this, Figure 6.6 shows the *ratio* of planning nodes using learned operators to the number of planning nodes using human coded operators, on the problems that are solved in *both* cases, where *planning_nodes_ratio* =

$$\frac{num_of_planning_nodes_using_learned_operators}{num_of_planning_nodes_using_human_coded_operators}$$

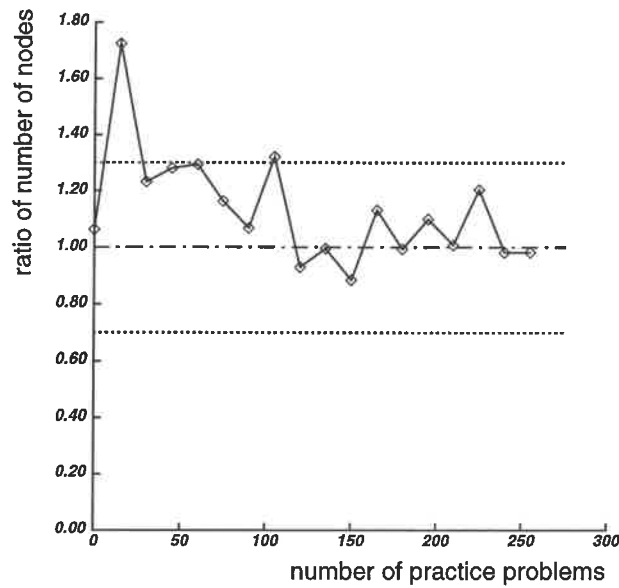
   on the same problems that are solved in both cases.

   We see that in both domains, this ratio decreases almost monotonically with more practice until it converges to one.

6.3.3 Summary of the effectiveness evaluation

In summary, the empirical results demonstrate that OBSERVER learns operators in the process planning and DSN antenna operations domains well enough to solve problems as effectively as with expert-coded operators, according to the following criteria: the total number of solved test problems, the average number of failures in solving a test problem, the average number of operator executions to solve each test problem, and the average number of planning nodes to solve each test problem. Moreover, OBSERVER learns equally effectively in two very different domains, providing evidence for the generality of our learning methods.

The Process Planning Domain:



The DSN Antenna Operations Domain:

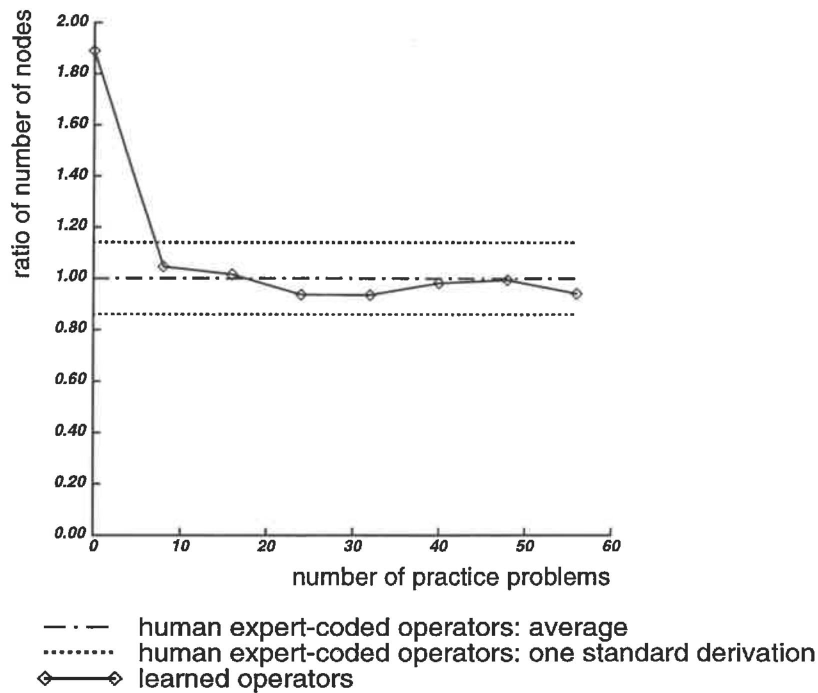


Figure 6.6: *Planning-nodes ratio*, as a function of the number of practice problems in training.

6.4 Role of Learning by Observation

This section evaluates the role of learning by observation. We describe the method for such evaluation and present empirical results.

To evaluate the role of observation, OBSERVER is run multiple times. During each run, OBSERVER is given the same problems during practice, but a *different number* of initial observation problems that are randomly drawn from a fixed set of problems. We measure the first criterion, i.e., the total number of solved test problems, after observation and practice, given a *different number* of initial observation problems.

Figure 6.7 shows the total number of solved problems, given a different number of observation problems, but the same set of practice problems.

The results in both domains show that:

1. The total number of solved test problems increases as the number of observation problems given to the system increases

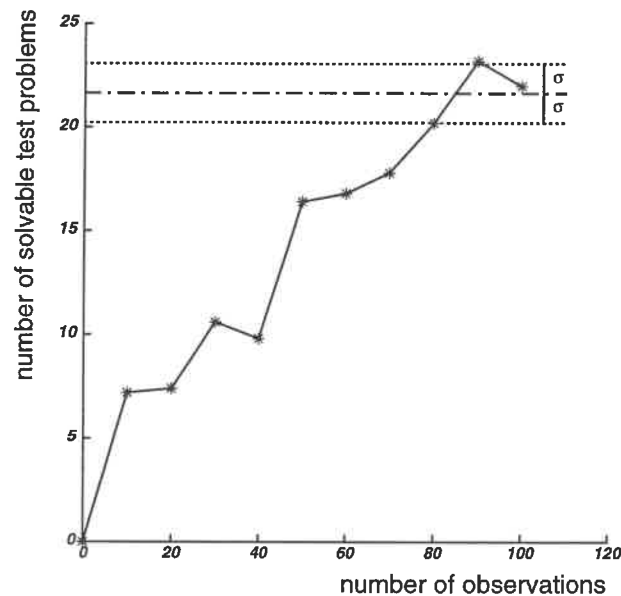
2. Learning by observation eventually reaches saturation. After observations of solution traces for eighty problems in the process planning domain, and twenty problems in the DSN domain, more observation problems do not significantly improve performance. When OBSERVER stops benefiting from more observation, it should start practicing

These empirical results in the two domains indicate that learning by observation contributes significantly to the learning process. This significance is due to the fact that generalization of domain constants to variables, as well as climbing up the type hierarchy for the variables are only feasible when learning by observation. This was a conscious decision when designing the learning system. Allowing OBSERVER to experiment with different domain constants or with types of objects not observed in expert solutions would dramatical increase the search space for planning with imperfect operators, and thus reducing the effectiveness of practice.

6.5 Role of Practice

In this section, we present empirical results to illustrate the role of practice. We describe our evaluation methods and then present empirical results.

The Process Planning Domain:



The DSN Antenna Operations Domain:

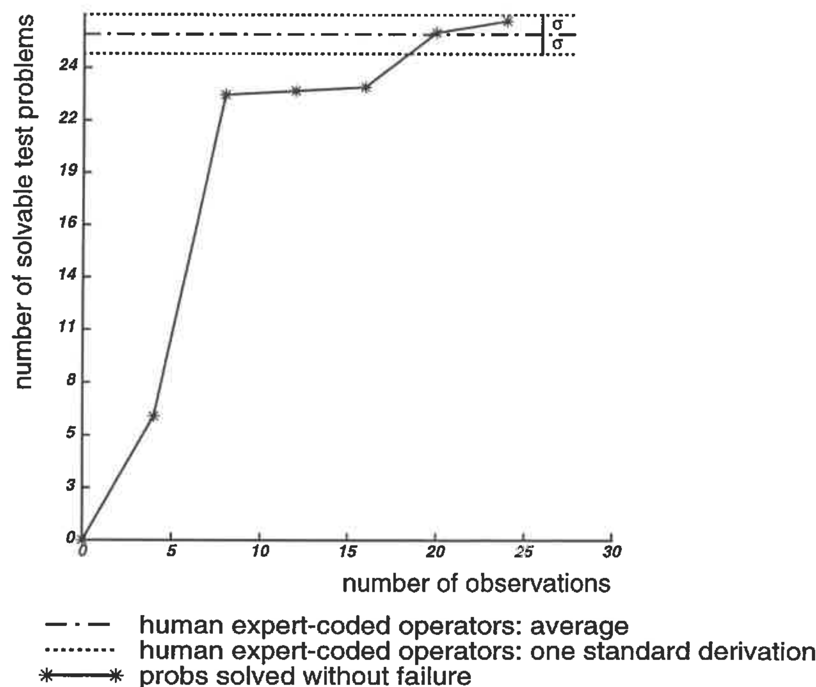


Figure 6.7: Total number of problems solved, using the same set of practice problems, but varying the amount of observation before practice. We see that after eighty observation training problems in the process planning domain and twenty problems in the DSN domain, more observation problems do not significantly increase the solvability. This indicates the appropriate point where learning by observation reaches saturation and where practice should start.

6.5.1 Method for demonstrating the role of practice

To demonstrate the role of practice, we show that, after initial operator acquisition by observation, OBSERVER learns operators better and faster if it practices using the planning and plan repair algorithms described in Chapter 4, than if it just continues observing expert solutions. To do so, we compare the learned operators in the following two scenarios, where the same initial problems are used for learning by observation, in terms of the total number of solved test problems.

scenario 1: OBSERVER is given a set of problems to practice. During practice, OBSERVER uses its planning and plan repair algorithms to solve practice problems, and refines any incorrect or incomplete operators using both successful and unsuccessful executions.

scenario 2: OBSERVER is given the same set of problems as in scenario 1; however, OBSERVER is only allowed to refine the operators based on the expert solutions for these problems, and is not allowed to practice (i.e., OBSERVER does not generate successful or unsuccessful executions for operator refinement.)

The evaluation of this criterion includes the following two measurements:

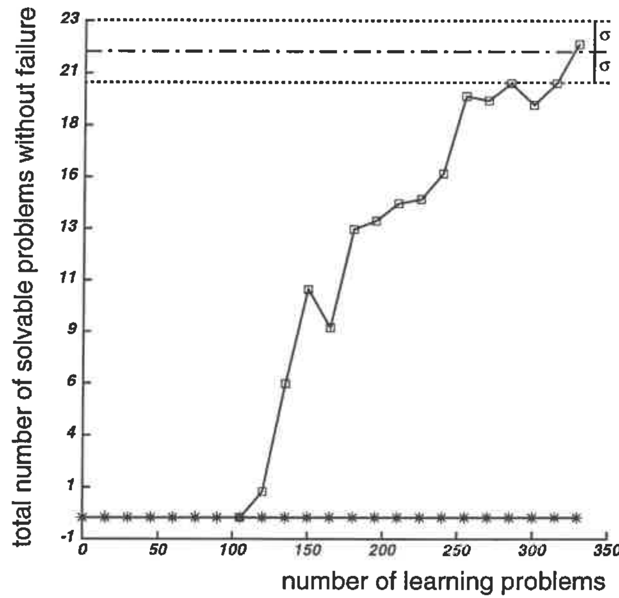
1. Total number of solved test problems without plan repair
2. Total number of solved test problems, including the problems that required plan repair

6.5.2 Results

Figure 6.8 illustrates the comparison of the learned operators that are refined through practice (scenario 1) with the operators that are refined based on observation only (scenario 2), in the total number of solved problems in the test set *without execution failure, and therefore without requiring plan repair.* We see that:

- If OBSERVER practices after learning by observation, the total number of solved test problems without execution failure increases steadily as the number of training problems increases
- If OBSERVER is only given the expert solution traces of the same problems used during practice in scenario 1, it cannot solve any test problems without failures

The Process Planning Domain:



The DSN Antenna Operations Domain:

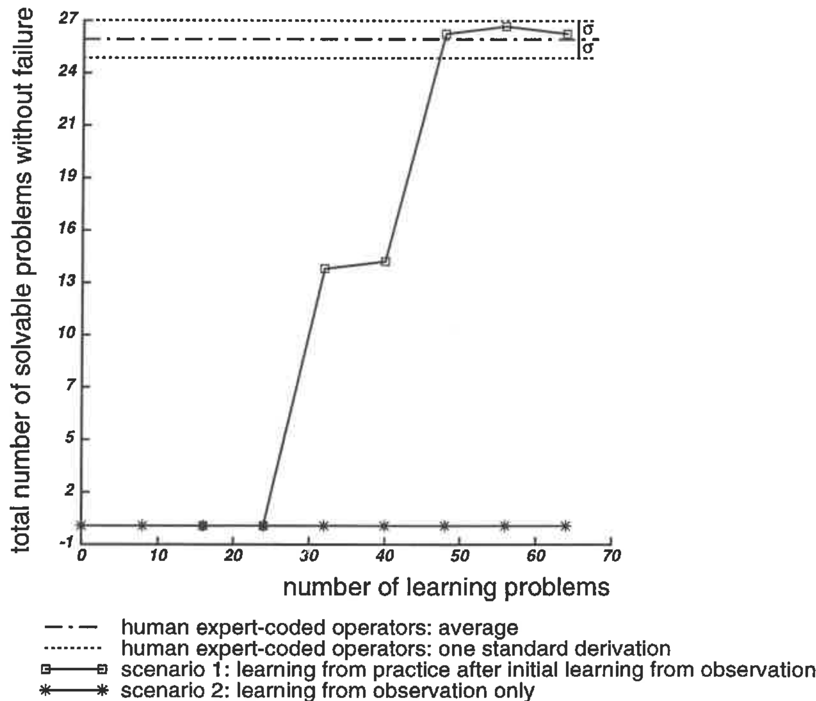
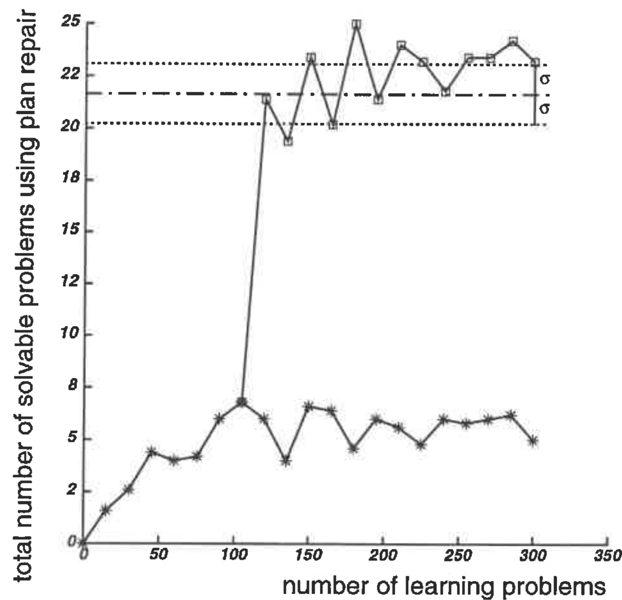


Figure 6.8: *Total number of solved test problems without execution failure during testing, as a function of the number of practice problems in training.* In both scenarios, OBSERVER first learns a set of operators from observation, as indicated by the first one hundred problems in the process planning domain and the first twenty-four problems in the DSN antenna operations domain. Then, OBSERVER refines operators from practice in scenario 1 or learns by observation only in scenario 2. The performances differ significantly in two scenarios.

The Process Planning Domain:



The DSN Antenna Operations Domain:

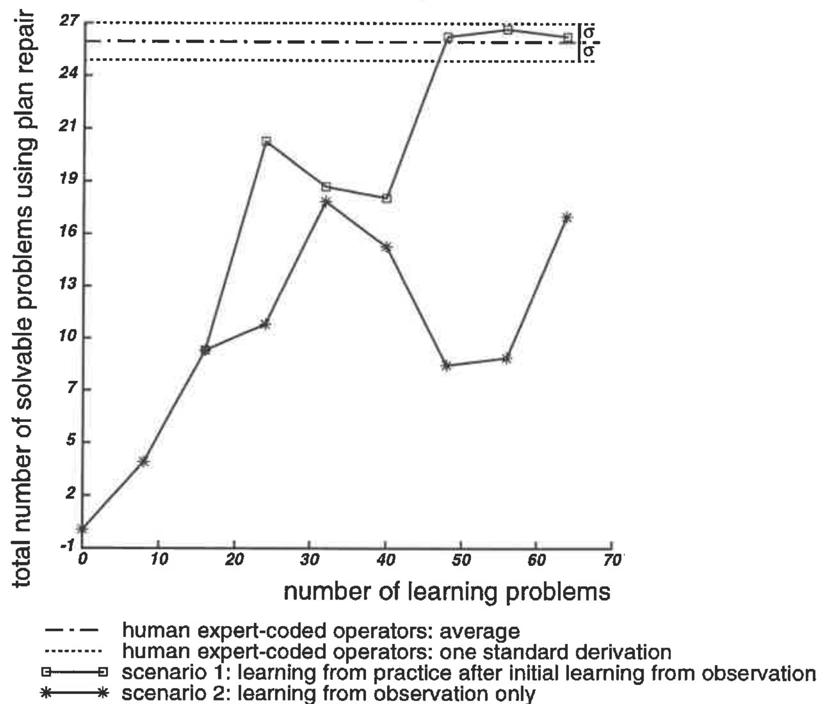


Figure 6.9: *Total number of solved test problems allowing execution failure and plan repair during testing, as a function of the number of practice problems in training.* In both scenarios, OBSERVER first learns a set of operators from observation, as indicated by the first one hundred problems in the process planning domain and the first twenty-four problems in the DSN antenna operations domain. Then, OBSERVER refines operators from practice in scenario 1 or learns by observation only in scenario 2. The performances differ significantly in two scenarios.

Figure 6.9 compares the two learning scenarios in terms of the total number of test problems that are solved, including problems solved where *executions failures occur but are subsequently repaired using our plan repair algorithm.* We see that:

- In both scenarios, the total number of solved test problems increases as the number of training problems increases, whether OBSERVER uses the training problems for practice or as observation
- The total number of solved problems increases much faster in scenario 1, where OBSERVER practices using our planning and plan repair algorithms, than in scenario 2, where OBSERVER is only given the expert solution traces of the same problems
- OBSERVER learns operators as well as a human expert if it practices, but is far from the human level of performance when it learns solely by observation

The above results can be explained by the fact that OBSERVER uses the **G-rep** for planning, and that the correctness of the plans generated depends on how well OBSERVER has learned the **G-rep**. With practice (scenario 1), OBSERVER is able to learn the **G-rep** of the operator preconditions using unsuccessful executions. But without practice (scenario 2), OBSERVER can never specialize the **G-rep**, because observations of expert solutions do not contain negative examples.[1]

6.5.3 Summary of the role of practice

In both comparisons (i.e., the total number of solved test problems *without execution failure* and the total number of solved test problems *if plan repair is used*), we see that significantly more test problems are solved using the learned operators in scenario 1, where OBSERVER refines operators during practice, than scenario 2, where OBSERVER can only learn by observation. We conclude that practice using our planning and plan repair algorithms is a crucial component of OBSERVER's learning architecture.

6.6 Summary

This section presented empirical results that demonstrated the effectiveness of our learning system, including the significance of and the synergism between learning by observation and learning by practice.

[1]One could also use the **S-rep** for planning to solve the test problems; however, the **S-rep** usually contains so many extraneous preconditions when learning only by observation that the planner cannot generate a plan within reasonable time.

The empirical results are presented in two domains: a process planning domain and a DSN antenna operations domain. Empirical results show that learned operators are as effective in problem solving as human expert-coded operators, and that both learning by observation and learning during practice contribute significantly to the learning process.

These two domains are significantly different. The empirical results are qualitatively the same in both domains in terms of the convergence properties and learning rate, although they are different quantitatively due to the differences between these two domains.

Chapter 7

Related Work

This chapter describes other research in areas closely related to this dissertation. Our work is a contribution to machine learning in structural domains, so the first part of related work is devoted to this area. Second, our work is a contribution to planning, including machine learning for planning systems, plan repair, and the integration of planning, learning, and execution. Related work in these areas are discussed. We also mention briefly more remotely related areas such as knowledge acquisition.

7.1 Rule Learning in Structural Domains

Much work in machine learning is concentrated on *attribute-based* domains in which each instance of a concept is characterized solely by a vector of values for a given set of attributes [Quinlan, 1986]. The operators that OBSERVER learns are in *structural* domains, in which each instance is composed of many objects and is characterized not only by the attributes of the individual objects it contains, but also by the relationships among these objects. In this section, we first review different algorithms developed for learning in structural domains. We then describe a number of implemented systems that learn in structural domains.

7.1.1 Algorithms for learning in structural domains

Version-spaces algorithm

Mitchell [Mitchell, 1978] gave an elegant framework for learning simple existential conjunctive concepts. The set of all hypotheses that are consistent with the sample is called the *version space* of the sample. The version space can be represented by maintaining

two boundaries of the version space: the S-set of the most specific hypotheses and the G-set of the most general hypotheses in the version space. The target concept consistent with all the data given so far lies in the space delineated by the S-set and G-set inclusively. Mitchell uses the *candidate elimination algorithm* to update the S-set and G-set and thereby reduce incrementally the set of consistent hypotheses. Given a new positive instance, the algorithm generalizes the S-set as little as possible so that they cover the new instance and remain consistent with past data, and it removes those elements of the G-set that do not cover the new instance. Given a new negative instance, the algorithm specializes elements of the G-set so that they no longer cover the new instance yet remain consistent with past data, and removes from the S-set those elements that cover the new negative instance.

Our approach to learning preconditions is different from the version-spaces approach in the following ways:

- The sizes of the S-set and G-set in the version spaces can grow exponentially in the number of examples, whereas `OBSERVER`'s the **S-rep** and **G-rep** are single generalizations

- The S-set and G-set in the version spaces approach are always the most specific and most general representation, respectively, that are consistent with the given examples, whereas the **S-rep** and **G-rep** are not guaranteed to be the most specific and most general boundary

- The S-set can only be generalized with more examples and the G-set can only be specialized; however, the **S-rep** can also be specialized and the **G-rep** be generalized when negated preconditions are learned. Such negated preconditions are not handled in the basic version-spaces formalism

- Unlike the version-spaces algorithm, `OBSERVER` handles typed variables. The variables used in the operator preconditions follow the object type hierarchy given as initial knowledge to `OBSERVER`. When updating the **S-rep** and **G-rep**, `OBSERVER` also updates the types of the objects

One computational problem associated with the version-spaces method is that the size of the S-set and G-set can become exponentially large in the number of examples [Haussler, 1989], and, thus, updating both S-set and G-set is exponential. The incremental Non-Backtracking focusing (INBF) algorithm [Smith and Rosenbloom, 1990] learns strictly tree-structured concepts, using the version-spaces approach, in polynomial time. When learning concepts with tree-structured features, the S-set contains exactly one concept [Bundy *et al.*, 1985] and can be computed in polynomial time; however, the G-set can still grow exponentially [Haussler, 1989]. The INBF algorithm updates the G-set by

only processing near misses. This approach is very similar to our approach for updating the **G-rep** of the operator preconditions. Smith and Rosenbloom have proved that, if we use the INBF algorithm, the positive examples plus the near misses will be sufficient for convergence if the initial set of examples is convergent. This is similar to our proof in Chapter 5; however, the description language for operators that OBSERVER learns is more complex than the strictly tree-structured concepts of INBF.

Learning maxmimal specific generalizations

Winston's [Winston, 1975] work on structural learning served as a precursor to the other learning methods in learning in structural domains. His system learns concept descriptions that characterize simple toy-block constructions. The toy-block assemblies are initially presented to the computer as line drawings. A knowledge-based interpretation program converts these line drawings into a semantic-network description. Winston also uses this semantic-network representation to describe the current concept and some background knowledge about toy blocks. The generalization algorithm uses the training instances incrementally to update a *single* current concept description in a depth-first fashion. The learning system does not concern itself with the possibility that the training instance matches the hypothesis in multiple ways or with the problem that there are multiple ways of generalizing or specializing the hypothesis. The learning system assumes that the training instances are presented in good pedagogical order, so that ambiguity is unlikely to arise. It also assumes that negative instances are near misses. These assumptions make it impossible to apply to more complex situations, such as our operator learning problem.

Vere's counterfactuals algorithm [Vere, 1980] learns concepts that are consistent with a set of positive and negative examples, by computing "counterfactuals." A "counterfactual" is a set of conditions which must be false if a generalization is to be satisfied. It computes multilevel counterfactuals by recursively reducing the original induction problem to a smaller "residual" problem, whose generalization gives the desired counterfactual. The counterfactuals algorithm can learn concepts with conjunctions, disjunctions, and negations. But it is not an incremental algorithm and it requires both positive and negative examples. Moreover, the counterfactual algorithm requires the computation of the maximally specific common generalizations of two given positive example, which is an exponential process.

Hayes-Roth's interference matching algorithm [Hayes-Roth and McDermott, 1978] is a heuristic algorithm that learns existential conjunctive concepts given positive examples only. The interference-matching algorithm starts out as a breadth-first search of all possible matches of one positive example with another. The search proceeds by "growing" common subexpressions until a space limit is reached. Unpromising matches are then pruned with a heuristic utility function and the growing process continues in a depth-first

fashion. The utility of a partial match is equal to *the number of predicates matched less the number of variables matched*; however, this heuristic is likely to prefer matches that miss some true preconditions.

Inductive Logic Programming (ILP)

Inductive logic programming [Muggleton, 1992] focuses on induction of logic programs from examples and has roots in early work in generalization in logic [Plotkin, 1970] and logic-program [Shapiro, 1983]. Given background knowledge, ILP induces complex concepts represented in Horn-clause logic; thus it learns in structural domains. A weakness of ILP methods is their inability to use existing knowledge to guide search through the extremely large space of possible concepts. As a result, search in an ILP system is often guided using an existing domain theory [Bergadano and Giordana, 1988, Cohen, 1991, Pazzani and Kibler, 1992] or by human [Sammut, 1996]. It remains a challenge to apply ILP to complex concept learning systems. We describe several well-known ILP algorithms related to OBSERVER's learning algorithm.

FOIL [Quinlan, 1990] is an inductive learning system that learns Horn-clause theories. Its outer loop is a greedy covering algorithm that learns one clause at a time. Each clause is constructed to maximize coverage of positive examples while excluding all negatives. Clauses are constructed by adding one literal at a time, using steepest-ascent hill-climbing. At each step, the literal that maximizes an information gain metric is added to the clause. Literals are added until all negative examples are removed. FOIL requires both positive and negative examples and it is not incremental; therefore, it cannot be used for learning operator preconditions by observation.

Golem [Muggleton and Feng, 1990] is an ILP system which learns a set of conjunctive clauses that cover a given set of positive and negative examples and extensional background knowledge. Clauses are induced by choosing a random seed instance from the set of positive instances and generalizing in a specific to general fashion as much as possible without covering any negative instances. Golem cannot be applied directly to OBSERVER's learning problem, because: *(i)* Golem is not an incremental learning algorithm, *(ii)* Golem does not learn negated literals, *(iii)* Golem requires both positive and negative examples, whereas negative are not available to OBSERVER while learning by observation, and *(iv)* Golem cannot be applied to learn the operator effects.

Progol [Muggleton, 1995] is another ILP algorithm similar to Golem. Progol begins by taking the description of a single positive example and saturates the description by adding to it literals derived from background knowledge. Progol permits *mode* declarations to restrict the application of predicates to avoid an explosion in the number of literals that the saturation procedure may try. When saturation is completed, Progol has a *most specific* clause to serve as a bound on a general-to-specific search. Beginning with the

most general clause, i.e., one with an empty body, Progol tries to find a subset of the most specific clause that satisfies a minimum description length criterion. Due to the large number of possible literals in the saturation procedure, Progol has mainly been used with human guidance in its *mode* declaration [Sammut, 1996]. For the same reasons as in Golem, Progol is not a good tool for learning operators preconditions and effects. But is a promising tool for learning functions and constraints for operators (for descriptions of functions see [Carbonell *et al.*, 1992]).

7.1.2 Systems for learning in structural domains

A number of systems have been implemented that learn in structural domains using the algorithms described above or other more specialized algorithms. We compare OBSERVER with these systems, along the following dimensions.

The type of learned knowledge. This falls into two broad areas: *concept learning* and *control learning*. Research in concept learning focuses on learning concept definitions from both classified examples and/or an existing incomplete and/or incorrect domain theory. Research in control learning focuses on learning search-control knowledge that improves the performance of an existing problem-solver. OBSERVER is a concept learning system, where the concept learned is planning operators.

The source of learning. Generally, two types of input sources can be explored by a learning system. One source comes from expert solution traces and the other comes from examples the learning system generates in a learning-by-doing fashion. Some systems explore one type of source of knowledge, whereas others combine both sources. OBSERVER learns from both types of inputs.

The learning algorithm. Many different types of learning algorithms are feasible for learning in structural domains, including inductive learning algorithms, for example, ILP, explanation-based learning algorithm, and inverse resolution. Different types of learning algorithms may be combined. OBSERVER uses an inductive learning method similar to the version-spaces approach.

Initial knowledge. Systems for learning heuristics assume initial correct domain knowledge. Systems that learn domain knowledge may either learn from scratch or refine existing partially correct domain knowledge. OBSERVER starts with very little initial knowledge, i.e., it has no knowledge about the preconditions or the effects of the operators at the start of learning.

LEX

LEX [Mitchell *et al.*, 1983] is a system that learns heuristic problem-solving strategies through experience in the domain of symbolic integration—LEX starts with operators with exact preconditions for matching the integral but lacks preconditions (control rules) for determining when the operators should be applied. LEX acquires and modifies heuristics by iteratively applying the following process: (*i*) generate a practice problem, (*ii*) use available heuristics to solve this problem, (*iii*) analyze the search steps performed in obtaining the solution, and (*iv*) propose and refine new domain-specific heuristics to improve performance on subsequent problems.

LEX learns problem-solving heuristics, whereas OBSERVER learns the preconditions and effects of the operators. The left-hand side of the heuristics is similar to the operator preconditions—they are both conjunctive literals. The left-hand side of LEX's heuristics are represented as version-spaces and thus the size of the G-set and S-set grow exponentially in terms of the input. This does not pose a big problem for LEX, since it is not concerned with learning from observing other agents, which would have caused the exponential growth of the S-set. OBSERVER learns the preconditions by keeping the **G-rep** and **S-rep**, each of which is a single boundary and is learned in polynomial time. The right-hand side of the heuristics is simply the decision of choosing which operator to apply when the left-hand side is satisfied, whereas the operator effects, with the add and delete lists and conditional effects, are much more complex. LEX uses its own problem solving traces to refine its heuristics, whereas OBSERVER is also concerned with learning from observations of other agents.

Since LEX learns search heuristics where the domain operators are given, the applicability of a heuristic can be determined from a problem-solving episode. OBSERVER learns domain operators, and the applicability of an operator can only be determined by execution the operator in the environment. Since execution failure may occur, OBSERVER integrates planning, execution, and plan repair; this issue does not arise in LEX.

ALEX

ALEX [Neves, 1985] is a program that acquires rules to solve simple linear algebraic equations by learning from worked-out examples in a textbook and from its own problem solving traces. Each rule is learned only once from one example, using several heuristics, and cannot be refined using multiple examples. ALEX's problem solver uses mean-ends analysis and is much simpler than the PRODIGY planner used by OBSERVER.

Porter and Kibler

Porter and Kibler [Porter and Kibler, 1986] examined the process of learning problem-solving heuristics using Experimental Goal Regression (EGR). The learner first forms heuristic rules from the user's suggestions and these heuristics are then used to guide the problem solver on subsequent problems. The learner also generates training examples automatically by perturbation. Perturbation automates the generation of training instances by making small changes to a single training instance supplied by the teacher. The learner than solves these problems and classify the examples as positive or negative. The positive examples are used for refining the heuristics. The left-hand side of the heuristics are learned in a conservative specific-to-general manner, which involves dropping conditions, turning constants to variables, and climbing concept hierarchy trees. This is similar to learning the **S-rep** of the operator preconditions in OBSERVER; In addition to learning from positive examples, EGR learns from positive examples only, because their negative examples may be incorrectly classified. In addition to learning from positive examples, OBSERVER also learns from negative examples during practice.

The idea of perturbation is similar to near miss negative examples that OBSERVER uses for specializing the **G-rep**. A near miss is a perturbation on the **S-rep** of the operator preconditions, but not necessarily on the pre-state of a positive example. EGR first generates a perturbation and then solves the problem, whereas OBSERVER reverses this order by solving a practice problem using planning, plan repair, and execution, and thus generating a near miss (i.e., perturbation). The reverse of the order is due to the fact that there may be many extraneous preconditions in the **S-rep** and achieving all of them during planning is resource-consuming and unnecessary.

OBSERVER also differs from EGR in that OBSERVER repairs plan for execution failures. EGR does not have sophisticated problem solving capability; thus, if a learned episode fails to solve a problem, no additional search (i.e., repair) is performed to find a solution.

Another contribution of Porter and Kibler's work is their algorithms for preventing over-generalization by adding needed constraints using background knowledge. Built-in biases (such as prefer more primitive relations) reduce the branching factor for searching a constraint with maximal coverage and minimal complexity. This may be incorporated in OBSERVER to learn functions and constraints of the operators (for descriptions of functions see [Carbonell *et al.*, 1992]).

SIERRA

SIERRA [Vanlehn, 1987] is a program that learns algebraic procedures incrementally from examples, where an example is a sequence of actions. SIERRA requires the teacher to give a sequence of "lessons," where a lesson is a set of examples that is guaranteed to

introduce only one new subprocedure. The conditions (left-hand sides) of the procedures are learned in a conservative specific-to-general fashion, using the candidate elimination algorithm for updating the S-set in version-spaces. OBSERVER differs in that it does not require the expert to give perfectly ordered examples; thus imposing much less burden on the expert. OBSERVER also generates training examples during practice for further operator refinement.

ARMS

ARMS [Segre, 1988] is an EBL system that acquires the ability to plan sequences of robot motions to accomplish assembly of simple mechanisms. It learns by unobtrusively observing an expert guide the robot through an assembly task via the robot arm's teaching pendant. Given an observation of an expert solution sequence, ARMS first builds a causal model to understand (explain) how the user solves the problem, using its domain theory. ARMS domain knowledge is represented using five different categories of schemata and is fairly sophisticated. ARMS then generalizes the explanation to produce a new composite operator schema, which can be subsequently used for both understanding and planning. Since ARMS uses explanation-based learning to learn from single example, good domain knowledge is crucial. In contrast, OBSERVER is an inductive learning system that generalizes from multiple examples.

LEAP

LEAP [Mitchell *et al.*, 1990] is a learning apprentice system for digital circuit design. It provides interactive advice to a circuit designer on how to hierarchically decompose abstract circuit modules into submodules, eventually resulting in a gate-level circuit design. It uses explanation-based learning and is able to learn new rules for decomposing circuit modules, as well as rules for choosing among alternative decompositions. LEAP is limited to learning rules for designing circuits with fairly simple functional specifications (primarily boolean functions), since its EBL methods require verification of the correctness of user circuits in order to generalize them. In contrast, OBSERVER uses an inductive learning technique.

DISCIPLE

DISCIPLE [Kodratoff and Tecuci, 1991] is a learning apprentice system that learns rules used for designing detailed manufacturing technologies, for establishing the assimilation strategy of a new product, and for aiding the manager of a computer center in the decision activity. It uses both explanation-based and similarity-based learning methods.

CAP

CAP [Hume and Sammut, 1991] is a system that observes sequences of actions performed by other agents and forms a theory that can be generalized and tested by experimentation. CAP uses inverse resolution to search for a theory, while OBSERVER uses induction. The training examples generated by CAP are dependent upon the current state of the world and, hence, it is "opportunistic" in the sense that it attempts to test hypotheses as circumstances permit. In contrast, OBSERVER uses sophisticated planning and plan repair algorithms to generate its own learning examples.

LIVE

LIVE [Shen, 1989, Shen, 1994] is a system that learns and discovers from the environment. It integrates action, exploration, experimentation, learning, and problem solving; however, it does not learn from observing others. LIVE has only been tested in simplistic domains (e.g., the Little Prince World with about fifteen possible states), and does not scale up to large domains. LIVE also avoids the complexity of planning with incomplete and incorrect operators by using a set of *domain-dependent search heuristics* during planning. These heuristics are part of the input to LIVE. OBSERVER differs in that it deals explicitly with imperfect operators without relying on any human-coded, domain-dependent heuristics.

EXPO

EXPO [Gil, 1992, Gil, 1994] is a system that refines initial approximate planning operators through learning by experimentation. EXPO is given incomplete operators with missing preconditions and effects. Learning is triggered when plan execution monitoring detects a divergence between internal expectations and external observations. EXPO designs experiments to learn the missing preconditions and effects, using domain-independent heuristics. OBSERVER differs from EXPO in the following ways:

- The initial knowledge given to the two systems is different. EXPO starts with a set of operators that miss some preconditions and effects, whereas OBSERVER starts with no knowledge about the preconditions or the effects of the operators
- OBSERVER learns from observations of expert solution traces, whereas EXPO does not
- EXPO only copes with incomplete domain knowledge, i.e., overly-general preconditions and incomplete effects of the operators, whereas OBSERVER also copes with

incorrect domain knowledge, i.e., operators with overly-specific preconditions and overly-general preconditions

- Operators are learned in a general-to-specific manner in EXPO, whereas OBSERVER uses a version-spaces like approach that generalizes the **S-rep** using positive examples and specializes the **G-rep** using near miss negative examples

The Operator Learner

The Operator Learner [desJardins, 1994] is an inductive learning-based tool for knowledge engineering within SOCAP, a prototype military operations planning system based on AI generative planning technology [Wilkins, 1988]. The system is given a set of partial operators that are edited by experts and inductively "fills in the blanks" of these partial operators, identifying approximate preconditions for the operators by generalizing using feedback from evaluation models, as well as from user's planning choices. OBSERVER starts with much less knowledge about the planning operators and learns from observing expert solution traces in addition to learning from its own execution traces.

Benson's work

Benson's system [Benson, 1995] learns action models from its own experience and from its observation of a domain expert. Benson's work uses a much simpler action model formalism that is suited for reactive agents without complex deliberation, which is different from STRIPS operators. It uses GOLEM [Muggleton and Feng, 1990] for inductive learning. It does not have a complicated planning and plan repair mechanism and relies on an external teacher when an impasse is reached during planning.

Oates and Cohen's work

Oates and Cohen [Oates and Cohen, 1996] described a method for learning planning operator with context-dependent and probabilistic effects. In their domain model, the agent is assumed to have a set of m sensors and a set of n possible actions. At each time step, each sensor produces a single categorical value. Thus, their state is represented as a vector and their precondition learning is a type of learning for *attribute-based* domains, as opposed to *structural* domains, as in OBSERVER. In addition, OBSERVER learns incrementally whereas [Oates and Cohen, 1996] is nonincremental.

To summarize, OBSERVER differs from all the algorithms and systems described above in that it uses a novel polynomial inductive algorithm for learning operators, and that OBSERVER learns from observations of experts as well as from its own execution traces that are actively generated using planning.

7.2 Related Work in Planning

The second major part of the contribution of this thesis is related to planning, hence we review related work on planning, learning, and coping with less-than-perfect domain knowledge.

7.2.1 Machine learning in planning

Most work on learning in planning systems has focused on learning control knowledge in order to speed up learning. The methods for speed-up learning include explanation-based learning (EBL) [Korf, 1985, Sacerdoti, 1977, Mitchell, 1983, Laird *et al.*, 1986, Chien, 1989, Minton, 1988, Minton *et al.*, 1989, Mostow and Bhatnagar, 1987], analogical learning [Veloso, 1994], and learning abstraction hierarchies [Knoblock, 1994]. Other work in learning control knowledge has addressed the issue of increasing the plan quality [Iwamoto, 1994, Pérez, 1995].

Our work differs in that OBSERVER learns domain models, i.e., planning operators. This is a important and challenging problem, because knowledge engineering is a key bottleneck for fielding planning systems. Some work that addressed the issue of learning planning operators in the context of classical planning systems includes [Gil, 1992, desJardins, 1994, Wang, 1995]. More recent work addressed the issues of learning planning operators with probabilistic effects [Tae and Cook, 1996, Oates and Cohen, 1996].

7.2.2 Planning with incomplete information

Early work in planning assumed complete information (see [Allen *et al.*, 1990]). As planning moves on to real-world problems, this assumption is no longer valid. Information for planning can be incomplete in several ways, including incomplete information about the state of the world, or the preconditions and/or effects of actions. In our work, we have extended the classical planning framework to handle incomplete and incorrect operators in the context of operator learning, which is a key characteristic that distinguishes our work from most other work in planning with incomplete information.

State information can be incomplete in several ways. Some researchers addressed the issue of planning with incomplete initial state information [Genesereth and Nourbakhsh, 1993, Kushmerick *et al.*, 1995]. Different approaches are taken to address the issues of planning with incomplete state information and planning to seek for relevant information [Etzioni *et al.*, 1992, Draper *et al.*, 1994].

Knowledge of actions in the domain can be incomplete in several ways. Numerous probabilistic techniques for dealing with sensor and actuator noise have been proposed

and explored in the context of mobile robotics [Cassandra *et al.*, 1994, Simmons and Koenig, 1995]. And a more general model for dealing with probabilistic knowledge in states, state transitions, and external events is being developed in [Blythe, 1994, Blythe, 1996].

7.2.3 Plan repair

Several previous investigations have addressed the problem of plan repair after an execution failure or unexpected external events, but none has addressed the issue of planning and plan repair in the context of incomplete and incorrect operators, as in OBSERVER.

CHEF [Hammond, 1989] is a case-based planning system for cooking. In CHEF, failures are all due to unforeseen goal interactions. The CHEF system classifies a failure, infers a missing goal, and applies a critic to repair the plan. This problem differs from our replanning problem—in our case, the problem is not an unrecognized goal interaction (the PRODIGY planner takes care of goal interactions) but an execution failure where some preconditions of the operator are not satisfied. Thus, instead of applying a critic as in CHEF, OBSERVER uses the specific representation of the operator preconditions to determine which additional preconditions must be achieved to repair the failed plan.

SIPE [Wilkins, 1988] also performs replanning in response to unexpected external events that change the state. SIPE first classifies the failure type and then uses this classification to apply a critic to repair the plan. SIPE assumes that the planner has correct domain knowledge, whereas OBSERVER uses an incomplete and incorrect domain knowledge and relies on the environment (or a simulator) to provide feedback.

Knoblock [Knoblock, 1995] developed a system for information gathering from large networks of distributed information. His approach integrates previous work on planning, execution, sensing, and replanning, and extends previous work to support simultaneous and interleaved planning and execution. The role of replanning is to recover from execution failure. The planner replans the failed portion of the plan while maintaining as much of the executional plan as possible. This is supported by requiring the domain designer to define a set of domain-specific failure handlers. When a failure occurs, the failure handler removes the failed portion of the plan and updates the model to avoid the same failure when the failed actions are replanned. OBSERVER differs in that it uses domain-independent method for plan repair and, hence, does not rely on the domain expert.

Some other previous work concentrates on adapting an existing plan (or case) to the new problem situation [Veloso, 1994, Kambhampati, 1990, Simmons, 1988]. These algorithms involve adding and deleting activities from the original plan based on an analysis of the applicability of the dependencies to the current problem context. OBSERVER differs

from these work in two ways: first, OBSERVER does not have a correct domain model for planning, whereas these systems do. Second, OBSERVER interleaves planning and execution, whereas these systems use a *plan-then-execute* paradigm, because they assume a correct domain model.

7.2.4 Interleaving planning and execution

Interleaving planning and execution has received some attention in the planning community. Earlier work developed frameworks for integrating planning and execution [Ambros-Ingerson and Steel, 1988, Kuokka, 1990]. Later work focused more on specific methods for interleaving planning and execution. [Firby, 1987, Beetz and McDermott, 1992] emphasize the ability to react to unexpected situations rather than assuming that a plan will usually work. [Etzioni *et al.*, 1992] uses execution to find missing information needed by planning. [Genesereth and Nourbakhsh, 1993] uses execution to make planning more efficient. [Knoblock, 1995] integrates planning and execution to allow the system to plan for new goals as they arrive, replan failed actions, and exploit sensing operations. [Haigh and Veloso, 1996] learns planning knowledge from execution. OBSERVER [Wang, 1996] differs in that learning operators is a primary goal.

7.3 Knowledge Acquisition

Considerable research has focused on knowledge acquisition for rule-based expert systems [Boose and Gaines, 1989, Davis, 1979], and for object-oriented/inheritance knowledge bases with procedures and methods [Gil and Tallis, 1995]. These research concentrates on knowledge acquisition *tools* and all requires significant amount of direct interactions between AI experts and domain experts. These tools enable domain experts to write and debug domain theories without relying on AI people. OBSERVER differs from all these approaches in that it is fully automated and learns domain knowledge through observing expert's solution traces and through direct interactions with the environment during practice.

Chapter 8

Conclusions

In this final chapter, we summarize this thesis and its contributions. We also outline some directions for future work.

8.1 Summary of the Thesis

We have developed a general framework for automatic acquisition of planning operators. Our approach is to learn operators by observing expert solution traces and to incrementally refine these operators through practice in a learning-by-doing paradigm. During observation, OBSERVER uses the knowledge observable when experts solve problems, without the need for explicit instruction or interrogation. During practice, the system generates its own learning opportunities by solving practice problems and refines the initial incomplete operators using the training examples thus generated.

The main research issues we addressed in this thesis span the areas of machine learning and planning. From a learning perspective, OBSERVER incrementally learns operators using inductive learning techniques from observation of expert solution traces and from its own execution traces during practice. OBSERVER learns operator preconditions by building a general representation, the **G-rep**, and a specific representation, the **S-rep**, in a manner similar to the version-spaces method. Both the **G-rep** and **S-rep** are updated in polynomial time. OBSERVER also incrementally learns the variables and effects of the operators.

From a planning perspective, OBSERVER integrates planning, learning, and execution, and effectively generates learning examples for operator refinement. During practice, the individual plans generated for top-level goals achieve the preconditions in the **G-rep** of each operator, but do not require achieving preconditions in the **S-rep**. This has the

advantage of quickly generating an initial (although possibly erroneous) plan, which will provide opportunities for operator refinement. OBSERVER then executes the plan in the environment. If an operator executes successfully when some preconditions in the **S-rep** are not satisfied in the state, then they are removed from the **S-rep**, resulting in more general operators; however, an operator may fail to achieve its intended effects due to unmet preconditions, necessitating plan repair. During plan repair, the preconditions in the **S-rep** are used to determine which additional preconditions need to be achieved to make the failed operator applicable. The near miss negative examples are used for specializing the **G-rep**.

We have fully implemented the algorithms and applied them to a process planning domain and a DSN antenna operations domain. Empirical results in both domains show that learning converges to the human level of performance in a reasonable amount of time. The results also show that the automatically learned operators can solve problems as effectively as expert human-coded operators according to four criteria: (*i*) total number of solved test problems, (*ii*) average number of failures in solving a test problem, (*iii*) average number of operator executions to solve each test problem, and (*iv*) average number of planning nodes required to solve each test problem.

In addition, empirical results show that both observation and practice are crucial components of the learning system. The total number of solvable problems in a test set increases with the number of observation problems the system is initially given, but eventually reaches saturation. OBSERVER is able to learn operators better and faster (i.e., solving more problems using the learned operators) if it practices using the planning and plan repair algorithms described in Chapter 4, than if it just continues observing expert solutions.

8.2 Contributions

To summarize, this thesis has made the following contributions:

- A novel, domain-independent approach to automatic acquisition of planning operators (i.e., learning by observation and practice). This thesis addressed a key knowledge acquisition bottleneck for building useful planning systems. Our approach does not require much interaction with domain experts
- A framework for integrating planning, learning, and execution. This framework has shown to be effective in both operator learning and problem solving with incomplete and incorrect domain knowledge
- A polynomial inductive algorithm for learning operator preconditions and effects

- An effective algorithm for planning and plan repair, which solves problems using incomplete and incorrect operators

- A demonstration of the effectiveness of integrated learning and planning in two complex domains, a process planning and a DSN antenna operations domains

8.3 Future Work

This thesis has addressed an important issue in fielding planning systems for realistic domains—the knowledge acquisition bottleneck. Our framework for learning planning operators opens some new areas for future research, as discussed in this section.

8.3.1 Handling uncertainty

OBSERVER assumes deterministic operators and error-free perception. Relaxing this assumption is an interesting and important direction for future work. There are several sources of uncertainty in a domain. The first is perception noise in which observation of states may be incorrect—some properties may be missing in the observation, while some others may be erroneously observed. The second form of uncertainty is in the operators—operators may have probabilistic effects. The third type of uncertainty comes from external exogenous events that change the state in a manner not fully predictable by the planner.

To develop a learning system to handle uncertainty in the domain, one must first study what types of noise are present. Assuming no noise in the domain enables OBSERVER to converge quickly in complex domains; however, our framework for learning operators by observation and practice is also valid for acquiring operators in the presence of noise. The learning algorithm can be extended to handle noise by, for example, maintaining an occurrence count for each literal in the **S-rep** and **G-rep**. The system would only remove preconditions from the **S-rep** if the occurrence count is lower than a pre-specified frequency, and only add a precondition to the **G-rep** if the occurrence count is higher than a pre-specified frequency. The frequency is essentially a function of the maximal level of noise the system will tolerate. The learning rate will likely be much slower, however, if there is noise in the domain. Some recent work along this direction [Oates and Cohen, 1996, Tae and Cook, 1996] has only been applied to simple domains.

8.3.2 Learning with unobservable features

Consider the set of predicates that describe the operators and states. Most predicates in the process planning domain, such as SHAPE-OF and HAS-CENTER-HOLE, are directly observable. But a subset of the predicates may be unobservable, e.g., HOLDING-WEAKLY. In this thesis, we assume that the learner can observe all these features; however, this assumption can be easily relaxed. We can provide the learning system with a set of observable and unobservable predicates. To learn under this circumstance, we assume that these unobservables can be determined by the system while it is executing its own plans (i.e., during practice). Consider HOLDING-WEAKLY as an example. Although the system cannot observe whether the expert is holding weakly or strongly, when it is applying the action itself, HOLDING-WEAKLY is observable. Therefore, the unobservables can be learned during the system's own plan execution. A similar example is the predicate HEAVY. The weight of an object is not determinable by remote observation, but is determinable by actual manipulation.

Having unobservable facts in the state of the other agent means that the observed state is a subset of the real state. Therefore, the operators we learn may miss some preconditions and effects that involve the unobservable facts. Assuming that these unobservable facts can be determined in the system's own state when it is executing plans, these missing preconditions and effects may be learned as follows:

- When `OBSERVER` executes an operator during practice, if some unobservable facts of the state are potentially present in the pre-state or delta-state, they can be added to the preconditions or effects of the operator. The relevance of the added preconditions will be determined in later practice and will be retained or disposed of accordingly

- When an operator fails to execute even though all its preconditions are satisfied, then the planner knows that it is missing some preconditions. The system can propose potentially unobservable facts as the preconditions of this operator, then generate plans to achieve them. If the operator is successfully applied in the environment after achieving a potentially relevant but unobservable precondition, that proposed precondition is added to the operator

8.3.3 Control knowledge learning

Another area of future research is the acquisition and maintenance of control knowledge in the context of changing domain knowledge. Control knowledge is used by planning systems to speed up planning [Korf, 1985, Sacerdoti, 1977, Mitchell, 1983, Laird *et al.*,

1986, Chien, 1989, Minton, 1988, Mostow and Bhatnagar, 1987], and to generate plans of higher quality [Iwamoto, 1994, Pérez, 1995]. Previous work on control knowledge learning presumes a correct model of operators; however, domain knowledge frequently changes in the process of knowledge acquisition as OBSERVER incrementally learns operators. Since control knowledge is dependent on the domain knowledge, the validity of previously acquired control knowledge is open to question. Can we learn control knowledge reliably while the domain representation is still being formed? What type of control knowledge learning is suitable for this situation?

In [Wang and Veloso, 1994], we investigated ways to apply previous work on derivational reuse of planning episodes [Veloso, 1994] in the presence of evolving domain knowledge. We show how learned control knowledge may still be useful in guiding a planner even when the domain theory is incrementally changing, if we allow for flexible reuse of the learned guidance. For instance, planning steps from past planning episodes may be skipped when the corresponding preconditions are no longer part of the domain and new planning steps may be added when new preconditions or effects change the course of planning.

Another possible approach is structure control knowledge as planning episodes of different levels of granularity, such as macros, and to flexibly reuse these acquired macros [Yang and Fisher, 1993]. Macro operators have the advantage that their validity can be justified solely based on the preconditions and effects of the operators. As long as the preconditions and effects of each operator in a macro operator are properly indexed, a macro operator can be easily updated and used as control knowledge when the operators in the domain change.

Appendix A

The PRODIGY Problem Solver

This Appendix is adapted from the Appendix in [Pérez, 1995].

PRODIGY is a domain-independent problem solver that serves as a testbed for planning and machine learning research. Given an initial state and a goal expression, PRODIGY searches for a sequence of operators that will transform the initial state into a state that matches the goal expression. The current version of PRODIGY, PRODIGY4.0, is a nonlinear and complete planner. It follows a means-ends analysis backward chaining search procedure reasoning about multiple goals and multiple alternative operators relevant to achieving the goals. Detailed descriptions of PRODIGY4.0 appear in [Carbonell *et al.*, 1992, Veloso *et al.*, 1995]. [Fink and Veloso, 1996] presented a formal description of the planning algorithm used in PRODIGY4.0.

PRODIGY4.0 provides a rich action representation language coupled with an expressive control language. A planning *domain* is defined by a set of types of objects, i.e., classes, used in the domain, and a library of operators and inference rules that act on these objects. Each operator is defined by its *preconditions* and *effects.* The description of preconditions and effects of an operator can contain typed variables. In addition variable bindings (i.e. values a variable can take) can be constrained by arbitrary Lisp functions. Preconditions in the operators can contain conjunctions, disjunctions, negations, and both existential and universal quantifiers. The *effects* of an operator consist of a list of predicates to be added or deleted from the state when the operator applies. An operator may also have *conditional* effects that are to be performed depending on particular state conditions. Inference rules deductively change a particular planning state by adding semantically redundant information to the state, in constrast to operators which specify *real* changes to the state. They have the same syntax as operators. PRODIGY4.0 allows two types of inference rules: *eager* inference rules fire automatically every time there is a change in the state whenever their preconditions are satisfied; they are used only in a

forward-chaining manner. *Lazy* inference rules are used for backward chaining; they only fire on demand and PRODIGY4.0 subgoals on their preconditions if they are not true. A truth-maintenance system (TMS) keeps track of all the inference rules (both eager and lazy) that are fired. When an operator is applied, the effects of inference rules whose preconditions are no longer true are undone.

A *planning problem* is defined by (1) a set of available objects of each type, (2) an *initial state* I, and (3) a *goal statement* G. The initial state is represented as a set of literals. The goal statement is a logical formula equivalent to a preconditions expression, i.e. it can contain typed variables, conjunctions, negations, disjunctions, and universal and existential quantifications. A solution to a planning problem is a sequence of operators that can be applied to the initial state, transforming it into a state that satisfies the goal.

Table A.1 describes the basic search cycle of PRODIGY4.0's nonlinear planner [Veloso, 1989]. This search algorithm involves several decision points, namely:

- Which goal to subgoal on, from the set of pending goals.
- Which operator to choose in order to achieve a given goal.
- Which bindings to choose in order to instantiate the selected operator.
- Whether to apply an applicable operator (and which one) or defer application and continue subgoaling.

Control knowledge may direct the choices in each of these decision points. In PRODIGY, there is a clear division between the declarative domain knowledge (operators and inference rules) and the more procedural control knowledge. This simplifies both the initial specification of a domain and the incremental learning of the control knowledge. Control knowledge can take the form of control rules (usually domain-dependent), complete problem solving episodes to be used by analogy [Veloso, 1994], domain-independent heuristics [Blythe and Veloso, 1992, Stone *et al.*, 1994], and control knowledge trees [Pérez, 1995].

Control rules are productions (*if-then* rules) that indicate which choices should be made (or avoided) depending on the current state and other meta-level information based on previous choices or subgoaling links. They can be hand-coded by the user or automatically learned. They are divided into these three groups: *select*, *reject*, and *prefer* rules. *Select* and *reject* rules are used to prune parts of the search space, while *prefer* rules determine the order of exploring the remaining parts. Alternatives pruned by select and reject control rules are not tried should the planner backtrack to the node where the rule fired. Control rules choose goals, operators, bindings, or subgoaling versus apply. They can also choose nodes to backtrack to.

1. Check if the goal statement is true in the current state, or there is a reason to suspend the current search path.

   If yes, then either return the final plan or backtrack.

2. Compute the *set* of *pending goals* $\mathcal{G}$, and the set of possible *applicable operators* $\mathcal{A}$.
   A pending goal is a precondition of an operator previously expanded (in Step 4) that is not true in the current state. An applicable operator is an operator whose preconditions are true in the state.
3. Choose a goal G from $\mathcal{G}$ or select an operator A from $\mathcal{A}$ that is directly applicable.
4. If G has been chosen, then
   - get the set $\mathcal{O}$ of *relevant operators* for the goal,
   - choose an operator O from $\mathcal{O}$,
   - get the set $\mathcal{B}$ of possible *bindings* for O,
   - choose a set B of bindings from $\mathcal{B}$,
   - go to step 1.
5. If an operator A has been selected as directly applicable, then
   - *apply* A,
   - go to step 1.

Figure A.1: A skeleton of PRODIGY4.0's nonlinear planning algorithm (adapted from [Veloso, 1989]). Problem solving decisions, namely selecting which goal/subgoal to address next, which operator to apply, what bindings to select for the operator, or where to backtrack in case of failure, can be guided by control knowledge. PRODIGY's trace provides all the information about the decisions made during problem solving so it can be exploited by machine learning methods.

PRODIGY is designed with a "glass-box" approach: all the decisions made by the search engine and all the information available to make those decisions are captured in a problem's trace. This provides an information context in which learning can take place. Figure A.2 shows the learning modules developed in PRODIGY, according to their learning goal, namely: learn control knowledge to improve the planner's *efficiency* in reaching a solution to a problem [Minton, 1988, Etzioni, 1990, Pérez and Etzioni, 1992, Knoblock, 1994, Veloso, 1994, Borrajo and Veloso, 1994b]; learn control knowledge to improve the *quality* of the solutions produced by the planner([Borrajo and Veloso, 1994b, Iwamoto, 1994, Pérez, 1995]); and learn *domain* knowledge, i.e., learn or refine the set of operators specifying the domain([Gil, 1992] and this thesis), or acquire them through a

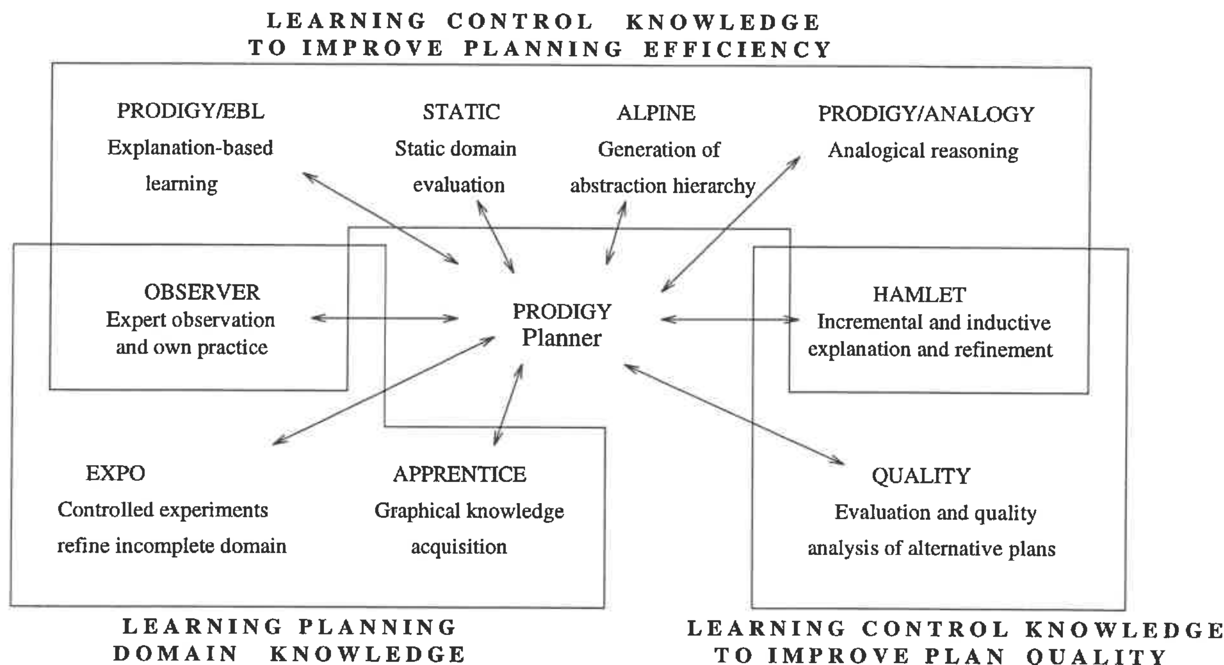


Figure A.2: The learning modules in the PRODIGY architecture (from [Veloso *et al.*, 1995]).

graphical apprentice-like dialog [Joseph, 1992].

The machine learning and knowledge acquisition work supports PRODIGY's casual-commitment method[1], as it assumes there is *intelligent* control knowledge, exterior to its search cycle, that it can rely upon to make decisions, both to make planning more efficient and to obtain good quality plans. PRODIGY has been applied to a wide range of planning and problem-solving tasks: robotic path planning [Haigh *et al.*, 1994], the blocksworld, several versions of the STRIPS domain, matrix algebra manipulation, discrete machine-shop planning [Gil and Pérez, 1994] and scheduling, computer configuration, logistics transportation planning, and several others. Other research in the PRODIGY project has focused in studying different planning techniques and heuristics [Blythe and Veloso, 1992, Veloso and Blythe, 1994, Stone *et al.*, 1994, Blythe, 1994].

[1]In a casual-commitment strategy at each decision point the planner commits to a particular alternative, and backtracks upon failure. This is in contrast to a least-commitment strategy where decisions are deferred until all possible interactions are recognized.

Appendix B

A Trace From Practice

Here is an example of how planning, learning, execution, and plan repair is integrated in OBSERVER in the process planning domain. The practice problem is illustrated in Figure B.1. Figures B.2, B.3, B.4, and B.5 illustrate some partially learned operators that are relevant to solving this practice problem. The **G-rep**, **S-rep**, and previously learned extraneous preconditions of these operators are shown.

In this example, OBSERVER first generates an initial plan to achieve the top-level goals of the problem. OBSERVER then starts executing the plan. The first operator in the plan, hold-with-vise, fails to execute in the environment. Since every precondition in the **S-rep** of the operator hold-with-vise is satisfied in the state, OBSERVER conjectures that (not (has-burrs <v3>)) is a negated precondition. OBSERVER then generates a plan segment (operators 3 and 4) to achieve this negated precondition. Note that during plan repair, the preconditions in the **G-rep** of the failed operator hold-with-vise (i.e., (is-clean part0)) must also be achieved. After plan repair, the plan now includes four operators: operators 3, 4, 1, and 2. OBSERVER successfully executes operators 3 and 4 (i.e., (remove-burrs part0) and (clean part0)). When OBSERVER successfully executes operator 1 (hold-with-vise), it adds the negated precondition (not (has-burrs part0)) to the **G-rep** of this operator, since this operator has previously failed to execute, and it is executed successfully now because the precondition (not (has-burrs <v3>)) is satisfied in the state. After OBSERVER successfully executes operator 2 (i.e., drill-with-spot-drill), the top-level goal is achieved and OBSERVER returns SUCCESS.

The trace for problem solving is as follows:

```
initial plan:
1. <hold-with-vise drill0 vise0 part0 side6 side1>
2. <drill-with-spot-drill
     part0 hole0 spot-drill0 vise0 vise0 drill0 0.75 0.5 side1>

plan = {1,2}

operator (1) executes unsuccessfully,
calling the learning module with negative example:
 #<Hold-with-vise [<v1> drill0] [<v2> vise0] [<v3> part0]
                  [<v4> side6] [<v5> side1] [<v6> ()]>

goal: (holding drill0 vise0 part0 side1)
goal-stack: ((has-spot part0 hole0 side1 0.5 0.75))
not-satisfied-lits-S-rep: nil
adding the following negated preconds to operator: #<op: hold-with-vise>
 (~ (has-burrs <v3>)))
plan repair for goals:
   (and (~ (has-burrs part0)) (on-table drill0 part0) (is-clean part0))

new plan segment:
3. <remove-burrs part0 vise0 drill0>
4. <clean part0 vise0 drill0>

plan = {3, 4, 1, 2}

operator (3) executes successfully,
deleting the following literals from the state:
  ((has-burrs part0) (is-clean part0))

calling the learning modifying with positive example:
 #<remove-burrs [<v19> part0] [<v25> vise0] [<v23> drill0]>

plan = {4, 1, 2}

operator (4) executes successfully,
adding the following literals to the state:
 ((is-clean part0))

calling the learning modifying with positive example:
 #<clean [<v28> part0] [<v34> vise0] [<v32> drill0]>
```

```
plan = {1, 2}

operator (1) executes successfully,
adding the following literals to the state:
  ((holding drill0 vise0 part0 side1))
deleting the following literals from the state:
  ((is-available-part part0)
   (is-empty-holding-device vise0 drill0)
   (on-table drill0 part0))

calling the learning modifying with positive example:
 #<hold-with-vise [<v1> drill0] [<v2> vise0] [<v3> part0]
                  [<v4> side6] [<v5> side1] [<v6> ()]>
updating the \grep\ of operator hold-with-vise by adding the following precondition:
  (~ (has-burrs <v3>)

plan = {2}

operator (2) executes successfully,
adding the following literals to the state:
 ((has-burrs part0) (has-spot part0 hole0 side1 0.5 0.75))
deleting the following literals from the state:
 ((is-clean part0))

calling the learning modifying with positive example:
 #<drill-with-spot-drill [<v137> part0] [<v136> hole0] [<v144> spot-drill0]
                         [[<v143> vise0] [<v142> vise0] [<v141> drill0]
                         [<v230> 0.75] [<v229> 0.5] [<v228> side1]>

TOP-LEVEL-GOALS-ACHIEVED
SUCCESS
```

```
(setf (current-problem)
  (create-problem
    (objects
     (drill0 drill)
     (spot-drill0 spot-drill)
     (vise0 vise)
     (mineral-oil mineral-oil)
     (part0 part)
     (hole0 hole))
    (state
     (and
       (holding-tool drill0 spot-drill0)
       (has-device drill0 vise0)
       (is-available-table drill0 vise0)
       (is-available-holding-device vise0)
       (material-of part0 brass)
       (hardness-of part0 soft)
       (size-of part0 length 4)
       (size-of part0 width 2.75)
       (size-of part0 height 2.25)
       (shape-of part0 rectangular)
       (on-table drill0 part0)
       (is-empty-holding-device vise0 drill0)
       (is-available-part part0)
       (is-clean part0)
       (has-burrs part0)))

    (goal (has-spot part0 hole0 side1 0.5 0.75))))
```

Figure B.1: Problem definition, which consists of the initial state and a goal description.

```
(operator hold-with-vise
(preconds ((<v1> machine)) (<v2> vise)
           (<v3> part) (<v4> side)
           (<v5> side))
(and (is-empty-holding-device <v2> <v1>)
     (has-device <v1> <v2>)
     (is-clean <v3>)
     (is-available-part <v3>)
     (on-table <v1> <v3>) )
(effects ((<v6> toe-clamp))
  ((if
    (and
     (is-empty-holding-device <v6> <v1>)
     (shape-of <v3> cylindrical)
   ((add (holding-weakly
            <v1> <v2> <v3> <v4>)))))
   (if (and (shape-of <v3> rectangular))
    ((add (holding <v1> <v2> <v3> <v5>))))
   (del (on-table <v1> <v3>))
   (del (is-available-part <v3>))
   (del (is-empty-holding-device
           <v2> <v1>))))
:G-rep
    ((on-table <v1> <v3>)
     (has-device <v1> <v2>)
     (is-clean <v3>))
:extraneous-preconds
    ((is-empty-holding-device <v6> <v1>)
     (is-available-tool <v8>)
     (is-available-tool <v9>)
     (is-available-tool <v11>)
     (holding-tool <v1> <v12>)
     (is-available-holding-device <v6>)
     (hardness-of <v3> <v10>)
     (shape-of <v3> <v417>)
     (is-available-table <v1> <v2>)
     (size-of-drill-bit <v9> <v13>)
     (size-of <v3> diameter <v14>)
     (size-of <v3> length <v15>)
     (material-of <v3> <v16>)
     (diameter-of-drill-bit <v8> <v17>)))
```

Figure B.2: Incomplete operator HOLD-WITH-VISE used to solve the practice problem.

```
(operator drill-with-spot-drill
 (preconds
  ((<v137> part) (<v136> hole)
   (<v144> spot-drill) (<v143> vise)
   (<v142> (or vise toe-clamp)) (<v141> drill)
   (<v230> number) (<v229> number)
   (<v228> side))
  (and
   (is-clean <v137>)
   (has-device <v141> <v142>)
   (has-device <v141> <v143>)
   (holding-tool <v141> <v144>)
   (holding <v141> <v142> <v137> <v228>)))
 (effects
  ((del (is-clean <v137>))
   (add (has-spot
          <v137> <v136> <v228> <v229> <v230>))
   (add (has-burrs <v137>))))
 :G-rep
 ((holding-tool <v141> <v144>)
  (holding <v141> <v142> <v137> <v228>)))
 :extraneous-preconds
 ((is-available-tool <v138>)
  (is-available-tool <v139>)
  (is-available-tool <v140>)
  (hardness-of <v137> <v1768>)
  (diameter-of-drill-bit <v139> <v424>)
  (size-of <v137> length <v231>)
  (material-of <v137> <v232>)
  (shape-of <v137> <v233>)
  (diameter-of-drill-bit <v138> <v235>)
  (diameter-of-drill-bit <v140> <v234>)
  (is-empty-holding-device <v143> <v141>)
  (size-of <v137> height 4.25)
  (size-of <v137> width 3))
```

Figure B.3: Incomplete operator DRILL-WITH-SPOT-DRILL used to solve the practice problem.

```
(operator clean
 (preconds ((<v28> part) (<v34> vise)
   (<v32> (or drill milling-machine)))
  (and
   (is-available-part <v28>)
   (is-empty-holding-device <v34> <v32>)))
 (effects
  ((add (is-clean <v28>))))
  :G-rep
   ((is-available-part <v28>))
  :extraneous-precondsープreconds
  ((is-empty-holding-device <v33> <v32>)
   (is-available-tool <v31>)
   (is-available-tool <v29>)
   (is-available-tool <v30>)
   (holding-tool <v32> <v35>)
   (has-device <v32> <v34>)
   (has-device <v32> <v33>)
   (shape-of <v28> <v189>)
   (size-of <v28> length <v190>)
   (size-of <v28> height 4.25)
   (size-of <v28> width <v113>)
   (material-of <v28> <v114>)
   (hardness-of <v28> <v116>)
   (diameter-of-drill-bit <v30> <v115>)
   (angle-of-drill-bit <v31> 80)
   (is-available-machine <v32>)))
```

Figure B.4: Incomplete operator `clean` used to solve the practice problem.

```
(operator remove-burrs
 (preconds ((<v19> part) (<v25> vise)
    (<v23> (or drill milling-machine)))
  (and
   (is-available-part <v19>)
   (is-empty-holding-device <v25> <v23>)
 (effects
  ((if (and (is-clean <v19>))
    ((del (is-clean <v19>))))
    (del (has-burrs <v19>))))
 :G-rep
  ((is-available-part <v19>))
 :extraneous-preconds
   ((is-empty-holding-device <v24> <v23>)
    (has-burrs <v19>)
    (is-available-tool <v22>)
    (is-available-tool <v20>)
    (is-available-tool <v21>)
    (has-device <v23> <v24>)
    (has-device <v23> <v25>)
    (holding-tool <v23> <v26>)
    (shape-of <v19> <v179>)
    (size-of <v19> length <v180>)
    (size-of <v19> height 4.25)
    (size-of <v19> width <v101>)
    (material-of <v19> <v102>)
    (hardness-of <v19> <v104>)
    (diameter-of-drill-bit <v21> <v103>)
    (angle-of-drill-bit <v22> 80)
    (is-available-machine <v23>))
```

Figure B.5: Incomplete operator `remove-burrs` used to solve the practice problem.

Bibliography

[Allen *et al.*, 1990] James Allen, James Hendler, and Austin Tate. *Readings in Planning.* Morgan Kaufmann Publishers, San Mateo, CA., 1990.

[Ambros-Ingerson and Steel, 1988] Jose' A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 1988.

[Anzai and Simon, 1979] Yuichiro Anzai and Herbert A. Simon. The theory of learning by doing. *Psychological Review*, 86:124–140, 1979.

[Beetz and McDermott, 1992] Michael Beetz and Drew McDermott. Declarative goals in reactive plans. In *Proceedings of the First International Conference on AI Planning Systems*, College Park, MD, 1992.

[Benson, 1995] Scott Benson. Inductive learning of reactive action models. In *Proceedings of Twelfth International Conference on Machine Learning*, Tahoe City, CA, July 1995.

[Bergadano and Giordana, 1988] F. Bergadano and A. Giordana. A knowledge intensive approach to concept induction. In *Proceedings of Fifth International Workshop on Machine Learning*, Ann Arbor, MI, June 1988.

[Blumer *et al.*, 1987] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, 1987.

[Blythe and Veloso, 1992] Jim Blythe and Manuela Veloso. An analysis of search techniques for a totally-ordered nonlinear planner. In *Proceedings of the First International Conference on AI Planning Systems*, College Park, MD, June 1992.

[Blythe, 1994] Jim Blythe. Planning with external events. In *Proceedings of the Conference on Uncertainty in AI*, Seattle, WA, 1994.

[Blythe, 1996] Jim Blythe. Decompositions of markov chains for reasoning about external change in planners. In *Proceedings of the Third International Conference on AI Planning Systems*, Edinborgh, Scotland, 1996.

[Boose and Gaines, 1989] John H. Boose and Brian R. Gaines. Knowledge acquisition for knowledge-based systems: Notes on the state-of-the-art. *Machine Learning*, 4:377–394, 1989.

[Borrajo and Veloso, 1994a] Daniel Borrajo and Manuela Veloso. Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning, ECML94*, Sicily, Italy, 1994. Springer Verlag.

[Borrajo and Veloso, 1994b] Daniel Borrajo and Manuela Veloso. Incremental learning of control knowledge for improvement of planning efficiency and plan quality. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, New Orleans, November 1994.

[Bundy *et al.*, 1985] Alan Bundy, Bernard Silver, and D. Plumer. An analytical comparison of some rule-learning programs. *Artificial Intelligence*, 27, 1985.

[Carbonell *et al.*, 1992] Jaime G. Carbonell, and the PRODIGY Research Group: Jim Blythe, Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn, Craig Knoblock, Steven Minton, Alicia Pérez, (editor), Scott Reilly, Manuela Veloso, and Xuemei Wang. PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, June 1992.

[Cassandra *et al.*, 1994] Anthony Cassandra, Leslie Kaelbling, and Michael Littman. Actin optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI94*, Seattle, WA, August 1994. AAAI Press/The MIT Press.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[Cheng and Carbonell, 1986] Patricia Cheng and Jaime Carbonell. The FERMI system: Inducing iterative macro-operators from experience. In *Proceedings of the National Conference on Artificial Intelligence*, pages 490–495, Philadelphia, PA, 1986.

[Chien *et al.*, 1995] Steve Chien, Randal W. Hill Jr., Xuemei Wang, Tara Estlin, Kristina Fayyad, and Helen Mortensen. Why real-world planning is difficult: A tale of two applications. In M. Ghallab, editor, *Advances in Planning*. IOS Press, 1995.

[Chien *et al.*, 1996a] Steve Chien, Tara Estlin, and Xuemei Wang. Hierarchical task network and operator-based planning: Competing or complementary? Technical Report JPL Technical Document D-13390, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 1996.

[Chien *et al.*, 1996b] Steve Chien, Xuemei Wang, Tara Estlin, and Anita Govindjee. Automatic generation of temporal dependency networks for antenna operations. *Telecommunications and Data Acquisition*, 1996.

[Chien, 1989] Steve Chien. Using and refining simplifications: Explanation-based learning of plans in intractable domains. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 590–595, Detroit, MI, 1989.

[Chien, 1994] Steve Chien. Towards an intelligent planning knowledge base development environment. In *AAAI-94 Fall Symposium Series: Planning and Learning: On to Real Applications*, New Orleans, LA, 1994.

[Chien, 1996] Steve Chien. Static and completion analysis for planning knowledge base development and verification. In *Proceedings of the Third International Conference on AI Planning Systems*, Edinborgh, Scotland, 1996.

[Christensen, 1990] Jens Christensen. A hierarchical planner that creates its own hierarchies. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1004–1009, Boston, MA, 1990.

[Cohen, 1991] William W. Cohen. Learning approximate control rules of high utility. In *Proceedings of Eighth International Workshop on Machine Learning*, Evanston, IL, June 1991.

[Davis, 1979] R. Davis. Interactive transfer of expertise: Acquisition of new inference rules. *Artificial Intelligence*, 12, 1979.

[desJardins, 1994] Marie desJardins. Knowledge development methods for planning systems. In *AAAI-94 Fall Symposium Series: Planning and Learning: On to Real Applications*, New Orleans, LA, 1994.

[Doyle, 1985] Lawrence E. Doyle. *Manufacturing Processes and Materials for Engineers*. Prentice-Hall, Englewood Cliffs, NJ, third edition, 1985.

[Draper *et al.*, 1994] Denise Draper, Steve Hanks, and Dan Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on AI Planning Systems*, Chicago, IL, 1994.

[Estlin and Mooney, 1996] Tara A. Estlin and Raymond J. Mooney. Multi-strategy learning of search control for partial-order planning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, August 1996. AAAI Press/The MIT Press.

[Estlin *et al.*, 1996] Tara Estlin, Xuemei Wang, Anita Govindjee, and Steve Chien. Dplan deep space network antenna operations planner version 1.0 programmer's manual. Technical Report JPL Technical Document D-13377, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 1996.

[Etzioni *et al.*, 1992] Oren Etzioni, Steve Hanks, Daniel Weld, Denis Draper, Neal Lesh, and Mike Williamson. An approach to plannin with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation an Reasoning*, pages 115–125, Cambridge, MA, 1992.

[Etzioni, 1990] Oren Etzioni. *A Structural Theory of Explanation-Based Learning.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 1990. Also appeared as Technical Report CMU-CS-90-185.

[Fikes and Nilsson, 1971] Richard E. 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.

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

[Fink and Veloso, 1996] Eugene Fink and Manuela Veloso. *Formalizing the Prodigy planning algorithm.* IOS press, Amsterdam, Netherlands, 1996.

[Firby, 1987] James R. Firby. An investigation into reactive plannin in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, WA, 1987.

[Genesereth and Nourbakhsh, 1993] Michael Genesereth and Illah Nourbakhsh. Time-saving tips for problem solving with incomplete information. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington D.C, 1993.

[Gil and Pérez, 1994] Yolanda Gil and M. Alicia Pérez. Applying a general-purpose planning and learning architecture to process planning. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, pages 48–52, New Orleans, November 1994.

[Gil and Tallis, 1995] Yolanda Gil and M. Tallis. Transaction-based knowledge acquisition: complex modifications mode easier. In *Proceedings of the Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1995.

[Gil, 1991] Yolanda Gil. A specification of process planning for PRODIGY. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1991.

[Gil, 1992] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1992.

[Gil, 1994] Yolanda Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of Eleventh International Conference on Machine Learning*, New Brunswick, NJ, July 1994.

[Haigh and Veloso, 1996] Karen Z. Haigh and Manuela M. Veloso. Interleavin planning and robot execution for asynchronous user requests. In *Working Notes of the AAAI 1996 Spring Symposium Series, Symposium on Planning with incomplete information for robot problems*, Stanford University, CA, March 1996.

[Haigh *et al.*, 1994] Karen Zita Haigh, Jonathan Richard Shewchuk, and Manuela Veloso. Route planning and learning from execution. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, pages 58–64, New Orleans, November 1994.

[Hammond, 1989] Kristian Hammond. *Case-Based Planning: Viewing planning as a memory task.* Academic press, New York, NY, 1989.

[Haussler, 1988] David Haussler. Quantifying inductive bias: AI learning algorithms and valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

[Haussler, 1989] David Haussler. Learning conjunctive concepts in structural domains. *Machine Learning*, 4:7–40, 1989.

[Hayes-Roth and McDermott, 1978] Frederick Hayes-Roth and John McDermott. An interference matching technique for inducing abstractions. In *CACM*, volume 26, pages 401–410, 1978.

[Hayes, 1990] Caroline C. Hayes. *Machining Planning: A Model of an Expert Level Planning Process.* PhD thesis, The Robotics Institute, Carnegie Mellon University, December 1990.

[Hill *et al.*, 1995] Randal Hill, Steve Chien, Kristina Fayyad, Crista Smyth, Trish Santos, and Richard Chen. Sequence of events driven automation of the deep space network. *Telecommunications and Data Acquisition*, 1995.

[Huffman *et al.*, 1993] Scott Huffman, Douglas Pearson, and John Laird. Correcting imperfect domain theories: A knowledge-level analysis. In Susan Chipman and Alan L. Meyrowitz, editors, *Foundations of Knowledge Acquisition: Congitive Models of Complex Learning*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1993.

[Hume and Sammut, 1991] David Hume and Claude Sammut. Using inverse resolution to learn relations from experiments. In *Proceedings of Eighth Machine Learning Workshop*, Evanston, IL, July 1991.

[Iwamoto, 1994] Masahiko Iwamoto. A planner with quality goal and its speedup learning for optimization problem. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 281–286, Chicago, IL, 1994.

[Joseph, 1992] Robert L. Joseph. *Graphical Knowledge Acquisition for Visually-Oriented Planning Domains*. PhD thesis, Carnegie Mellon University, School of Computer Science, August 1992. Also appeared as Technical Report CMU-CS-92-188.

[Kambhampati, 1990] Subbarao Kambhampati. A theory of plan modification. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[Katukam and Kambhampati, 1994] Suresh Katukam and Subbarao Kambhampati. Learning explanation-based search control rules for partial order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 582–587, Seattle, WA, July 1994. AAAI Press/The MIT Press.

[Knoblock, 1994] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68, 1994.

[Knoblock, 1995] Craig A. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, CA, 1995.

[Kodratoff and Tecuci, 1991] Yves Kodratoff and Gheorghe Tecuci. DISCIPLE-I: Interactive apprentice system in weak theory fields. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sydney, Australia, 1991.

[Korf, 1985] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[Kuokka, 1990] Daniel R. Kuokka. *The deliberative integration of planning, execution, and learning.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990.

[Kushmerick *et al.*, 1995] Nick Kushmerick, Steve Hanks, and Dan Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76, 1995.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Langley, 1985] Pat Langley. Learning to search: From weak methods to domain-specific heuristics. *Cognitive Science*, 9:217–260, 1985.

[McAllester and Rosenblitt, 1991] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

[Minton *et al.*, 1989] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-based Approach.* Kluwer Academic Publishers, Boston, MA, 1988. PhD thesis available as Technical Report CMU-CS-88-133, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Mitchell *et al.*, 1983] T. Mitchell, P. Utgoff, and R. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristic. In *Machine Learning, An Artificial Intelligence Approach, Volume I.* Tioga Press, Palo Alto, CA, 1983.

[Mitchell *et al.*, 1990] Tom M. Mitchell, Sridhar Mahadevan, and Louis I. Steinberg. LEAP: A learning apprentice system for VLSI design. In Yves Kodratoff and Ryszard Michalski, editors, *Machine Learning: An Artificial Intelligence Approach*, volume III, pages 271–289. Morgan Kaufmann, San Mateo, CA, 1990.

[Mitchell, 1978] Tom M. Mitchell. *Version Spaces: An Approach to Concept Learning.* PhD thesis, Stanford University, 1978.

[Mitchell, 1983] Tom M. Mitchell. Learning and problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI83*, volume 2, pages 1139–1151, Karlsruhe, Germany, 1983. Computers and Thought Lecture.

[Mostow and Bhatnagar, 1987] Jack Mostow and Neeraj Bhatnagar. Failsafe — a floor planner that uses EBG to learn from its failures. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milano, Italy, 1987.

[Muggleton and Feng, 1990] Stephen H. Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithm Learning Theory*, Tokyo, Japan, 1990.

[Muggleton, 1992] Stephen H. Muggleton. *Inductive logic programming.* Academic press, New York, NY, 1992.

[Muggleton, 1995] Stephen H. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[Nau, 1987] Dana S. Nau. Automated process planning using hierarchical abstraction. *Texas Instruments Technical Journal*, Winter:39–46, 1987.

[Neves, 1985] David Neves. Learning procedures from examples and by doing. In *Proceedings of IJCAI 85*, 1985.

[Oates and Cohen, 1996] Tim Oates and Paul R. Cohen. Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, August 1996. AAAI Press/The MIT Press.

[Pazzani and Kibler, 1992] Michael Pazzani and Dennis Kibler. The utility of background knowledge in inductive learning. *Machine Learning*, 9:57–94, 1992.

[Penberthy and Weld, 1992] J. Scott Penberthy and Dan Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of KR-92*, 1992.

[Pérez and Etzioni, 1992] M. Alicia Pérez and Oren Etzioni. DYNAMIC: A new role for training problems in EBL. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference, ML92*, pages 367–372. Morgan Kaufmann, San Mateo, CA., 1992.

[Pérez, 1995] M. Alicia Pérez. Learning search control knowledge to improve plan quality. Technical Report CMU-CS-95-175, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1995. PhD thesis.

[Plotkin, 1970] Gordon D. Plotkin. A note on inductive generalization. *Machine intelligence*, 5:153–163, 1970.

[Porter and Kibler, 1986] Bruce Porter and Dennis Kibler. Experimental goal regression: a method for learning problem-solving heuristics. *Machine Learning*, 1:249–284, 1986.

[Quinlan, 1986] Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Quinlan, 1990] Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Ruby and Kibler, 1990] David Ruby and Dennis Kibler. Learning steppingstones for problem solving. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 366–373, San Diego, CA, November 1990.

[Sacerdoti, 1977] Earl Sacerdoti. *A Structure for Plans and Behavior.* Elsevier, North Holland, New York, 1977.

[Sammut, 1996] Claude Sammut. Using background knowledge to build multistrategy learner. In *Proceedings of Third International Workshop on Multistrategy Learning*, Harpers Ferry, WV, 1996.

[Segre, 1988] Alberto Maria Segre. *Machine Learning of Robot Assembly Plans.* Kluwer Academic Publishers, Boston, MA, 1988.

[Shank, 1982] Roger C. Shank. *Dynamic Memory: a theory of reminding and learning in computers and people.* Cambridge University Press, Boston, MA, 1982.

[Shapiro, 1983] Ehud Y. Shapiro. *Algorithmic Program Debugging.* MIT press, Cambridge, MA, 1983.

[Shen, 1989] Wei-Min Shen. *Learning from Environment Based on Percepts and Actions.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1989.

[Shen, 1994] Wei-Min Shen. *Autonomous Learning from the Environment.* Computer Science Press, W.H. Freeman and Company, 1994.

[Simmons and Koenig, 1995] Reid Simmons and Sven Koenig. Probabilistic robot navigation in partically observable environments. In *Proceedings of the fourteenth International Joint Conference on Artificial Intelligence*, Montreal, CA, 1995.

[Simmons, 1988] Reid G. Simmons. A theory of debugging plans and interpretations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 94–99, St Paul, MN, 1988. Morgan Kaufmann.

[Smith and Rosenbloom, 1990] Benjamin Smith and Paul Rosenbloom. Incremental non-backtracking focusing: A polynomial bounded generalization algorithm for version space. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[Stone *et al.*, 1994] Peter Stone, Manuela Veloso, and Jim Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, pages 164–169, Chicago, IL, June 1994.

[Tadepalli, 1989] Prasad Tadepalli. Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 694–700, Detroit, MI, 1989.

[Tae and Cook, 1996] Kang Soo Tae and Diane J. Cook. Experimental knowledge-based acquisition for planning. In *Proceedings of Thirteenth International Conference on Machine Learning*, Bari, Italy, July 1996.

[Vanlehn, 1987] Kurt Vanlehn. Learning one subprocedure per lesson. *Artificial Intelligence*, 31(1):1–40, 1987.

[Veloso and Blythe, 1994] Manuela Veloso and Jim Blythe. Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, pages 170–175, Chicago, IL, June 1994.

[Veloso and Stone, 1995] Manuela Veloso and Peter Stone. Flecs: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research*, 3, 1995.

[Veloso *et al.*, 1995] Manuela M. Veloso, Jaime G. Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), January 1995.

[Veloso, 1989] Manuela M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.

[Veloso, 1994] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning.* Springer Verlag, Berlin, Germany, 1994. PhD thesis available as technical report CMU-CS-92-174, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Vere, 1980] Steven. A. Vere. Multilevel counterfactuals for generalizations of relational concepts and productions. *Artificial Intelligence*, 14:139–164, 1980.

[Wang and Veloso, 1994] Xuemei Wang and Manuela Veloso. Learning planning knowledge by observation and practice. In *ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative*, Tucson, Arizona, February 1994.

[Wang, 1995] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of Twelfth International Conference on Machine Learning*, Tahoe City, CA, July 1995.

[Wang, 1996] Xuemei Wang. Planning while learning operators. In *Proceedings of the Third International Conference on AI Planning Systems*, Edinborgh, Scotland, 1996.

[Wilkins, 1988] David Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, San Mateo, CA., 1988.

[Winston, 1975] Patrick. H. Winston. *Learning structural descriptions from examples.* McGraw-Hill, New York, 1975.

[Yang and Fisher, 1993] Hua Yang and Douglas Fisher. Planning speedup by learning, reusing, and patching macro operators. In *Proceedings of Third International Workshop on Knowledge Compilation and Speedup Learning*, Amherst, MA, 1993.