LEARNING ACTION MODELS FOR REACTIVE AUTONOMOUS AGENTS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Scott Sherwood Benson
December 6, 1996

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Nils J. Nilsson
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Yoav Shoham
Professor, Computer Science

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Barbara Hayes-Roth
Senior Research Scientist, Computer Science

Approved for the University Committee on Graduate Studies:

Abstract

To be maximally effective, autonomous agents such as robots must be able both to react appropriately in dynamic environments and to plan new courses of action in novel situations. Reliable planning requires accurate models of the effects of actions—models which are often more appropriately learned through experience than designed. This thesis describes TRAIL (Teleo-Reactive Agent with Inductive Learning), an integrated agent architecture which learns models of actions based on experiences in the environment. These action models are then used to create plans that combine both goal-directed and reactive behaviors.

Previous work on action-model learning has focused on domains that contain only deterministic, atomic action models that explicitly describe all changes that can occur in the environment. The thesis extends this previous work to cover domains that contain durative actions, continuous variables, nondeterministic action effects, and actions taken by other agents. Results have been demonstrated in several robot simulation environments and the Silicon Graphics, Inc. flight simulator.

The main emphasis in this thesis is on the action-model learning process within TRAIL. The agent begins the learning process by recording experiences in its environment either by observing a trainer or by executing a plan. Second, the agent identifies instances of action success or failure during these experiences using a new analysis demonstrating nine possible causes of action failure. Finally, a variant of the Inductive Logic Programming algorithm DINUS is used to induce action models based on the action instances. As the action models are learned, they can be used for constructing plans whose execution contributes to additional learning experiences. Diminishing reliance on the teacher signals successful convergence of the learning process.

To My Parents

Acknowledgements

This thesis is a product of the contributions of many people. First of all, Nils Nilsson was always there to support and supervise my research work during my years at Stanford, and constantly ready with intelligent and witty advice, comments, criticisms, and suggestions. The ideas in this thesis owe much to our many thought-provoking discussions. Nils has also been an inspiring example as a professor, both in sharing his enthusiasm for the art and science of AI and in his help with the details of preparing a new undergraduate course in the field.

The other members of the committee, Yoav Shoham and Barbara Hayes-Roth, have also contributed much to the shaping of this thesis with their comments and advice, particularly on the overall directions of the research. Although not on the committee, Pat Langley had a major influence on the thesis through long discussions and a ready supply of technical knowledge on all areas of machine learning.

I would also like to thank the various members of the Bots and Nobots research groups over the years, who listened to presentations of many stages of this research and were always there to provide helpful suggestions, blunt criticisms, and endless argumentation. In particular, I'd like to thank Marko Balabanovic, Lise Getoor, George John, Steve Ketchpel, Ronny Kohavi, Daphne Koller, Andrew Kosoresow, Ofer Matan, Illah Nourbakhsh, Karl Pfleger, and Mehran Sahami. There have also been many other colleagues both at Stanford and elsewhere who provided useful and interesting feedback on the research, among them Marie desJardins, Scott Huffman, Arjun Kapur, Jean-Claude Latombe, Seth Rogers, Marcel Schoppers, Walter Tackett, and Xue Mei Wang. I would also like to thank Jutta McCormick, Mina Madrigal, and Sara Merryman for their support and assistance during my time at Stanford.

A great debt of gratitude is owed to Devika Subramanian, whose Introduction to Artificial Intelligence course at Cornell University instilled in me a love of the subject and inspired me to continue on in the field. If it had not been for CS472 and the Lego Project, most likely none of this thesis would have happened.

Most of this thesis was completed while supported by the National Science Foundation under grants IRI-9116399 and IRI-9421839. My first few years at Stanford were supported by a fellowship from the National Defense Science and Engineering Graduate Fellowship Program.

Finally, my friends and family have always made the graduate student life a memorable and interesting one: Colleen, Susan, Steve, Sarah, Herb, Annegret, Raf, and all the rest of the Lutherans; Bert, Cyndy, Andrei, Doug, Steve, Debbie, Joel, Floyd, Rupert, Brian, and the other members and ex-members of the Stanford Bridge Club; and from afar, Tejas, Pat, Lisa, Laura, Ted, and Mark. Finally, Mom and Dad have always been there with their support, love, and understanding for a son far away from home.

Typographical Conventions

Throughout this thesis, we will be using the following typographical conventions to indicate the function of various items:

- Predicate names and constants will be in capitalized italics, e.g. $Holding(?x)$, $Article1$.
- Function names will be in lower-case italics, e.g. $copies(2, Article1)$. (Technically, constants are 0-ary functions but the distinction between predicate names and constants should always be clear.)
- The universally true predicate will be indicated by T, the universally false predicate by F.
- Action names will be in lower-case typewriter font, e.g. `turn`, `move-forward`.
- Indexical-functional variables (introduced in Chapter 5) will be in capitalized typewriter font, e.g. `Location-of-Robot`. (Technically, each capitalized word indicates the start of a new indexical-functional object.)
- Names of computer systems will be in sans serif font, e.g. STRIPS, LIVE.
- Variables are lower-case italic letters preceeded by a question mark, e.g. $?x, ?z$.
- World states will be indicated by a lower-case italic s_i. For any state s_i, the description of the state in predicate logic will be denoted by S_i. Finally, we will occasionally need to to refer to the state in predicate logic. Such references will be indicated by an italic upper-case ST_i.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we are interested in autonomous agents that learn from their environments. Autonomous agents are ubiquitous in the field of Artificial Intelligence, ranging from robots to intelligent characters for interactive entertainment (Bates 1992, Bates 1994, Hayes-Roth 1995) to various Softbots and web agents (Etzioni & Weld 1994, Jennings & Wooldridge 1996). "Autonomous agent" is a term that is difficult to define precisely, but in general we use the term to refer to a computer system that can act with only minimal human supervision. A human might give advice or instructions to such an agent, but would not be controlling its low-level interactions with the world. Thus, the agent must have some ability to act independently.

1.1 Autonomous Agents That Learn from Environments

An *environment* in general consists of that part of the world with which an agent must interact. As mentioned above, autonomous agents can exist in both physical environments, as do robots, and computational environments, as do software agents and virtual characters. An agent will generally have some way of sensing at least part of the state of its environment, and can take actions that affect its environment. An autonomous agent will also usually have *goals*, which correspond in some way to states of the environment that are "desirable" for the agent. If the agent is well-constructed, its behavior should be directed at achieving these goal states.

The behavior of an intelligent agent can come from two different sources: behaviors can be programmed directly into the agent, or the agent can learn goal-achieving behavior through experience in its environment over time. Both approaches have difficulties. Purely programmed behavior can be inflexible and costly to design, while learning can be unpredictable and can take many experiences to achieve competent behavior. This thesis presents one approach to designing an autonomous agent that can combine these two sources of behavior in what we believe are interesting and useful ways.

1.1.1 Controlling Autonomous Agents

Assume for the moment that we wish to program the behavior of an autonomous agent. There are a number of approaches we might take. As an initial attempt, we might simply give it a large table of condition-action (or *stimulus-response*) pairs that cover every possible sensory state. Thus, the agent perceives the world (the stimulus), determines which condition in the list holds given the perception, and takes the associated action (the response). If the goals of the agent change, it can simply begin executing a different table of condition-action pairs. This stimulus-response model of agent programming may seem simplistic, but when used cleverly, it can result in surprisingly effective behaviors, such as the locomotion behaviors of various robots from Brooks' research group (Brooks 1989*a*, Brooks 1989*b*, Flynn & Brooks 1988).

For many practical environments, however, it is either very difficult or simply too time-consuming to precompute a condition-action table that covers all the situations that may arise, especially if the domain tends to be unpredictable. If the agent has a model describing its environment and the effects of the actions it can take in the environment,[1] it can use various techniques such as dynamic programming

[1]Such a model can take a variety of forms. Environmental models are discussed further in Section 3.1.2

(Bellman 1962) or symbolic planning (Tate, Hendler & Drummond 1990) to determine appropriate behavior for a given goal. Since the environmental model is independent of the particular goal, this method can effectively handle situations where the agent has many different goals to accomplish. A good historical example of this method is found in Shakey the robot (Nilsson 1984), which was given a description of its environment in terms of STRIPS operators (Fikes & Nilsson 1971), and when given a goal, would construct a plan to achieve that goal, and carry out the plan in the real world.

The difficulty with this solution is that giving the agent a complete model of the environment is often as difficult, if not more so, than specifying a stimulus-response behavior table. While it was easy to describe Shakey's environment using discrete STRIPS operators, most domains are significantly more complicated. The knowledge acquisition bottleneck is a well-known problem in the field of expert systems, and is clearly a difficulty in world modeling as well. It is difficult to get human experts to produce models in forms corresponding to those used by automated planning systems, and even then there may be inaccuracies or omissions in the model. Therefore, an agent that hopes to be able to make use of a model in such environments will need some capability for learning and modifying its environmental model.

However, while we believe that learning will be an essential component of the behavior of an autonomous agent, we do not believe it should be the *only* component of such an agent. Learning complete behaviors in an arbitrary environment is a very difficult problem, as we will discuss further in Section 3.2. Therefore, we would like our agent to be able to make use of human knowledge as well as knowledge that it has learned independently. Even though a human programmer may not be able to write a complete control program for the agent, he or she may be able to provide either a partial condition-action table or a partial model of the environment. Our agent should be able to take advantage of either of these forms of assistance.

Regardless of the learning methods used by an autonomous agent, we can view its actions as being determined by some control structure. For instance, this control structure may be a piece of code, a stimulus-response table, a symbolic plan, a neural network, or any of a variety of other mechanisms. Given our discussion above, this control structure must be able to support the execution of human-written control programs, the creation of new control programs through planning based on human-written environmental models, and the learning of new control programs through experience in the environment.

In this thesis, we have chosen to use teleo-reactive trees, or TR trees (Nilsson 1994), as our agent control structure. TR trees have a number of advantages as an agent control mechanism. They are easy for humans to write, and their symbolic structure allows for easy creation and modification by planning and learning algorithms. In addition, they are reactive, allowing the agent to operate in continuous, dynamic, and noisy environments. TR trees are presented in more detail in Section 2.1.

1.1.2 Action Model Learning

Learning can be defined in many different ways (see Langley (1996) for some discussion on this) but from the point of view of an autonomous agent, learning can be defined as anything that allows the agent to improve its performance. There are various ways in which learning might be used for this purpose, which are examined in more detail in Chapter 3. This thesis is concerned with one particular form of learning, known as action-model learning. Rather than try to learn a set of TR trees directly from experience, our system attempts to learn models of the environment that the agent can then use to construct TR trees through well-known automated planning techniques. This allows for the transfer of learning from one task to another in a natural and effective manner, resulting in a highly versatile agent that can deal with a wide variety of tasks.

There is a variety of existing work on the subject of action-model learning, most of it done at Carnegie Mellon University within the past several years (Gil 1992, Shen 1989, Shen 1994, Wang & Carbonell 1994, Wang 1995*b*). However, all of this work has assumed a particular model of operator execution, which we call the STRIPS model (Fikes & Nilsson 1971). This model is appropriate for certain domains, but does not apply well to domains that contain continuous variables, noisy sensors, or other agents – in short, the sorts of domains that TR trees were originally designed to deal with.

This thesis presents an action-model learning system known as TRAIL (*T*eleo-*R*eactive *A*gent with *I*nductive *L*earning). Like other action-model learning systems, TRAIL operates by building models that describe the effects of actions in new domains, which can then be used by a backward-chaining planner to construct plans that achieve new goals. But in combination with the TR tree control structure, TRAIL can also deal with continuous state variables, non-atomic actions, unreliable sensors, and unpredictable events.

1.2 The Problem Domains

TRAIL has been experimentally tested in three different domains: a simulated construction domain known as Botworld, a simulated office delivery domain, and the Silicon Graphics Flight Simulator. Examples of TRAIL's behavior and results on its performance in each of the domains can be found in Chapter 6.

As we discuss below, each of the three domains provides a different set of difficulties for a learning system. However, the reader will note that none of these domains involves physical robots. This is primarily because the technology of mobile robotics is not really sufficiently advanced to allow us to test TRAIL in interesting ways. Most of the difficulties involved in implementing, say, an office delivery robot, are low-level difficulties in navigation and manipulation. We believe that the techniques used in TRAIL are more appropriate for higher-level tasks, which would be difficult if not impossible to test on a physical robot at present. It would certainly be possible to implement something like the office delivery domain on a simple mobile robot by assuming "virtual manipulation" of objects, but it does not seem that this would be any more meaningful than the simulator in evaluating the *high-level* performance of TRAIL.

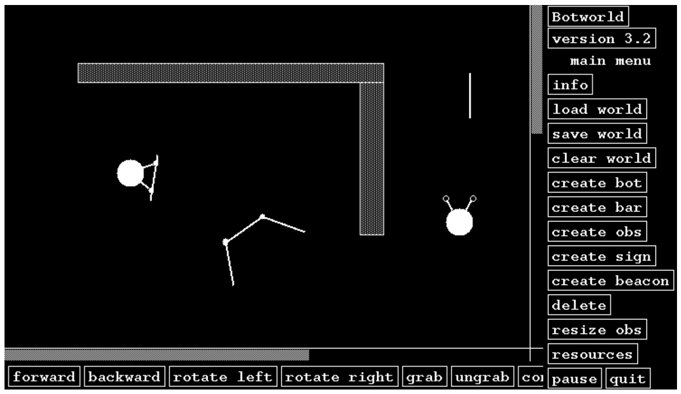

Figure 1.1: The Botworld Domain

1.2.1 Botworld

The first problem domain is a simulated construction domain, Botworld (Teo 1992). Botworld is a continuous two-dimensional space of robots (known as "bots"), obstacles, bars, and assemblies of bars. Each robot can pick up bars, move them around the environment, and connect them to other bars and structures. Robots can move over bars lying on the "floor" but cannot move through other robots or obstacles.

Each Botworld robot senses the environment at a rapid sampling rate, receiving information about its own position and orientation and whether or not it is holding a bar. It also senses the locations and orientations of nearby bars, obstacles, and robots. We can simulate imperfect sensors by adding noise to these measurements. A sample scene from Botworld is shown in Figure 1.1.

Unlike other simplified experimental domains such as grid worlds, actions in Botworld can be *durative* as well as discrete. A durative action is one that can continue indefinitely, such as `move`, as opposed to an atomic action such as `pick-up(Block1)` or `move-one-foot`. Botworld also has atomic actions, such as the bar-grabbing and welding operations, but the moving and turning operators are fundamentally durative.

A bot has the following perceptual predicates available to sense a particular bar (see Figure 1.2):

Holding(?b) is true iff the bot is holding bar ?b.

Free(?b) is true iff bar ?b is not being held by any bot (no more than one bot can

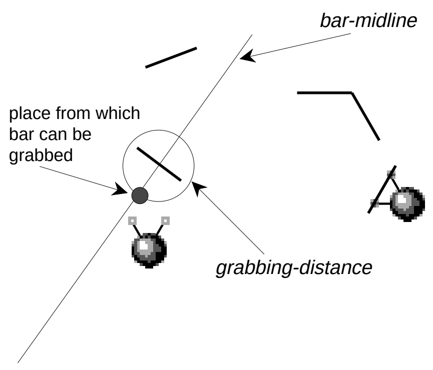

Figure 1.2: Terms Used in Botworld Predicates

be holding a given bar.)

FacingBar(?b) is true iff the bot is facing the center of the bar.

AtGrabbingDist(?b) is true iff the bot is at the appropriate distance from the center of the bar to grab it.

TooFar(?b) is true iff the bot is further from the center of the bar than the proper "grabbing distance".

OnMidline(?b) is true if the bot is on the "midline", an imaginary line going through the center of the bar and perpendicular to the bar.

FacingMidline(?b) is true if the bot is facing toward the midline rather than away from it.

ParallelTo(?b) is true if the bot is facing parallel to the long axis of the bar.

The bot also has the following set of actions it can take in the world:

`forward` moves the bot forward continuously.

`backward` moves the bot backward continuously.

`turn` rotates the bot to the left continuously. There is an analogous `right-turn` operator which is not used in this thesis.

`grab` is an atomic action that attempts to grab the bar one time. It either succeeds or fails almost immediately.

There are a number of other actions a bot can take, such as welding bars together and "speaking", but they are not used in any of our Botworld examples.

It is important to note that the bot's actions do not always have deterministic effects with respect to the predicates given above. For instance, the *FacingBar* predicate is actually defined in terms of the angle between the bot's heading direction and the center of some bar. As the bot moves forward towards the bar, this angle will increase, and thus may cause *FacingBar* to become false. As a result, if the bot moves forward while *FacingBar* is true, either the bot will reach the grabbing distance while *FacingBar* remains true, or *FacingBar* will become false before the bot reaches the grabbing distance. Of course, we could have designed the *FacingBar* predicate so that this did not occur, but in many domains it will not be possible to define predicates such that all action effects are deterministic. Therefore, our learner needs to be able to at least deal with this simple form of nondeterminism.

A standard benchmark problem in Botworld is the task of picking up a bar from any given starting position. A bar can only be grabbed if it is free, and the bot is on the midline, facing the bar, and at the correct distance. (The appropriate positions are shown in Figure 1.2.) In order to get to such a position, the bot must move to the midline, turn toward the center of the bar, and move forward or backward until it gets to the correct distance. A typical program to accomplish the bar-grabbing task can be found in Section 7.2.

The task of bar-grabbing is clearly a relatively simple one. However, it does involve durative actions and sometimes unpredictable action effects. Therefore, Botworld served as a useful domain for developing our system and testing out learning ideas.

1.2.2 The Delivery Domain

The second problem domain is a fairly standard one in the planning literature. It involves a simulated robot in an office environment of interconnected rooms inhabited by various people. Tasks for the robot include delivering messages to people, fetching objects for people, and making copies of articles and delivering then to people. Low-level navigation routines are built into the robot, so the robot only needs to do planning at higher levels.

A typical task in the delivery domain might be "deliver two copies of Article1 to George". The plan for this task consists of seven steps: go to the library, ask the librarian for Article1, go to the copy room, set the copier mode correctly, make two copies, go to George's office, and deliver the copies.

Obviously, the delivery domain is a fairly simple testbed for an autonomous agent. For planning and learning purposes, all of the delivery actions are discrete and deterministic, and none of the planning problems are particularly difficult. However, the delivery domain is illustrative of TRAIL's ability to transfer knowledge across tasks. Unlike Botworld, the delivery domain allows for a wide variety of goals, requiring TRAIL to transfer knowledge from one type of problem to another. Also, it shows that the TRAIL system, originally designed to deal with durative actions and continuous domains, can deal with discrete environments and actions as well. Finally, unlike the Botworld domain, which can essentially be described propositionally, the world states in the delivery domain are fundamentally "structured" in that they can only be described economically using first-order logic. We will examine this issue in further detail in Section 3.3.

1.2.3 The Flight Simulator

The third problem domain, and certainly the most interesting, is the Silicon Graphics, Inc. flight simulator.[2] Through a socket-based interface, TRAIL can communicate with the simulator, passing control commands for an aircraft and receiving periodic updates on the value of a number of state variables. Possible tasks within the environment include taking off, maintaining level flight, navigation, and landing. (The simulator also allows dogfight-style combat with other aircraft, which we did not implement in TRAIL.)

[2]We gratefully acknowledge the help of Seth Rogers in providing code and assistance with the simulator interface.

The main mechanism by which TRAIL controls the aircraft is by setting the position of the "stick" that controls the elevators and ailerons. (This value corresponds to the position of the mouse when a human is controlling the simulator.) Moving the elevators up or down changes the plane's pitch, while moving the ailerons left or right causes the plane to begin to roll in that direction, eventually causing it to turn. In addition, TRAIL can also increase or decrease the throttle setting, raise or lower the flaps, or apply the brakes (when the aircraft is on the runway.)

The effect of the vertical position of the stick on the plane's pitch is in general highly non-linear – a series of small changes will have no detectable effect, until the setting reaches a certain level, at which point the pitch of the aircraft will begin to change rapidly. TRAIL was not designed to learn behaviors at this low level, so we have given it a set of PID-based controllers (Bollinger & Duffie 1988) that will approximately keep the plane's climb rate at a particular level. If, for instance, the plane is climbing too quickly, the controller will lower the elevators until the nose of the plane begins to go quickly down, at which point the controller will recenter the elevators to stabilize the plane. But, during this process, the pitch of the plane has been reduced significantly. Thus each PID-based controller provides a durative action, such as "climb at approximately 200 feet per minute", that can be executed over an indefinite period of time until some goal has been achieved. (The controllers are a bit more complicated than a simple PID system, as our original pure PID controller could not recover from the quick changes in climb rate that were caused by the non-linear nature of the simulator control.)

In order to control the aircraft, TRAIL also needs to have input from the environment. Whenever a state variable changes significantly in the environment, the simulator sends an update message to TRAIL. (We can define what variables TRAIL is updated on, and how much change is "significant" – we do not want TRAIL to be updated every time the altitude changes by a foot, for instance.) At present, TRAIL makes use of 12 state variables: altitude, speed, throttle setting, flaps setting, x and y position, heading, roll, climb rate, and the first derivatives of heading (closely related to roll value), roll, and climb rate (related to pitch and speed.)

The flight simulator is a challenging environment in which to test an autonomous agent. Of course, the flight-related tasks are both more complicated and less artificial than the Botworld and delivery tasks. The flight simulator is also a challenging domain for learning, as it includes many continuous features, actions that are basically durative rather than discrete, and state variables that change value constantly during action execution. Furthermore, small timing delays in the interface between TRAIL and the simulator introduce an element of randomness into the simulation, ensuring that no two runs will be exactly alike. Experiences with the development of TRAIL in the flight simulator led to several of the important features of TRAIL, in particular tolerances on attribute values and the interval handling mechanism, discussed in Chapter 5.

The behavior of TRAIL in the flight simulator domain, as well as some of the difficulties that it encountered in this domain, are recounted in Sections 6.3 and 6.7 of the thesis.

1.3 Overview

This thesis presents TRAIL (*T*eleo-*R*eactive *A*gent with *I*nductive *L*earning), an architecture for an autonomous agent that learns to behave independently in new environments. TRAIL operates by building models that describe the effects of actions in new domains, which can then be used to construct plans that achieve new goals. TRAIL can deal with complex world state descriptions (as in the delivery domain), both continuous and discrete state variables (as in the flight simulator), and both durative and atomic actions (as in both Botworld and the flight simulator).

As described earlier, an essential component of the TRAIL architecture is the TR tree control structure (Nilsson 1994). Chapter 2 describes the teleo-reactive tree formalism, presents an action representation known as *teleo-operators* suitable for describing actions in TR trees, and discusses methods for planning TR trees using teleo-operators.[3]

[3]The more general issues involved in building an agent using teleo-reactive trees are covered in a paper by Benson and Nilsson (1995). Topics covered include hierarchical trees, different methods for node selection in tree execution, and arbitration among multiple competing goals.

Chapter 3 discusses a number of ways in which learning can be applied to the problems faced by an autonomous agent, and describes where the action-model learning methods used by TRAIL fit into the categorization. It then discusses the important features of an environment that must be considered in designing a system for action-model learning such as TRAIL. Finally, it presents the overall architecture of TRAIL, illustrating the relation between planning, learning, and TR execution.

Chapter 4 breaks down the action-model learning problem into two distinct phases: example generation and concept learning. It then covers the example generation phase of TRAIL's learning, explaining how TRAIL learns action models by observing the teacher, how it updates incorrect action models from action failures, how it replans on-line when action models are corrected, and how it conducts simple experiments to distinguish among cases of action failure.

Chapter 5 gives a brief introduction to the field of Inductive Logic Programming, and then describes how TRAIL uses ILP to learn the preconditions for action models. TRAIL's learning algorithm is based on a re-examination of the Inductive Logic Programming problem using the concept of indexical-functional variables(Agre & Chapman 1987, Schoppers & Shu 1990). The chapter concludes by discussing the methods that TRAIL uses to include intervals in induced conditions when necessary.

Chapter 6 begins by presenting an overview of the behavior of the complete TRAIL system, and illustrates the TRAIL learning system through a series of extended examples in Botworld and the Flight Simulator. It then presents experimental results from the TRAIL system in the three domains described above. We also discuss methodologies for evaluation of autonomous learning systems, and examines some of the issues related to TRAIL's performance in the three domains.

The thesis concludes with a summary and a discussion of possible avenues for future research.

1.4 Principal Contributions

Following is a list of the main scientific contributions of this thesis:

- *A new model of actions appropriate to reactive agents in continuous domains.* Teleo-reactive trees aid agents in dealing effectively with continuous and dynamic environments. The use of the teleo-operator formalism (presented in Section 2.3) allows us to describe the durative actions that are useful in such domains. In addition, teleo-operators free the learner from attempting to predict the effect of an action on the complete world state. Since most features of an environment are irrelevant to any particular task, such prediction can be expensive and unnecessary.

- *An implemented action-model learner that operates in non* STRIPS*-like environments.* Other existing action-model learning systems assume discrete world states, atomic actions, deterministic action effects, and noise-free state descriptions. Section 3.4 presents the assumptions made by TRAIL about its environment and contrasts those assumptions with those made by other action-model learning systems.

- *A thorough analysis of the possible causes of action failure for durative actions.* In general, a STRIPS-like atomic action either succeeds or has no effect in the environment. If the action is durative, the action can fail in a number of different ways. Chapter 4 analyzes the different kinds of action failure for durative actions and the possible causes of each. This failure analysis is a significant component of TRAIL's action-model learning.

- *The first successful use of Inductive Logic Programming in action-model learning.* The machine learning paradigm of Inductive Logic Programming, or ILP, is a natural candidate for action-model learning. ILP is intended to learn concepts from examples with more internal structure than the attribute-value examples used in most concept learning work. Since the descriptions of world states that arise during action-model learning have such internal structure, ILP should be applicable to these learning problems. In addition, there is considerable existing work on noise-handling mechanisms for ILP learners, while the learning algorithms used in most action-model learners (e.g. Shen's CDL algorithm (Shen 1994)) do not have any noise-handling provisions. The use of ILP in TRAIL is covered in Chapter 5.

Prior to TRAIL, no action-model learning system has made use of ILP. Sablon and Bruynooghe (1994) proposed using ILP for action-model learning within their event calculus formalism. However, they have done little work on the subject beyond their proposal (Sablon 1994). Wang tried applying a FOIL-like algorithm in her action-model learning system OBSERVER (Wang 1995*b*) but found that it did not work well. This is apparently due to certain interactions between FOIL and the OBSERVER learning mechanism, as OBSERVER has a difficult time recovering from certain types of induction errors that FOIL may make during learning (Wang 1995*a*).

- *A reconstruction of the ILP system* DINUS *(Džeroski, Muggleton & Russell 1992) from the point of view of indexical-functional variables.* We can view the determinate variables (Muggleton & Feng 1990) found in a state representation as indexical-functional quantities (Agre & Chapman 1987, Schoppers & Shu 1990). This allows us to assign a natural semantics to the induced concepts, in a manner that we have not previously seen. This view is presented in more detail in Section 5.4.

- *A new way of measuring the learning performance of an autonomous agent.* It is difficult to evaluate the performance of an integrated architecture for an autonomous agent. Evaluations of the individual components do not necessarily provide an accurate evaluation of the entire system, and the variety of goals and approaches used in autonomous agent learning has prevented the development of a standardized set of high-level benchmark tasks. In Section 6.4 we discuss these issues in more detail, and present a performance metric that is appropriate for TRAIL and other systems that learn in part by observing a teacher.

- *An integration of planning, learning, and execution within a unified framework.* The TR tree control structure naturally allows for reactive execution, automated generation and modification of plans, and model acquisition through learning. TR trees were introduced by Nilsson (1994), but this thesis is the first work to demonstrate their suitability for automated planning and learning.

Chapter 2

Action Representation in TRAIL

As described in Section 1.1.1, TRAIL makes use of a control structure known as Teleo-Reactive trees, or TR trees. Virtually all of TRAIL's behavior arises from the execution of TR trees, and TRAIL's learning is directed towards learning action models that can be used in the creation of new TR trees. This chapter discusses the TR tree control structure, the methods by which TRAIL generates new TR trees, and the representation of actions which TRAIL uses in order to reason about TR trees.

2.1 Teleo-Reactive Trees

A *teleo-reactive (TR)* program is an agent control program that directs the agent toward a goal in a manner that continuously takes into account changing environmental circumstances. Teleo-reactive programs were introduced in two papers by Nilsson (Nilsson 1992, Nilsson 1994). In its simplest form, a TR program consists of an ordered list of production rules:

$$\begin{aligned} K_1 &\rightarrow \mathtt{a_1} \\ &\cdots \\ K_i &\rightarrow \mathtt{a_i} \\ &\cdots \\ K_m &\rightarrow \mathtt{a_m} \end{aligned}$$

The K_i are conditions (on perceptual inputs and on a stored model of the world), and the $\texttt{a}_\texttt{i}$ are actions (on the world or that change the model). In typical usage, the condition K_1 is a goal condition, which is what the program is designed to achieve, and the action $\texttt{a}_\texttt{1}$ is the null action. In a general TR program, the conditions K_i may have free variables that are bound when the TR program is called to achieve a particular ground instance of K_1. These bindings are then applied to all the free variables in the other conditions and actions in the program. Actions in a TR program may be primitive or may themselves be TR programs. Thus, recursive TR programs are possible, although they do not play a part in this thesis.

A TR program is interpreted in a manner roughly similar to the way in which ordered production systems are interpreted: the list of rules is scanned from the top for the first rule whose condition part is satisfied, and the corresponding action is then executed. A TR program is designed so that for each rule $K_i \rightarrow \texttt{a}_\texttt{i}$, K_i is the regression, through action $\texttt{a}_\texttt{i}$, of some particular condition higher in the list. That is, K_i is the weakest condition such that the execution of action $\texttt{a}_\texttt{i}$ under ordinary circumstances will achieve some particular condition, say K_j, higher in the list (with $j < i$). We assume that the set of conditions K_i covers most of the situations that might arise in the course of achieving the goal K_1. (Note that we do not require that the tree be a universal plan, i.e. cover all possible situations.) Therefore, if an action fails, due to an execution error or the interference of some outside agent, the program will nevertheless continue working toward the goal in an efficient way. We will discuss this issue further in Section 2.1.2.

TR programs differ substantively from conventional production systems, however, in that their actions can be *durative* rather than discrete. A durative action is one that can continue indefinitely. For example, a mobile robot is capable of executing the durative action `move`, which propels the robot ahead (say at constant speed) indefinitely. Such an action contrasts with a discrete one, such as `move forward one meter`. In a TR program, a durative action continues so long as its corresponding condition remains the highest true condition in the list. When the highest true condition changes, the action changes correspondingly. Thus, unlike ordinary production systems, the conditions must be *continuously* evaluated; the action associated with the *currently* highest true condition is always the one being executed. An action terminates only when its associated condition ceases to be the highest true condition. The regression condition for TR programs must therefore be rephrased for durative actions: For each rule $K_i \rightarrow \texttt{a}_\texttt{i}$, K_i is the weakest condition such that continuous execution of the action $\texttt{a}_\texttt{i}$ (under ordinary circumstances) eventually achieves some particular condition, say K_j, with $j < i$. (The fact that K_i is the *weakest* such condition implies that, under ordinary circumstances, it remains true until K_j is achieved.)

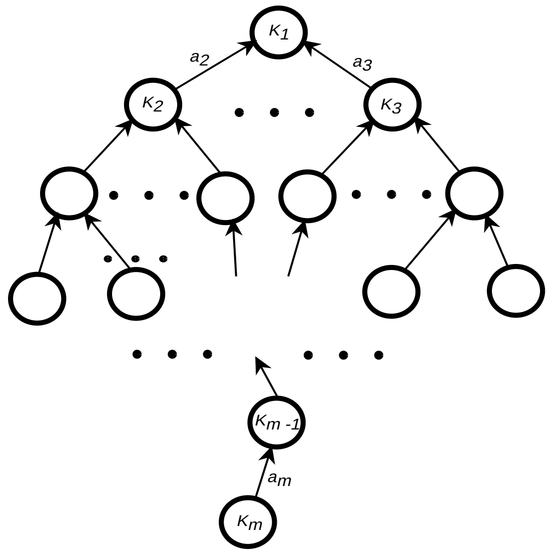

Figure 2.1: A TR Tree

In our work, we have found it convenient to represent a TR program as a tree, called a *TR tree*, as shown in Figure 2.1. Suppose two rules in a TR program are $K_i \rightarrow \texttt{a}_\texttt{i}$ and $K_j \rightarrow \texttt{a}_\texttt{j}$ with $j < i$ and with K_i the regression of K_j through action $\texttt{a}_\texttt{i}$. Then we have nodes in the TR tree corresponding to K_i and K_j and an arc labeled by $\texttt{a}_\texttt{i}$ directed from K_i to K_j. That is, when K_i is the shallowest true node in the tree, execution of its corresponding action, $\texttt{a}_\texttt{i}$, should achieve K_j. The root node is labeled with the goal condition and is called the *goal node*. When two or more nodes have the same parent, there are correspondingly two or more ways in which to achieve the parent's condition. Continuous execution of a TR tree would be achieved by a continuous computation of the shallowest true node and execution of its corresponding action.[1] We call the shallowest true node in a TR tree the *active node.*

In the hierarchy of agent control, we have found TR programs to be most appropriate for what might be called *mid-level* control. Consider the implementation of a control program for a mobile robot. Many of these control programs are designed using three layers or levels of control. At the lowest level, classical control theory is required for the feedback control of motors and other effectors. Since there is less demand for continuous feedback at the highest levels, conventional program control structures suffice there. A typical example of such a three-layered architecture is the SSS architecture of Connell (1992). In the SSS architecture, the top (*S*ymbolic) layer does overall goal setting and sequencing, the middle (*S*ubsumption) layer selects specific actions, and the lowest (*S*ervo) layer exerts feedback control over the effectors. While SSS does not use TR trees, one could imagine using TR trees for the action selection level of such an architecture.

The TR formalism is related to a number of other "circuit-based" agent control methods such as the subsumption architecture (Brooks 1986), universal plans (Schoppers 1987), and situated automata (Kaelbling & Rosenschein 1990). Comparisons with these architectures are discussed in (Nilsson 1994). The TR formalism is advantageous because, as we shall see, it is more readily incorporated in an architecture that accommodates planning and learning.

2.1.1 Simulating Continuous Execution

In thinking about the semantics of TR programs, it is important to imagine that the conditions, K_i, and all of their parameters, are being *continuously* computed. However, in computational implementations of TR programs, we compute the conditions (and the parameters upon which they depend) at discrete time steps, and then execute small increments of durative actions. A sufficiently high sampling rate is chosen—depending on the domain—to approximate continuous computation and

[1]Ties among equally shallow true nodes can be broken by some arbitrary but fixed tie-breaking rule.

execution.

This sort of approximation to continuous computation opens the possibility that a condition might be achieved between time steps without being noticed. For instance, if a condition in a TR program is satisfied if and only if the agent is facing within 0.1 degrees of a specific desired direction, the agent might turn sufficiently fast that this condition would hold only *between* two sampling points—leading the agent to turn past the desired direction rather than switching to the appropriate next action. There are at least two obvious solutions to this problem. First, the condition can be relaxed so that at least one sampling point must fall within it, given the agent's execution speed and sampling rate. Second, some measure of the time needed to satisfy the expected next condition can be used to insure that the agent slows down as that condition is approached. This solution has the effect of decreasing the distance between sampling points when necessary so that again, at least one sampling point will necessarily fall within the condition region. Both methods have been implemented successfully for various agent tasks, although the first method occasionally results in activation conditions so weak that the recommended action is no longer always appropriate.

Approximating continuous computation by searching the entire TR tree at each time step becomes impractical for sufficiently large TR programs. Therefore, we have developed a heuristic method of action selection that usually produces the same result and runs in constant time in nearly all cases regardless of the size of the tree.

In this heuristic method, the agent remembers which node K_i (with associated action $\mathtt{a_i}$) was active during the previous time step. It expects that in the absence of surprises, K_i will remain active until its parent node K_j (with associated action $\mathtt{a_j}$) becomes active. Therefore, so long as K_i remains true while K_j remains false, $\mathtt{a_i}$ is selected as the default action. When and if K_j becomes true, $\mathtt{a_j}$ will be selected as the default action. Only when neither K_i nor K_j holds will the agent fail to have a default action to fall back on.[2]

[2]Note that only in this case can we not guarantee constant time execution. Since this case arises only due to execution failures, it seems reasonable then to expect a reaction delay. Naturally, we would be forced to provide explicit error-handling routines to handle those execution failures in which such reaction delays could not be tolerated.

This default action computation is supplemented by a separate process that scans through the rest of the tree examining a few nodes on each cycle (only nodes higher than the default node need be examined.) If this separate process ever finds a true node that is higher than the default node, this higher node will be selected as the active node and the new default. In summary, the agent will execute a normal sequence of goal-achieving actions (so long as they have their expected effects) while searching for serendipitous situations with whatever extra time it has.

2.1.2 Theoretical Properties of TR Trees

If actions can be guaranteed to have their expected effects, we can guarantee that if any node in a TR tree is active, the tree will eventually achieve its intended effect. The fixed tie-breaking rule that is used for selection of the active node imposes a total ordering on the nodes in the tree. Once a node N_i has become active, it will remain active until some node with a higher priority becomes true. Since executing the action `ai` is guaranteed to make the node's parent condition become true eventually, some higher priority node will eventually become active. Since the set of nodes is finite, eventually the highest priority node, which is the root node, will become active, and thus the goal must be achieved.

As a side note, observe that the above argument does not hold in the absence of a fixed tie-breaking rule among nodes at the same level. Suppose that there are two nodes N_i and N_j at the same level of the tree such that the action associated with node N_i is `turn-left` and the action associated with N_j is `turn-right`. If the activating conditions on both nodes are satisfied, unless a fixed tie-breaking rule is used, nothing prevents the executer from repeatedly selecting N_i and N_j on alternate time steps, leaving the agent turning back and forth while never achieving the goal.

In the more realistic case where we cannot guarantee that all actions have their expected effect, we can still guarantee that the goal will be achieved with probability 1 if we make three necessary assumptions:

- The success of each node execution must be an independent event with nonzero probability. (Note that this implies that multiple executions of the same node in the tree must be independent events.) We say that the execution of a node N in a tree is successful if and only if some higher node eventually becomes true, and N remains active in the meantime. If the execution is unsuccessful, we refer to this as a node failure.

- If a node fails, it does so by eventually becoming false, rather than simply remaining true and having no effect.

- If a node fails, at least one node in the tree must still be true.

If all of the above conditions hold, then for each node N_i in the tree, if N_i is the current active node, there is some nonzero probability $p(N_i)$ that the goal will be achieved without any node failures occurring. Let p_{min} be the lowest such probability for any node in the tree. Therefore, for any active node, either the goal will be achieved without a failure with probability at least p_{min}, or an action failure will occur leaving the agent in some other node from which the goal can be achieved with probability at least p_{min}. This sequence of repeated attempts forms a series of independent events, each with probability at least p_{min} of success. Therefore, with probability 1 the goal will eventually be achieved.

It is important to note that the above three assumptions usually do not hold in real domains, particularly the assumption that node failures are independent events. If a node failure is due to the action of some other agent, it probably cannot be accurately modeled as a probabilistic event. Even if other agents are not involved, there may be some unknown but persistent condition in the environment that makes action failures more likely. Thus, once one failure has occurred, further failures are more likely. In general, we cannot use the assumption of independent node failures to model most real environments. Therefore, we cannot guarantee the success of most TR trees, beyond the claim that the actions work "under ordinary circumstances." However, it is important to note that this difficulty is not unique to TR trees – any control program given the same perceptual apparatus and the same environment will have similar uncertainties in execution.

2.1.3 Nonlinear Plans and Conjunctive Tree Nodes

We have extended the basic TR tree formalism to deal more efficiently with certain kinds of conjunctive conditions. Suppose that a condition K in a TR node is a conjunction of subconditions: $K \equiv C_1 \wedge C_2 \wedge \cdots \wedge C_m$. Suppose also that there are TR programs for achieving each of these subconditions such that once one of the subconditions, say C_i, is satisfied, the achievement of any other of them (by their corresponding TR programs) will not cause C_i to become false. That is, the TR programs for achieving the subconditions can be executed in any order. In this case, we can label the TR node with condition K as a *conjunctive node*, which has m independently achievable successor nodes in the tree, each labeled by one of the conjuncts, C_i. (By convention, these successor nodes are called AND nodes, and we do not label the arcs between the AND nodes and their parent conjunctive node.) Each AND node is the root of a TR subtree that achieves its condition C_i without interacting with the other conditions in K, and thus the TR subtrees can be executed in whatever order circumstances dictate.

A TR tree containing AND nodes can be executed by the usual TR execution mechanism: we execute that action corresponding to the shallowest true node, not counting the AND nodes since they have no action labels on the arcs exiting them. Note that due to the properties of teleo-reactive execution, once we begin executing a node within a subtree rooted at an AND node, execution of that subtree will continue (barring unforeseen occurrences) until the condition on the root AND node becomes true. Therefore, in constructing a conjunctive node, we do not need to worry about interactions between the actions of nodes in subtrees rooted at different children. We only need to insure that the actions within a given subtree do not interfere with the root condition C_i of any other subtree of the conjunctive node.

The use of conjunctive nodes allows us to considerably simplify our TR trees in certain circumstances. Suppose a node condition K has m subconditions, and that a plan for each subcondition contains n nodes. Further suppose that these m subconditions can be achieved in arbitrary order. Then a conjunctive node for K will have m subtrees, each of size n, for a total of approximately mn nodes. Without the use of conjunctive nodes, we would need to either arbitrarily impose an ordering on the m subgoals, or create subtrees for each of the $m!$ possible orderings of the subgoals, producing a tree of size $m!mn$.

An example of a TR tree with AND nodes is shown in Figure 2.2. This tree will cause a Botworld robot to place two bars at specific locations and orientations in the world and then return to a "home base." Since the acts of placing the two bars can successfully be done in either order, the program for positioning each bar becomes an independent subtree. (Note that the goal node itself in Figure 2.2 cannot be split into non-interacting conjuncts, as the act of placing each bar would interfere with the condition $At(50\,50)$.)

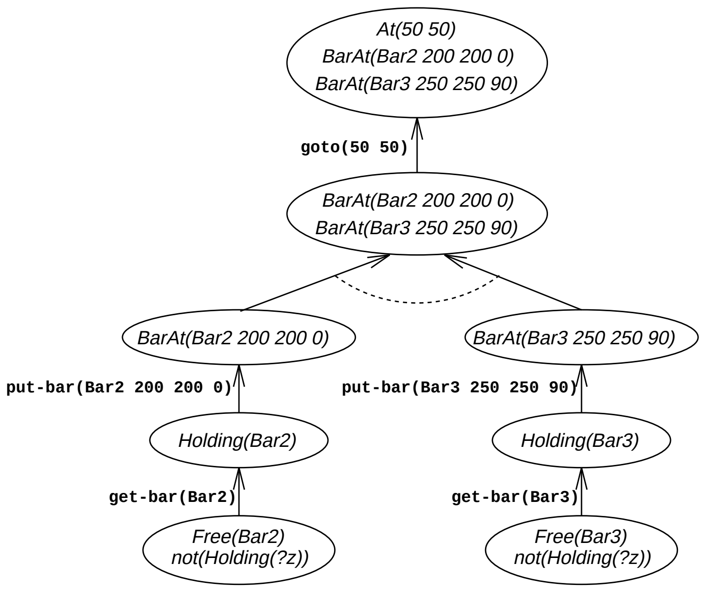

Figure 2.2: A TR Tree With Conjunctive Nodes

2.2 Planning and Teleo-Reactive Trees

We have considerable experience writing teleo-reactive programs to control various agents, including the robots of Botworld, the Silicon Graphics Flight Simulator, and an actual Nomad 100 robot (Galles 1993). However, each teleo-reactive program we construct covers only one (possibly parameterized) goal and (usually) some subset of the possible situations that might arise. It is difficult for the designer of an agent to anticipate in advance all the possible situations and goals that the agent might find itself confronted with. Therefore, we would like our agent to be able to construct its own teleo-reactive programs in response to new circumstances in which it finds itself; this is one of the reasons for selecting TR programs as a control mechanism.

2.2.1 Automated Planning

The automated construction of control programs is the subject of a large branch of artificial intelligence known as automated planning (for a good overview, see Tate et al. (1990).) Given a current state s_i and a goal state s_g, the purpose of an AI planning system is to produce a control program, or *plan*, that is expected to transforms s_i into s_g. In order to do this, the planner is given a set of *operators*, also known as action models, each of which describes how an action transforms the world state. Normally, a planner does some form of search (Korf 1988) to discover a sequence of operators that achieves the goal state.

Often, search in an AI planner is done through *backward chaining*, in which the planner begins at the goal state s_g and works backwards, looking for states from which s_g can be achieved. If the planner has a description of an action a, it can compute the *regression* of the goal state description S_g through a. The regression R of S_g through a is defined as the weakest condition that must hold to ensure that taking action a will achieve S_g. In other words, once R is true, the planner knows how to achieve S_g. This process is recursively repeated until a condition is found that holds in s_i, at which point the planner has found a way to achieve the goal state s_g from s_i.

TR programs closely resemble the search trees constructed by such backward-chaining AI planning systems. The overall goal corresponds to the root of the tree; the condition K_i on any non-root node is the regression of the condition K_j on its parent node, through the action, $\mathtt{a_i}$, connecting them. This similarity has allowed us to develop an automatic planning system that regresses conditions through durative actions to build a search tree. The search tree is then converted in a straightforward manner to a TR program.

2.2.2 Action Models in STRIPS and TRAIL

The action descriptions used by AI planning systems have commonly been represented using a formalism known as STRIPS operators, first used in the planning system STRIPS (Fikes & Nilsson 1971). STRIPS operators compactly describe the change from one world state to another when an action is taken in the world. For our purposes, the exact syntax of STRIPS operators is less important than the assumptions implicit in their use.

In the STRIPS operator model, each operator describes a primitive, atomic unit of action that is executed at a particular point in time, and either has a particular desired effect on the world, or has no effect at all. This model comes to us directly from the situation calculus (McCarthy & Hayes 1970), the predecessor of the STRIPS action representation. In order to describe an action in the situation calculus, we have a state object ST_i representing the state s_i of the world at one particular instant in time, an action a that is executed in the state, and a function that represents the following state of the world, designated $result(\mathsf{a}, ST_i)$. If a certain set of *preconditions* held in the state represented by ST_i, then the desired condition would hold in $result(\mathsf{a}, ST_i)$, else $result(\mathsf{a}, ST_i)$ would be equal to ST_i.[3] STRIPS operators introduced a much simpler way of reasoning about such world changes, but actions themselves are still considered to be instantaneous changes in the state.

Many real actions do fit this model, of course. Turning on a light switch, for instance, is easiest to model as a discrete, instantaneous action that either has an effect or does not. However, there are also many actions that do not fit the model very well. Consider the act of guiding an airplane down to land on a runway. This has one main action, pointing the nose of the aircraft downwards, and one main effect, the aircraft reaching the ground. However, the actual action occurs over some period of time, during which many other relevant variables are changing (speed, altitude, etc.) Also, the controller needs to be alert to other aspects of the environment during the action, perhaps changing the setting of the flaps if the approach speed is inappropriate

[3]The use of conditional effects could make this mapping more complex but the principle remains the same.

or even circling back around for another try if the end of the runway is approaching too quickly.

Therefore, in order to reason about actions for an autonomous, reactive agent we need to use a representation that views actions as continuous processes during which facts about the world are constantly changing. Reasoning about continuous action processes is quite complicated in general, but as a start, TRAIL uses operators which are partitioned according to action-effect pairs rather than according to the actions themselves. An operator in TRAIL, instead of describing a particular action, describes the use of an action to achieve a particular effect. Thus, the action may be executed for variable amounts of time, and may cause a variety of effects other than the one described by the operator. These representations of actions are known as *teleo-operators*, and are discussed in detail in the following section.

Of course, TRAIL is not the only system that uses a non-STRIPS representation of actions. Recent research in the area of temporal reasoning (Shoham & Goyal 1988) has investigated a variety of representations for actions that are richer than the STRIPS model, but little of this work has yet been applicable to control programs for autonomous agents. In particular, it has not generally been applied to the problem of action-model learning. DeJong's work (DeJong 1994) is a notable exception, which will be discussed further in Section 4.5.4.

2.3 Teleo-Operators

As stated above, TRAIL's operators are defined in terms of action-effect pairs rather than actions themselves. For any literal λ_i of interest in the environment, and any durative action $\mathsf{a_j}$, consider whether the execution of action $\mathsf{a_j}$ might ever cause λ_i to become true. If so, then we can define a *T*eleo-*OP*erator, or *TOP*, for the $(\mathsf{a_j}, \lambda_i)$ pair. This TOP describes the *process* of executing the action $\mathsf{a_j}$ until λ_i becomes true. Therefore a TOP is, in effect, a conversion of the durative action $\mathsf{a_j}$ into an appropriate atomic unit.[4] When a plan calls for the execution of an action corresponding to a

[4]A durative action can also be converted into atomic units by simply selecting a fixed time of execution, producing "actions" such as "Move forward for five seconds." The planning task is much

TOP, it will be executed as long as necessary to achieve its intended effect.

TOPs can be used to model the effects of atomic actions as well, as the execution of the action will then simply occur for a fixed time rather than until some condition becomes true. In this case, a teleo-operator will look very similar to the STRIPS operator that would describe the same action.

2.3.1 The Teleo-Operator Formalism

Throughout the remainder of this thesis, we will be using the convention $\xrightarrow{\mathtt{a_j}}\lambda_i$ to designate the TOP that models the use of action $\mathtt{a_j}$ to achieve condition λ_i. The arrow under the action name $\mathtt{a_j}$ indicates that $\mathtt{a_j}$ is to be executed for some period of time, after which λ_i should become true. A teleo-operator has four main components, as shown in Table 2.1.

1. the name of the action, $\mathtt{a_j}$.
2. the *postcondition* literal, λ_i, representing the intended effect of the TOP.
3. the *preimage condition*, π_{ij}, of the TOP, corresponding roughly to a STRIPS precondition. The exact semantics of the preimage condition are described beginning in Section 2.3.2.
4. the set of *delete list elements*, D_{ij}, of the TOP, corresponding roughly to a STRIPS delete list with likelihood statistics.

Table 2.1: The Components of a TOP

Teleo-operators are in fact *operator schemas*, meaning that they may contain variables and describe a class of operators rather than only one specific operator. The TOP action may of course contain variables; for instance, `pickup(?x)` describes the action of picking up an arbitrary object $?x$. The postcondition literal may contain the same variables, and may also contain variables that are not found in the action name. For instance, the TOP for turning to face a bar in Botworld has action `turn`

more difficult when given descriptions of such actions than it is when given TOPs.

and postcondition $FacingBar(?x)$, indicating that the TOP will work for any binding of the variable $?x$ (assuming that the preimage condition is satisfied, of course.)

We call the set of variables that are found in the postcondition and action the *TOP variables.* As in STRIPS, each of these variables can be thought of as universally quantified - for any binding of the variables, the TOP describes the process of using the (bound) action to achieve the bound condition. At present, the TR execution mechanism is not designed to determine bindings for variables in node actions. Thus, if the action of the TOP contains variables not found in the postcondition, the planner must assign bindings to these variables during planning.

The preimage condition of the TOP is normally written as a schema of predicate calculus literals. These literals may contain TOP variables, of course, and may also contain new variables. The TOP variables will be bound when the TOP is used in a plan, but the non-TOP variables need to be assigned a semantics. In TRAIL, unbound variables contained in a TOP preimage are considered to be existentially quantified if they are found in any unnegated literal, and universally quantified if they are found only in negated literals.

As an example, if the preimage condition of a `pickup(?x)` TOP is $At(Robot, ?y) \wedge At(?x, ?y) \wedge \neg Holding(?z)$, the TOP is applicable to a particular object $?x$ if there exists some $?y$ such that $At(Robot, ?y)$ and $At(?x, ?y)$ both hold, and for all values of $?z$, $Holding(?z)$ does not hold. This interpretation is considerably more intuitive than the alternative interpretation of $\neg Holding(?z)$, that there exists at least one $?z$ such that $Holding(?z)$ does not hold. Obviously any reasonable environment would satisfy the latter interpretation.

The rest of this chapter examines the semantics of the preimage condition and the delete list in more detail, and describes the use of TOPs in planning.

2.3.2 Teleo-Operators in Deterministic Environments

Let us assume for the moment that the effects of actions in an agent's environment are entirely deterministic. Therefore, for any given state of the world in which a TOP action can be taken, either the action eventually achieves the TOP postcondition or it does not. The purpose of the TOP preimage condition is to describe the subset of the possible world states in which the TOP's action will have its intended effect.

In order to reason about the effects of actions on the world state, we visualize the set of possible world states as a *state space*, or a multi-dimensional space in which each possible world state is a point in the multi-dimensional space. The idea of a state space is analogous to the idea of a configuration space in robot motion planning, with the modification that dimensions can correspond to any fact about the world that is relevant to the agent, not just physical positions and orientations. (State spaces are often extremely high-dimensional, but this does not generally present a computational problem, as we are not interested in reasoning about the geometry of the entire state space.) Since actions change the world state, an action can be viewed as a transition within state space. In particular, the execution of a durative action will form a path through the state space, with each point on the path representing a state of the world during the durative action.

Following work on directional preimages in robot motion planning (Lozano-Pérez, Mason & Taylor 1984, Christiansen 1991) we define the *preimage region*, of a literal, λ_i, with respect to a durative action, $\mathtt{a_j}$, as the set of world states in which continuous execution of $\mathtt{a_j}$ will eventually satisfy λ_i. The preimage region is a subset of state space from which the agent can reach the goal region, much like a preimage in robotics is a subset of physical space from which the robot can reach the goal region.

For a TOP $\xrightarrow{\mathtt{a_j}}\lambda_i$, we define the *preimage condition* π_{ij} as the weakest formula such that:

Effect rule: In any state in which π_{ij} holds, continuous execution of $\mathtt{a_j}$ will eventually satisfy λ_i.

Closure rule: In any state in which π_{ij} holds, if $\mathtt{a_j}$ is executed, π_{ij} will remain true until λ_i does becomes true.

Intuitively, the preimage condition corresponds to the precondition in a STRIPS-like operator, and can be used in much the same way to compute the regression of a goal through a TOP. The planning process using teleo-operators is covered in greater detail in Section 2.4.

In order to reason about states of the world, states are generally described as conjunctions of ground predicate calculus literals. There is actually a many-to-many relation between states of the world and these state descriptions – some state descriptions describe impossible states of the world, while some distinct states of the world may be indistinguishable in the state description language. Thus, our descriptions of states do not correspond perfectly to the actual state of the world.

Ideally, the preimage condition of a TOP corresponds exactly to the preimage region of that TOP, but as in the case with state descriptions, the vocabulary of our description language for preimage conditions may not be expressive enough to describe the preimage region exactly. Therefore, our preimage condition may in fact describe a subset of the actual preimage region. As an example of this, consider the diagram shown in Figure 2.3. The diagram shows part of a two-dimensional state space. The dark region corresponds to the area where some literal λ holds. The shaded region shows the preimage region of that literal with respect to some action `a`. However, suppose our description language only allows concepts to be described as axis-parallel rectangles in the given state space. In that case, the region described by the preimage condition will correspond to only a subset of the true preimage region. One possible such subset is shown in the figure by the dark rectangle.

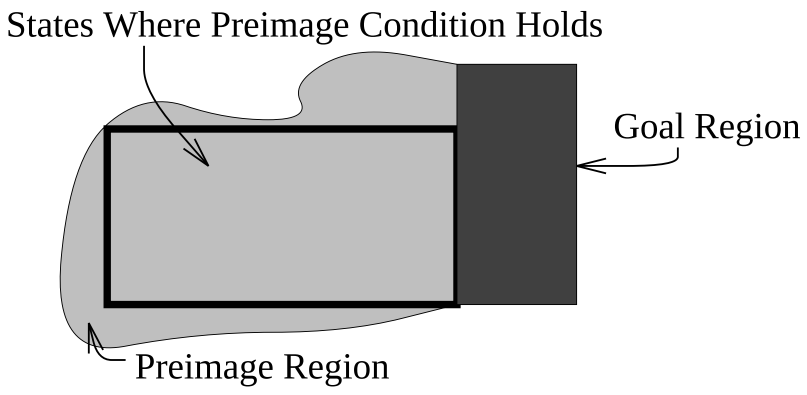

Figure 2.3: A Preimage Region and Preimage Condition

The closure rule in the definition of the preimage condition arises from this fact. Note that the actual preimage region can be said to be "closed" under execution of action `aj`. If the agent is in some state from which `aj` should eventually achieve λ_i, the state in the next instant will either be one in which λ_i holds or one in which, again, `aj` will eventually achieve λ_i. Therefore, execution of `aj` can never carry the agent outside the preimage region before λ_i has become true. (Remember that, for the moment, we are assuming that effects are deterministic and that there are no execution errors.) This corresponds very naturally to the execution mechanism for TR trees presented in Section 2.1; once a node N_i becomes active, it should remain active until its parent node becomes true. This is guaranteed to happen if and only if the closure rule holds for the condition and action of the node N_i. Therefore, we require that preimage conditions of TOPs have this same closure property, that π_{ij} must remain true throughout execution until λ_i becomes true. (Note that there is no way of telling from the diagram in Figure 2.3 whether the preimage condition does satisfy the closure rule.)

2.3.3 Teleo-Operators in Non-Deterministic Environments

The definitions in the previous section all assume that the world is entirely deterministic, and thus any state of the world is either inside or outside the preimage region. However, if the effects of actions are nondeterministic, it may no longer be possible to classify states as either definitely inside or outside the preimage region. There may be many states in which the action has some probability p ($0 < p < 1$) of achieving the TOP postcondition. One could then define the preimage region as the set of states where the probability of achieving λ_i is at least some threshold p_{min}. However, this definition does not provide a satisfying counterpart to the closure rule. More importantly, it is not a very practical definition. In most of the environments we are dealing with, we will not have a complete environmental model, so we may not know the exact probability that an action will have a particular effect in a particular state. Furthermore, the planner does not actually need action models with an exact probabilistic semantics.

Our solution to the nondeterminism problem is the same one that we adopted earlier when defining teleo-reactive programs. Recall the definition of a condition in a TR tree node: "the weakest condition such that execution of the action associated with the arc *under ordinary circumstances* achieves the condition associated with the parent." We did not define "ordinary circumstances" any more precisely, leaving the interpretation of this condition up to the programmer. We adopt the same convention for teleo-operators in non-deterministic environments: the preimage condition is the weakest condition such that execution of $\mathtt{a_j}$ under ordinary circumstances will achieve λ_i while maintaining π_{ij} until λ_i becomes true. A human who is designing teleo-operators is free to write operators that do not work 100% of the time, while an automated learning system that is learning teleo-operators (as we will discuss in a Section 2.5) can learn imperfect TOPs. Therefore, we also associate with each TOP a "success rate" that estimates the probability that λ_i will be achieved given that π_{ij} holds and the TOP is executed.

2.3.4 Side Effects of Teleo-Operators

When we conceptualize operators in terms of action-effect pairs rather than in terms of atomic action executions, we are focusing on one particular desired effect of the action. However, actions will usually have other effects as well. If an agent is using teleo-operators to plan for conjunctive goals, it will need to consider these other effects. Suppose the agent is planning to achieve some condition $\lambda \wedge \phi$ using a teleo-operator $\xrightarrow{\mathtt{a}}\lambda$. There are at least three possible circumstances:

- $\mathtt{a}$ may not affect condition ϕ.
- $\mathtt{a}$ may cause ϕ as well as λ to become true
- $\mathtt{a}$ may cause ϕ to become false before λ becomes true

In the first case, the planner can create a plan to achieve ϕ and then use the TOP to achieve λ as well. In the second case, the planner can simply use the TOP to achieve both λ and ϕ. In the third case, the planner will either need to achieve λ first or use some other TOP to achieve λ. In general, in order to compute the regression of $\lambda \wedge \phi$ through a TOP, the planner will need to take into account the possible effects of the TOP on ϕ.

Therefore, the teleo-operator model includes a "delete list" of conditions which may become false during execution. A condition ϕ is included in the delete list of a

TOP $\xrightarrow{\mathsf{a}}\lambda$ if there is some state s in its preimage region such that ϕ is true in s but ϕ is false at some point along the path (through state space) taken while a is executed, between s and the first state in which λ holds.

Note that ϕ is still in the delete list even if it could be guaranteed that ϕ would become true again by the time λ becomes true. This is due to the fact that the teleo-reactive tree built from the TOP is being continuously interpreted during execution. Suppose the plan segment shown in Figure 2.4 is generated. If node N_i is active and the agent begins executing a, the plan will only work if we are guaranteed that ϕ will remain true continuously until $\lambda \wedge \phi$ becomes true. If ϕ may become false during the execution of a, N_i will no longer be the active node and the agent will take some other action. Therefore, we require that ϕ must be on the delete list unless it cannot become false while a is being executed from the preimage region of $\xrightarrow{\mathsf{a}}\lambda$.

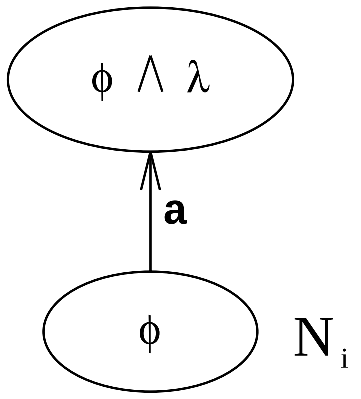

Figure 2.4: A TR Node With a Conjunctive Goal Condition

The deletion of conditions during execution will, in general, not be deterministic. It may be that the effects of actions themselves are nondeterministic, or it may be that the condition will become false only if the agent begins in a certain part of the preimage region. Therefore, it is desirable to keep track of the frequency of these effects for use in planning; a delete list effect that is very unlikely may not prevent the agent from constructing a plan, as long as the agent does not need to have a plan that is guaranteed to be correct. Thus, when TRAIL's learner constructs teleo-operators, it records the likelihood of each of the delete list elements, for later use in planning. Further details on the learning process are found in Chapter 5.

Note that in fact it may be the case that the delete list effects are conditional, i.e. they only occur when some other conditions are true in the environment. However, learning such dependencies greatly increases the complexity of learning, as the learner would need to learn "preimage" regions for each possible side effect. Rather than move further in the direction of building a complete model of the environment, we have chosen for the moment to keep side effect statistics based on the cases actually seen during learning. This issue is discussed somewhat further in the Future Work section of Chapter 7.

2.3.5 Semantics of Delete List Elements

Suppose that we are regressing a condition $\lambda \wedge \phi$ through a TOP $\xrightarrow{\mathsf{a}}\lambda^*$, where ϕ is a conjunction of one or more literals and λ^* is a (possibly) non-ground literal that unifies with λ. The unification of λ and λ^* will produce a binding for the TOP variables. Given this binding, the planner needs to be able to determine which literals in ϕ match the delete list elements of the TOP. If a delete list element μ has probability p, any literal in ϕ that matches μ will reduce the probability of success of the action by a factor of p.

For condition-matching purposes, there are four different categories of delete list elements:

- **Literals without arguments**. For instance, picking up a bar makes the proposition $NotHolding$ become false. Literals in ϕ can be matched with these conditions directly.

- **Literals containing only TOP variables**. For instance, executing the action `pickup(?x)` makes the condition $Free(?x)$ become false. Given a binding for the TOP variables, all the variables in the delete list element will be bound, so literals in ϕ can again be matched directly with the bound delete list elements.

- **Positive literals containing unbound variables**. For instance, a `goto` action would cause the robot to no longer be at its previous location. Therefore, we would include the condition $At(Robot, ?y)$ as a delete-list element of the TOP

$\xrightarrow{\texttt{goto(?x)}} At(Robot, ?x)$. The intended interpretation of such a delete-list element is that for any condition in the environment that matches the element, there is a probability p that it will become false during execution of the TOP. Literals in ϕ can be matched with these delete list elements by an obvious variant of the standard unification algorithm.

- **Negated literals containing unbound variables**. We interpret such literals as meaning that if there is a universally quantified negated literal in ϕ, such as $\neg Holding(?z)$, the presence of the same negated unbound literal in the delete list means that this universal condition may become false during execution of the TOP. For instance, if we apply the TOP $\xrightarrow{\texttt{grab}} Holding(?x)$ with $?x$ bound to $Bar1$, the TOP should delete the general condition $\neg Holding(?z)$ as well as the more specific condition $\neg Holding(Bar1)$. (Note that these negative delete list elements could ordinarily be computed by simply examining the non-negated effects of the TOP; however, as described in the next section, TRAIL does not keep detailed statistics on other non-negated TOP effects.)

  There is also a possible alternative interpretation of a negated delete list element such as $\neg Holding(?z)$: for any specific binding of some variable $?y$, it expresses the chance that $\neg Holding(?y)$ becomes false. However, the semantics of this interpretation are not well defined. Consider a delete list element such as $\neg Facing(?z)$. If we assume that there are an infinite number of objects in the world, and that any particular `turn` operation only causes the robot to be facing a finite number of objects (perhaps $Facing$ can only hold for nearby objects), then the probability p that `turn` will cause $Facing(?y)$ to become true for some arbitrary value of $?y$ is zero. Even if we make some assumption about the number of possible bindings of $?y$, such an assumption is essentially arbitrary and not likely to be very meaningful. Therefore we adopt the interpretation from the previous paragraph.

2.3.6 Add-List Elements

In STRIPS, the Add List contains literals that are expected to become true as a result of executing the operator. Since TOPs are defined with respect to action-effect pairs, there is one particular effect that is designated as the desired effect of the TOP, but there of course may also be other conditions that become true during execution.

These conditions are roughly analogous to the delete list conditions that were discussed in the previous sections. However, there is an important difference in the way they could be used. Delete list elements are used entirely to prevent incorrect regressions in which the TOP causes some important side condition to become false before the main effect occurs. Add list elements could be used this way as well, insuring that negated literals are correctly regressed through TOPs. However, the correct regression of negated literals is already handled by mechanisms discussed in the previous section. Therefore, it is not necessary to use them in this way.

Instead, add list elements can sometimes be employed to make the regressed conditions simpler. If a condition $\lambda \wedge \phi$ is being regressed through a TOP $\xrightarrow{\mathtt{a}}\lambda$, and one of the side effects of the TOP is to make ϕ as well as λ true, then the regression should actually be T rather than ϕ. If we can find side effects with sufficiently high probabilities, this can simplify the planning task somewhat.

However, such regression has an additional complexity due to the TR interpretation mechanism. If the side effect ϕ becomes true only when action a causes λ to change from false to true, then executing the TOP will only cause ϕ to become true if λ is false to begin with. This difficulty has actually been observed to occur in the Botworld domain, as follows:

The TOP $\xrightarrow{\mathtt{backward}} FacingMidline(?x)$ moves the bot backward until it is facing the bar midline. If the bot is known to be parallel to the bar, this should always work, since if the bot is not already facing the midline, it is facing away from it and backing up will cause it to cross the midline. Thus, one reasonable preimage estimate for the TOP is $ParallelTo(?x)$. Now, this TOP, when executed, always has the side effect of causing $OnMidline(?x)$ to become true since the bot is on the midline at the point that $FacingMidline(?x)$ becomes true. Suppose that $OnMidline(?x)$ is thus included as a guaranteed add-list element of the TOP. Therefore, the regression of the condition $OnMidline(Bar1) \wedge FacingMidline(Bar1)$ through the TOP $\xrightarrow{\texttt{backward}} FacingMidline(?x)$ would be $ParallelTo(Bar1)$. This regression results in the plan segment shown in Figure 2.5. But when the plan is actually applied, the bot may get into a situation where $ParallelTo(?x) \wedge FacingMidline(?x)$ holds. Thus, node N_1 in the figure is activated, and the bot continues to move backward indefinitely without ever causing its parent node to become true.

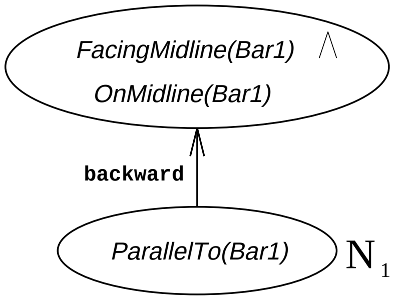

Figure 2.5: A TR Node Created Using a Faulty Add-List Element

Obviously, this problem could be solved for this particular instance by simply revising the set of predicates used in the domain (for instance, we might make $OnMidline(?x) \supset \neg FacingMidline(?x)$.) However, the need for this particular constraint only became clear after extensive experimentation with the Botworld domain. There is no reason to expect the designers of future domains to be able to conceptualize their domain so as to avoid all such potential difficulties. And the process of later human modification of a domain is time-consuming and difficult. Therefore, in designing TRAIL, we attempted to include mechanisms that could deal with domain descriptions that may be imperfect without requiring human intervention.

Therefore, due to situations such as the *OnMidline* add-list difficulty, a condition can only be a reliable add list element if it is in fact a logical consequent of the goal of the TOP. Therefore, our use of add list elements in simplifying regressions is limited to *domain axioms* specifying that one literal implies another. These domain axioms can either be specified by the designer of the domain or learned through a quite straightforward process of keeping statistics on the co-occurrence of predicates.

2.3.7 An Example of a TOP

As an example of a complete teleo-operator, we describe in this section a typical TOP, which is used to achieve the predicate $AtGrabbingDist(?x)$ using the `forward` action. (Recall that $AtGrabbingDist(?x)$ is true if and only if the bot is a particular distance from the center of bar $?x$, whether or not the bot is aligned with the bar midline.) The TOP itself is shown in Figure 2.6.

Postcondition:	$AtGrabbingDist(?x)$	
Action:	`forward`	
Preimage Condition:	$FacingBar(?x) \vee \neg TooFar(?x)$	
Success Rate:	95%	
Average Time:	75 ± 25	
Delete List:	$TooFar(?x)$	100%
	$FacingMidline(?x)$	15%
	$OnMidline(?x)$	10%
	$FacingBar(?x)$	5%
	$AtGrabbingDist(?y)$	100%
	...	
Add List:	$\neg TooFar(?x)$	
Instances:	...	

Figure 2.6: A Sample TOP

The postcondition and action of the TOP are self-explanatory. The only TOP variable in this TOP is $?x$, which refers to the bar that is being approached. Note that the precondition and all but one of the delete list elements are defined in terms of $?x$. (The other delete list element includes the unbound variable $?y$, and will be discussed below.)

The preimage condition of the TOP is relatively straightforward. There are two conditions under which moving forward is guaranteed to achieve $AtGrabbingDist(?x)$. First, if the bot is very close to bar $?x$, moving forward will eventually cause it to get far enough from the bar that $AtGrabbingDist(?x)$ will become true, hence the $\neg TooFar(?x)$ disjunct in the preimage condition. Second, if the bot is further from the bar, then as long as it is approximately facing the bar, moving forward will bring the bot close enough that $AtGrabbingDist(?x)$ becomes true. Thus, the predicate $FacingBar(?x)$ is included as a disjunct of the preimage condition. However, note that $FacingBar(?x)$ and $\neg TooFar(?x)$ do not completely cover the true preimage region of the TOP. Suppose the bot is further away from the bar than the grabbing distance, and is facing a few degrees to the right or left of the bar. In this case, moving forward may well cause $AtGrabbingDist(?x)$ to become true. However, given the set of Botworld predicates presented in Section 1.2.1, there is no way for TRAIL to include this region in the preimage condition, short of changing the preimage to T. This is a good example of the situation depicted in Figure 2.3, in which the preimage description language is insufficiently expressive to describe the actual preimage of a teleo-operator. (However, in this case, this limitation is probably an advantage for TRAIL as the given preimage condition is very usable for planning.)

The success rate of the TOP expresses the percentage of the time that the TOP is applied that it successfully achieves the postcondition. This TOP actually fails about 5% of the time, due to the inaccuracies of the $FacingBar$ predicate first discussed in Section 1.2.1. The 5% failure rate reflects those runs in which the bot is facing the bar from a distance, but is not quite facing the bar directly, causing $FacingBar(?x)$ to become false before the postcondition becomes true. Thus, the preimage condition becomes false, and TRAIL detects a so-called *activation failure*, as will be presented in Section 4.4.1.

The next item is the TOP execution time. TRAIL maintains the mean and standard deviation of the execution times for runs of each TOP. TRAIL will later use these times to determine whether the TOP has failed during execution. (This process is also discussed in Section 4.4.1.)

Next, the TOP has a list of delete list elements, corresponding to predicates that might become false during execution of the TOP. The first condition, $TooFar(?x)$, is guaranteed to become false since it is inconsistent with the postcondition of the TOP. The next condition, $FacingMidline(?x)$, might become false in the case where the bot is very near the bar and facing the midline. As it moves forward, it crosses the midline before reaching the correct grabbing distance. Thus, $FacingMidline(?x)$ could become false during execution of the TOP and is included as a delete list element.

Now, we skip to the fifth delete list element, $AtGrabbingDist(?y)$. Recall that since $?y$ is an unbound variable, the element $AtGrabbingDist(?y)$ matches any literals of the form $AtGrabbingDist()$ in the state description. Thus, the delete list element expresses the fact that if the bot is at the proper grabbing position for some other bar $?y$, moving forward to the grabbing position for bar $?x$ will change that fact.[5]

In contrast to the delete list, the add list of the TOP is quite simple. As was explained in Section 2.3.6, the add list only contains literals that are implied by the postcondition. In this case, the add list contains only the obvious condition $\neg TooFar(?x)$.

Finally, the TOP includes a pointer to a list of instances in which TRAIL has observed the TOP either succeed or fail. These instances are retained in case TRAIL later observes successes or failures that contradict the current TOP preimage, allowing TRAIL to learn from the unexpected success or failure. TRAIL's learning process is discussed in detail in the remainder of this thesis, beginning in Section 2.5.

2.4 The Use of Teleo-Operators in Planning

Given a set of TOPs, a goal condition, and an initial state, TRAIL can use well-known AI planning techniques to create a teleo-reactive tree that is expected to achieve the goal. Recall that a TR tree is very similar to the search tree that would be generated by some form of backward-chaining planner. The root node corresponds to the goal condition, and the conditions on nodes further down in the tree are simply regressions of their parent nodes through some TOP. At least one leaf node of the tree must be labeled with a condition that is true in the initial state; this is the point at which the search was terminated. Other branches in the tree correspond to other paths that were searched during planning; they may or may not have leaf nodes that are true in the initial state.

[5] TRAIL assumes that the postcondition of a TOP takes precedence over any delete list elements of that TOP, thus avoiding any confusion over whether $AtGrabbingDist(?x)$ should be deleted from the state description.

Thus, in order to create a plan, TRAIL does depth-first iterative deepening search (Korf 1985), beginning with the goal node as the root. Search continues until a node is found that is true in the initial state. At this point, the remainder of that depth level is searched, potentially producing alternative plan branches with the same length. Branches of the search that do not end in leaf nodes that are true in the initial state can either be retained, potentially allowing the tree to cover additional situations, or pruned as likely to be irrelevant. TRAIL prunes these unsuccessful branches, resulting in relatively small trees that may still have multiple branches (due to the fact that the entire level is searched.) Thus, in TRAIL, all leaf nodes are initially true when a tree is created.

Nonlinear plans with conjunctive nodes are created using essentially the same search mechanism. When the planner is attempting to plan a conjunctive goal or subgoal, it tries to construct a plan that treats each of the m conjuncts as an independent subgoal. Meanwhile, it also creates a different branch that treats the conjunction as a single goal and searches backwards from there. If the planner cannot construct non-interfering plans for the individual conjuncts, it can still construct a linear plan for the goal by regressing the entire conjunction through TOPs.[6]

Recall that TR trees can also be hierarchical, in that the actions associated with a node can be calls to other TR trees as well as primitive actions. In order to create hierarchical trees during planning, the planner needs to have descriptions of higher level operators, similar to macro-operators, that can be used for top-level planning. Descriptions of these higher-level operators can be used in the same way as TOPs are used for lower-level planning. The high-level operators themselves are also implemented as TR programs, which can be generated by the planner and generalized by replacing constants with variables, as is done in Explanation-Based Generalization (Mitchell, Keller & Kedar-Cabelli 1986). TRAIL does not presently learn higher-level operators; this issue is very important in scaling up to more complicated domains and will be discussed further in Section 7.2.

The TRAIL planner does not represent any advances of the state of the art in planning systems, but it has proven adequate for our learning experiments. Plans

[6]The details of this planning process are not important to the rest of this thesis.

using the learned TOPs are generated in anywhere from a few seconds to about a minute, depending on the complexity of the task.

2.4.1 Regressing a Condition Through a TOP

The fundamental step in the TR planning process is the backward regression of a condition through a teleo-operator. This process is very similar to the regression of a condition through a STRIPS operator. Suppose that a plan node with condition $\lambda \wedge \phi$ is to be regressed through the TOP $\xrightarrow{\mathsf{a}}\lambda^*$, where λ^* unifies with λ. We wish to generate the correct condition on the child node, along with the probability of success of the action taken in the child node.

The regression computation begins by finding the unification θ of λ and λ^*. The basic regressed condition is then $\pi\theta \wedge \phi$, where π is the preimage of the TOP. Note that since π can contain variables not found in λ, the regressed condition may contain variables. Thus, a TR tree may include conditions containing variables, such as $At(Robot, ?y) \wedge At(John, ?y)$.[7]

In order to compute the correct regression, the planner also needs to compare the literals in ϕ to the delete list elements of the TOP, as discussed in Section 2.3.5. Since delete-list elements are nondeterministic, they are taken into account in the process of computing the probability of success of the action. The planner begins with the estimated success rate recorded for the TOP. For each literal in ϕ that matches a delete list element of the TOP, it records the probability p that the literal will be deleted. The probability of success of the child node is then multiplied by $(1-p)$. If the probability of success drops below some threshold, the regression returns F, else it returns the condition $\pi\theta \wedge \phi$ along with the probability that taking action a will cause $\lambda \wedge \phi$ to become true.

These node success probabilities are used in two different ways in TRAIL. In Section 4.4.4 we will see how the learning mechanism uses them to guess whether an action failure is due to a random execution error or an actual error in the TOP.

[7]We realize that the regression of conditions containing variables through actions can raise a number of difficult questions. These issues did not affect the performance of TRAIL and are beyond the scope of this thesis. See Nilsson (1980) for some further discussion of regression.

Meanwhile, the planner also uses them to estimate the probability of success for a branch of the tree. Assuming again that action failures are independent events, the probability of success for any node N_i in the tree is simply the product of the node success probabilities for all nodes on the path between N_i and the root node of the tree. The planner keeps track of these probabilities and terminates search any time either the regression returns F or the probability of success of the branch drops below some threshold.

2.4.2 Plan Libraries

If the agent is going to be executing tasks repeatedly over the course of its lifetime, it will often be advantageous for the agent to keep a library of previously constructed TR trees. Since each tree was constructed using a set of TOPs, the trees can be generalized fairly easily by simply substituting variables for constants whenever the TOP indicates that the value of the constant is not relevant to the plan. The overall process is very similar to the generalization method used in Explanation-Based Generalization (Mitchell et al. 1986). Thus, when TRAIL is given a new task, it can examine the existing trees to find out if one is appropriate for the new goal. If a tree is found such that the goal of the tree is a generalization of the new goal, TRAIL can employ that tree instead of constructing a new tree through planning.

2.4.3 Iterative Replanning

A key advantage of the TR tree formalism over other reactive control formalisms is the ease of replanning during execution (Nilsson 1994). Since the trees generated by the planner or retrieved from the plan library do not usually represent universal plans, situations may arise during execution that are not covered by any node in the current tree. Such a circumstance can occur for a variety of reasons, including nondeterministic action outcomes, execution errors, and the actions of other agents. Fortunately, these unexpected situations will usually require only an extension of the existing TR program, rather than a complete replanning.

This extension process is very similar to the original planning process discussed earlier. The existing TR tree can be viewed as a partially constructed plan. Thus, in order to cover a new and unexpected situation, the planner simply does a backward-chaining search from each node in the current TR program until a condition is reached which holds in the novel situation.

Since the existing nodes in a TR tree are retained in the replanning process, the coverage of the tree increases monotonically as the agent gains more experience in the world. (However, it may decrease if TRAIL's learning mechanism corrects an incorrect TOP used in the construction of the tree.) As described above, the revised trees can be kept in a plan library for future use. Thus the agent's TR programs cover an increasing variety of situations, representative of the situations actually encountered.

Of course, the process of plan extension during execution will cause a delay before action can be resumed, but we assume that such planning can occur asynchronously with continuous monitoring of the other active trees in the agent's memory. If a situation that requires immediate action is likely to arise while the agent is replanning, the agent will need to have a reactive tree in its memory to insure an appropriate response The issue of real-time action selection is discussed further under future work in Section 7.2.

2.5 Correcting Trees Built with Incorrect TOPs

The above sections assume that TRAIL is given a complete and correct set of TOPs which can be used for planning. However, it is often difficult for the human designer to give a complete specification of the effects of every action in a form that is useful for the planner. (We will examine this issue further in the next chapter.) Therefore, we would like our agent to have the ability to learn these TOPs through experience in its environment. The remainder of this thesis will discuss the methods that TRAIL uses for such TOP learning. But regardless of the specific learning methods used, the fact that TOPs are being learned while the agent is acting affects the planning system of the agent.

First of all, incorrect TOPs will often require the agent to do replanning during execution. Incorrect TOPs will in general cause the planner to construct an incorrect plan for some goal. This plan will fail during execution (as discussed in Chapter 4,) allowing TRAIL to correct the faulty TOP through its learning mechanism. Once the TOP is corrected, the plan will need to be modified so that it is consistent with the new TOP. This can be done by simply recomputing the regression for each action arc that used the incorrect TOP. If this modifies the condition on the child node, the new condition will need to be regressed through the entire subtree rooted at the child node. The plan is of course cut off at any point where the regression produces the condition F. If this modification process produces a tree in which no condition holds in the current situation, then TRAIL will need to extend the tree using the iterative replanning process described in the previous section.

If TRAIL is modifying its set of TOPs through learning, plan reuse is also made somewhat more complicated. Suppose TRAIL had constructed a TR tree using a set of TOPs, which are later modified. Then, before reusing the stored tree, it will need to verify that the stored TR tree is consistent with the updated TOPs. This is true even if the stored TR tree was successful when used previously: the TR tree may have been overgeneralized during storage (if the TOPs used were overly general) or there may be an error in a branch of the tree that was not previously executed. In either case, if TRAIL discovers that the tree is incorrect, it can use the same process described above, recomputing the regressions and extending the tree through iterative replanning. Once again, this iterative replanning is usually considerably simpler than a complete replanning of the tree from scratch.

Chapter 3

Learning in Autonomous Agents

Machine learning is a large and varied field, with methods ranging from sub-symbolic methods such as neural networks to high-level symbolic methods such as Inductive Logic Programming. In this thesis, however, we are interested not so much in the learning methods themselves as in the application of these methods to autonomous agents, in particular those that modify their behavior in response to their experiences in their environments.

The design of a learning mechanism for a particular autonomous agent depends on a number of factors. First of all, it depends on what the agent is trying to learn. Is it trying to learn to achieve one particular goal? Does it need to transfer knowledge from one task to another? Is it trying to develop a complete model of the environment? Second, it depends on what kind of help is available to the agent. Does it have some initial knowledge of the environment? Does a trainer tell it what it is doing wrong? Finally, it depends on the type of environment the agent will be in. For instance, continuous and dynamic environments are harder to learn from than discrete, static ones.

The remainder of this chapter discusses these issues, and describes how they affected our choices for TRAIL's learning algorithm.

3.1 Approaches to Autonomous Agent Learning

Learning from an environment can take many forms. At one extreme, we might choose to build a complete model of the environment. Given such a model, the agent can use standard graph search techniques to select appropriate actions. At the other extreme, we might simply build a large table stating what to do in every possible situation. Clearly, each method has its advantages and disadvantages. In this section, we present four different categories of autonomous agent learning and discuss the benefits and drawbacks of each.

3.1.1 Policy Learning

Our first method of learning for autonomous agents is based on the straightforward idea of learning what to do in every possible situation. If we assume that the goals of a learning agent are fixed, then the agent has no need to develop a model of the environment itself. Instead, it suffices simply to learn how to respond to every case that might arise. We call this approach *policy learning* because the agent is learning a policy of what action to take rather than learning about the environment itself.

The purest implementation of the policy learning approach can be found in the field of behavioral cloning (Urbančič & Bratko 1994, Sammut, Hurst, Kedzier & Michie 1992). In behavioral cloning, the agent observes another agent (usually a human trainer) successfully performing the task, and records what the training agent does in each situation. When faced with a situation in the future, the clone agent compares the situation to situations in which it saw the trainer act and chooses an appropriate response. Behavioral cloning has many desirable properties, including an ability to learn from few trials and an ability to perform more consistently on many tasks than the human trainers it learned from (Bratko, Urbančič & Sammut 1995).

It might seem that this approach would be difficult to implement in complex environments; after all, it must specify what to do in every possible situation, including every possible value of each real-valued variable. However, behavioral cloning can make use of the fact that the action that should be taken is usually identical over large portions of the possible state space. Once the learner has observed an action for a subset of the possible states, it can generalize over its inputs to produce a policy that covers new states as well. Neural network methods (Hertz, Krough & Palmer 1991) and modern concept learning algorithms such as C4.5 (Quinlan 1992) are very good at creating simple policies from examples, including appropriate discretizations of continuous input variables. On the other hand, the learner might simply record the observed state-action pairings explicitly and use a nearest-neighbor matching scheme to determine the correct action, as was done in the robot learning work of Moore (1990).

An impressive application of behavioral cloning is found in the ALVINN system (Pomerleau 1991, Pomerleau 1993). Steering a car or van is a natural task for behavioral cloning, as most of the time the goal is essentially fixed – stay on the road and within the lane. ALVINN observed the actions of a human driver and used a back-propagation neural network to generalize a mapping from views of the road to steering directions. ALVINN has successfully driven on unlined paved paths, jeep trails, city streets, and interstate highways.

The various forms of behavioral cloning all represent supervised approaches to policy learning, in which the learner is given a set of correct state-action pairings. Policy learning can also be done by unsupervised learners, using reinforcement learning methods such as Q-learning (Watkins 1989). In Q-learning, the agent learns the value of each action in each particular state, assuming that certain states have a numerical "reward" associated with them. Learning is essentially done by propagating rewards backward through the actions that led to them, eventually converging on an estimate for the expected value of each action that might be taken in each state. The learned policy is thus to choose the action that maximizes the expected future reward. Q-learning has been applied successfully in a number of agents, such as a box-pushing physical robot (Mahadevan & Connell 1992).

Both behavioral cloning and Q-learning suffer from the obvious problem of inflexibility in the face of different goals that an agent might have. The policy learned for one task may well be useless for another. (If the goals do vary, we could still use this approach by assuming that the goal is part of the agent's environment. Unless there are only a few possible goals, this approach makes the state space so large as to be completely intractable.) Thus, if our autonomous agent is to be versatile enough to handle a variety of possible goals, it will need to learn at least a partial model of its environment.

3.1.2 Environmental Modeling

Our next strategy for learning from the environment takes an almost entirely opposite approach. Rather than explicitly learn what actions to take, an agent may be able to learn a complete description of the environment itself. Building such a model is clearly the safest way to achieve good performance; if an agent has a complete description of the environment, it can (theoretically at least) compute its optimal behavior for any goal and any situation in which it finds itself.

Of course, in order to learn such a description, the agent must first make some assumption about the nature of the environment. If we assume that the environment can be modeled as a deterministic finite state automaton, where the inputs are the agent's actions and the outputs are the agent's perceptions, then Rivest and Schapire's (1993) extension of Angluin's (1987) L^* algorithm can in principle be used to produce an exact model of the environment with high probability. Somewhat more realistically, the world can be modeled as an action-driven hidden Markov model (Rabiner & Juang 1986), in which actions nondeterministically move the environment to some new state. A variety of algorithms allow for the approximation of small hidden Markov models, such as the Baum-Welch or forward-backward algorithm (Baum, Petrie, Soules & Weiss 1970, Rabiner 1990).

The choice of model is very important to any attempt to accurately learn an environment. Most environments cannot be modeled accurately as deterministic finite automata, due to noise, other agents, or simply random action effects. Most environments *can* be modeled as hidden Markov models, although real-valued variables still present a problem - any environment containing a real-valued variable can be in an infinite number of possible states. Of course, such a state space can be made finite by discretizing the real-valued variables, but how best to do these discretizations is an open question.

The main difficulty with this style of learning lies in the complexity of scaling up the learning to environments that are more complicated than the small examples usually used to illustrate the learning techniques. Hidden Markov models of environments can often be very large, especially if they use discretizations of real-valued variables, and the various learning algorithms converge very slowly, if they converge at all. At this point, it appears extremely difficult for hidden Markov model learning algorithms to cope with environments of more than about 16 states (Littman, Cassandra & Kaelbling 1995). Thus, as we saw in behavioral cloning, our learning agent will need to abstract its learned models from the details of the particular states involved.

3.1.3 Action Model Learning

A more practical approach to learning a model of an environment can be found by considering the abstraction methods used by the planning community. The introduction of STRIPS operators (Fikes & Nilsson 1971) and similar representations allowed planners to avoid having to consider all attributes of a possible world state. Using STRIPS operators, a planner can reason about only those aspects of the world that are relevant to a particular action (namely, the conditions that must be true for it to succeed, and the conditions that may change as a result.) Such *operator schemas* abstract the behavior of the environment into forms that can more easily be used by the agent to construct plans to achieve its goals. And unlike behavioral policies, operator schemas also allow for the natural transfer of learning across tasks, as operators learned in the context of one task can later be used in planning to achieve another task.

In the lowest-level case, these operator schemas simply take the form of local models of the environment from the agent's sensory perspective. Mahadevan (1992) has implemented a system in which action models are pairs of certainty grids (Moravec 1988) connected by actions, representing the effects of a one-step action from the perspective of the robot. These models can be used for action planning, and demonstrate a transfer of knowledge across tasks on both simulated and real robots. Another low-level action modeling system, SPLICE (Rogers & Laird 1996), learns models of control actions in continuous domains by developing local models of the effect of a control action on a target variable. However, at present SPLICE is limited to solving problems involving a single monotonic variable and a single control input.

At a higher level, the learning of symbolic models of actions for planning has been the subject of a large body of recent work, most notably by Shen (1989, 1994), Gil (1992), and Wang (1994, 1995*b*). The work of each of these earlier authors will be discussed and compared with TRAIL in a number of places in the remainder of this thesis.

As was true in the case of environmental modeling, an action-model learner must make assumptions about the properties of the environment in which it is learning. These assumptions will be discussed further in Sections 3.3 and 3.4; for now we will just mention that they have a significant effect on the process of action-model learning. As stated earlier, TRAIL builds on these existing action-model-learning systems by extending action-model learning to dynamic, continuous domains.

3.1.4 Specialized Knowledge Acquisition

Before we move on, there is one more important category of learning for autonomous agents that deserves mention. It is often possible to program most of the necessary knowledge into an agent while leaving one particular type of knowledge to be acquired from the environment. The most obvious example of such learning for autonomous agents is map learning - there have been a number of impressive experiments in which robots have been programmed with general navigational knowledge and have learned to navigate successfully in a new environment by learning a map of it. What differentiates these experiments from other types of learning from the environment is that the knowledge that must be acquired is of a particular format (namely, geometrical knowledge) and does not involve the complexity involved in, say, a hidden Markov model of the environment.

Such map learning is generally done by learning to identify distinctive locations, based on the robot's sensor readings. The robot then builds a topological map connecting the locations, which allows the robot to navigate successfully through the environment. Examples of such map learning are found in the work of Kuipers and Byun (1991), Galles (1993), and Yamauchi and Langley (1996).

3.2 Learning From Exploration and Teaching

For any given learning task, the difficulty of the task varies considerably depending on how much help is given to the learner by a human user. The learner may be asked to act completely independently, it may be given partial information about the environment beforehand, or it may have an on-line trainer that either makes suggestions while it is learning or provides demonstrations of successful behavior.

Independent learning is clearly the most impressive form of learning. Drescher (1991) has proposed a *schema mechanism* learner that operates in this manner. Drescher's system is a *tabula rasa* agent that tries to replicate Piaget's stages of learning, beginning with the simple observation that moving its hand to the right causes the agent to "see" its hand to the right of where it was before. From there, it is supposed to construct successively more complicated theories about the world. However, in practice it does not move much beyond connecting motor actions to sensory perceptions, falling considerably short of hoped-for high-level concepts such as object permanence.

The main difficulty with completely autonomous learning is one of exploration. If the environment has a very regular structure, an agent can learn the regularities of the world without actually exploring it in detail. The action-model learning system LIVE (Shen 1994) takes advantage of this fact in the Towers of Hanoi domain – once it has learned the rules for moving disks around, it can solve arbitrary problems even though it has not seen very many of the 3^n possible world states. However, most environments have considerably less regularity than Towers of Hanoi. For example, consider the task of grabbing a bar in the Botworld domain. The precondition for the `grab` action to succeed is in fact the conjunction of the four conditions $AtGrabbingDistance$, $FacingBar$, $AlignedWith$, and $\neg Holding$. If the bot tries to grab the bar from this situation, it will succeed and learn at least one path to the goal. However, until it happens to be in that situation, it has no knowledge whatsoever about how to achieve the goal. There is no way for it to know whether it is getting close or not. Since the complexity of exploration can often be exponential in the size of the domain, this problem can make autonomous learning extremely difficult in many domains.

There have been a few attempts to solve the exploration problem without outside assistance, generally through some form of directed experimentation. If the agent can intelligently decide which actions to try, it can perhaps guide its search toward unexplored regions of the state space and thus explore the environment more quickly. Intelligent exploration has been studied somewhat in the reinforcement learning community (Moore & Atkinson 1993, Peng & Williams 1993) as well as in LIVE, but there is still no way to avoid lengthy random exploration in domains in which goal achievement requires a particular sequence of actions, such as the bar-grabbing action sequence in Botworld.

One way of simplifying the exploration process is to provide partial rewards along the path to the goal, acting to focus the agent's explorations in the direction of the goal. Partial rewards would clearly be useful for an unsupervised learning method such as Q-learning, but it is difficult in general to come up with these partial rewards. These rewards are also quite goal-dependent, so they must be computed separately for each goal.

Another way of simplifying exploration is to provide the agent with a partial model of the environment, which the agent can then modify. This makes learning significantly easier, and is often much less difficult than the task of providing the agent with a complete and accurate model. Within the context of action-model learning, this approach was used in Gil's EXPO system (Gil 1992). EXPO assumes that the user provides a set of approximately correct action models, which the system uses to make tentative plans to achieve the given goals. If the tentative plan fails, EXPO can update the operators appropriately. The partial action models in EXPO can have missing preconditions or effects, but cannot have extraneous preconditions or effects.

Finally, a trainer may be available to the agent while it is actually doing learning. There are various ways in which a trainer might aid a learning agent, such as suggesting actions to try or placing the agent close to the goal so that random exploration is likely to succeed. One very convenient way of aiding an agent is simply to be available to complete tasks on request. The learner can then observe the trainer and learn from the successful task completion. The action-model learning system OBSERVER (Wang

1995*b*) uses this method of trainer assistance, and we have chosen to use it in TRAIL as well.[1] Further details on TRAIL's use of the trainer are found in Chapter 4.

A significant advantage of having the trainer complete tasks in this way is that the trainer does not need to have any knowledge whatsoever about the agent's representation of the domain. All the trainer needs to be able to do is to complete the task. This method also allows for a natural method of measuring learning performance - the more frequently the learner can complete a task without consulting the trainer, the better its learning is. We discuss this measure further in Section 6.4.

3.3 Environments

As we stated earlier, the assumptions that an agent makes about an environment greatly affect the design of its learning algorithm. However, it is difficult to categorize environments in terms that will allow us to do an exact analysis of the expected behavior of a learning agent. It is sometimes possible to analyze behavior precisely given that the environment can be modeled as a simple mathematical structure such as a finite state automaton, but as we argued earlier, most real environments cannot be categorized in such simple terms. Factors such as real-valued variables and the unpredictable actions of other agents make modeling real environments very difficult.

However, complicated real environments are clearly not all the same with respect to the difficulty of learning, or of action-model learning in particular. Even if we cannot fully classify the sorts of environments our agents might encounter, we can still identify dimensions along which they vary, dimensions that might well affect learning performance. What follows is not intended as a complete characterization of environments, but rather as a suggestion as to what some of those useful dimensions of variability might be. A similar classification can be found in Chapter 2 of Russell & Norvig (1995).

[1] Behavioral cloning also uses this model of training, but TRAIL's learning focuses on the effects of the teacher's actions rather than on the actions themselves.

3.3.1 Continuous and Discrete Features

If an agent is to interact with its environment, it must have some way of sensing that environment.[2] For our purposes, the way in which this sensing is done is a fixed property of the agent, and can therefore be considered as an attribute of the environment.

The sensors of an agent might include both discrete features (boolean predicates and ordinally-valued variables) and continuous features (real-valued variables). If the agent senses only discrete features, we say that the environment is discrete (regardless of whether the underlying environment is in fact discrete or continuous.) Otherwise we say that the environment is continuous.

Many environments can be usefully described using only discrete features. For instance, if low-level navigation routines are built into a robot, an office delivery domain is essentially discrete - the robot is in one of a finite set of rooms, objects are possessed by one of a finite set of people, etc. Even the Botworld domain described in Section 1.2.1 is discrete if the bot is given the set of predefined predicates - while the simulation itself takes place in a continuous world, the bot only senses a finite number of predicates, i.e. for each bar, is it facing the bar, is it at the proper grabbing distance, and so on. This set of predicates is the bot's only way of sensing the world, so the world appears discrete to the bot.

On the other hand, there are environments that for practical purposes cannot be described using discrete features. One good example of this is the flight simulator domain. Most of the features in this domain are continuous, such as altitude and speed. Of course, continuous features can be discretized, but it is difficult to decide in advance on a discretization that keeps all the relevant information in a feature.

This distinction can be critical for learning. Behavioral cloning can often be done whether features are continuous or discrete, as most concept learning algorithms can deal with real-valued attributes. However, algorithms that build models of the environment normally assume that the world can be modeled as a structure with

[2]Of course, this does not necessarily mean it can sense the complete world state; see the next section.

discrete states. Most of these algorithms cannot deal with real-valued variables. Finally, much of the existing action-model learning work assumes that the world can be described by discrete predicates, and thus does not reason about continuous features. LIVE's Towers of Hanoi domain is purely discrete, and although EXPO and OBSERVER's part machining domain contains numeric constants, the system's learner does not do any reasoning about their possible range of values.

3.3.2 Propositional and Structured State Descriptions

The sensors through which an agent perceives an environment can often provide useful structure to a learning system. The sensors provide a description of each state of the environment, which may later be used as a learning instance. In the simplest case, these descriptions are purely propositional, consisting of a set of boolean predicates such as $HandFree$ and $BlockAonTable$. More usefully, the descriptions can be represented using an attribute-value format, consisting of a number of attributes and their values in the state. Each attribute is either real-valued, such as $BatteryLevel$, or discrete, such as $RoomWhereRobotIs$. This representation allows for the expression of real-valued quantities, and also avoids the need to have a large number of propositions such as $RobotInRoom101$, $RobotInRoom102$, etc.

Finally, the most complex form of representation that we will examine is ground first-order logic. Each state description consists of a set of ground literals, which may describe the properties and relations among a number of objects in the environment. We call such state descriptions *structured*, since each description contains internal structure, in the form of relationships between the objects used in the descriptions. It is important to note that structured representations are not any more expressive than attribute-value representations (any ground situation description can be converted to a set of propositions and continuous attributes) but that a first-order description may be much easier to learn from. In fact, many concepts that are simple to express in first-order logic, such as relationships between objects, may be very difficult to express if the situation description is strictly propositional. Therefore, a realistic mechanism for learning from environments will have to be able to deal with structured situation descriptions.

3.3.3 Determinism

Most environments are not completely predictable. In any real environment, an action taken multiple times in apparently identical states may have different effects each time. There are at least three possible reasons for such nondeterminism. First, other agents may be taking actions which affect the agent and its environment. Second, states which appear identical to the agent may not actually be identical, a condition known as *perceptual aliasing* (Whitehead & Ballard 1990). Perceptual aliasing can result either from noise in the agent's sensors or from sensors that are unable to sense some relevant features of the environment, such as whether a door is locked. Finally, the effects of the action may simply be random. Truly random effects could actually be considered as a special case of perceptual aliasing, but the term is usually used to refer to circumstances where there is a well-defined difference between the states that is simply not perceptible to the agent. On the other hand, the occasional failure of a robotic grasper, for instance, may not be due to any detectable cause.

It is extremely important for a learner to know whether randomness occurs in its environment. Methods that are based on an assumption of determinism usually fail totally when presented with apparently inconsistent behavior. LIVE, EXPO, and OBSERVER all assume that their environments are deterministic. TRAIL does not; its methods for dealing with nondeterminism are discussed in the next two chapters.

There are at least three possible ways of dealing with nondeterminism in a learning agent. First, the agent may model action effects as a random process, as would be done in a hidden Markov model learner. Second, it may attempt to discover hidden variables which explain the apparently random effects, such as a *DoorLocked* predicate. We discuss such automated feature construction in the future work discussion in Section 7.2. Finally, it may simply acknowledge that the models it is learning are not guaranteed to be correct, but are correct under most circumstances. Due to the robustness inherent in the TR execution mechanism, TRAIL can learn and use models that will work under most circumstances but may sometimes fail.

3.3.4 Durative and Atomic Actions

Another dimension along which environments can vary is the types of actions that the agent can take. An atomic action is one in which the agent does not sense the state of the world while the action is taking place, and thus cannot respond to anything that happens in the meantime. Thus, even if the action itself takes time (consider the act of picking up a block in Blocksworld) it is in effect instantaneous as far as the agent is concerned. On the other hand, actions such as a robot moving through a room are better described as *durative* actions in that they continue over some period of time during which the robot is sensing, and perhaps taking other actions as well. Any such durative action could also be represented as a series of small atomic ones (in this case, perhaps an `accelerate` action followed by a number of `wait` actions) but this is not a very intuitive or convenient representation.

Nearly all of the existing environmental learning work has focused exclusively on atomic actions, since they can be easily reasoned about using straightforward *state-action-next state* representations such as finite state automata and STRIPS operators. (Two notable exceptions are the work on map learning and DeJong's work on learning to plan in continuous domains (DeJong 1994), which will be discussed further later.) However, we believe that durative actions are a fundamental component of many environments and that they must be considered when designing an autonomous agent's learning algorithm.

3.3.5 Hostile Environments

Finally, in any environment in which other agents can act, there is a danger that the behavior of the other agents will be hostile, that is, intentionally or unintentionally blocking our agent's attempts to achieve its goals. This case clearly differs from cases in which the goal is simply unreachable from the current state; in the case of hostile agents, the agent theoretically should be able to achieve its goals, but is repeatedly prevented from doing so. It also differs from the case where action failures occur randomly; if the failures are truly independent events, repeated attempts will eventually achieve the goal. The effects of other agents may well be non-random, thus leaving the agent with no guarantee of eventual success.

Of course, we do not have any way of avoiding this problem for our agents. We are merely noting that the possibility of hostile agents is another reason it may be difficult to build a useful model of a real environment, or to prove useful theoretical results about agent performance. Teleo-reactive trees are built to handle occasional action failures or outside interference, but of course will fail if some outside agent is repeatedly causing action failures. For more on this issue, see the discussion of TR trees in Section 2.1.

3.4 TRAIL's Environmental Assumptions

As we have been discussing, TRAIL has made a certain set of assumptions about the environment in which it will function. These assumptions do not guarantee that TRAIL will successfully perform in any particular domain, but they are necessary conditions in order for TRAIL to be able to operate. In brief, TRAIL makes the following assumptions about the environments in which it is learning:

- A teacher is available who can guide the agent to complete any goal that TRAIL does not know how to achieve. The teacher does not need to give any explanations of its behavior or have any knowledge of TRAIL's domain representations.
- The environment can contain both continuous and discrete features. Learning will be faster, however, if the environment is purely discrete, since the learner will not have to induce intervals over continuous attributes.
- The state descriptions are in ground first-order logic. The learner can thus use Inductive Logic Programming to learn first-order concept descriptions that are more expressive than the attribute-value concepts used by most learners, as will be discussed in more detail in Chapter 5.
- The state descriptions are complete. This assumption allows the learner to include negative literals in learned concepts, by assuming that literals that are not included in the state description do not hold. Note that this assumption

does not require the agent to sense the complete world state at each time step; a stored world model can be used to generate some of the literals in the state description.

- Actions can be durative or discrete, and action names can contain variables, as in `pickup(?x)`. (Operators, of course, can contain variables as well.) We do assume that TRAIL is given the set of low-level actions that it can perform, although it does not initially have any information as to their effects.

- The preimages of the learned operators can be effectively approximated using the given concept language and perceptual predicates. TRAIL may not be able to learn a precise description of each operator preimage, due either to perceptual aliasing or to limitations in the concept description language, but it should be able to approximate each preimage such that:

  1. The learned preimage is sufficiently general to be useful for planning.
  2. The learned preimage does not contain states such that once the agent is in the state, it is repeatedly unable to achieve the goal due to some hidden condition. (This difficulty could be resolved using feature creation, which is not currently implemented in TRAIL.)

- The behavior of the environment may change over time, but is not changing so fast that TRAIL's learning mechanism is unable to modify its set of TOPs rapidly enough to keep up with the world.

- Other agents may take actions that interfere with TRAIL's goals, but this interference should not be intentionally or unintentionally hostile. In other words, the interference must not be such that TRAIL is repeatedly frustrated in its attempts to achieve goals.

- The world is benign, in that the agent will not encounter any situations during learning from which it cannot recover. Most learning systems implicitly make this assumption, as it is impossible to avoid unrecoverable situations without very specific domain knowledge. Incidentally, most simulations are trivially benign, as the agent can simply restart the simulation.

In the next chapter, we contrast the action-model learning methods used in TRAIL with the methods used in more traditional, STRIPS-like environments. We use the term "STRIPS-like" to describe domains in which many of the traditional assumptions of a generic STRIPS-style planner hold. Many of the above assumptions, such as the assumption of a benign world, are also made in such environments, but there are three particular assumptions which are critical in a STRIPS-like environment, as follows:

- State descriptions are in ground first-order logic. They may contain continuous features, but actions can only set these features to specific values, and action preconditions can only require the feature to have a specific value, not a range of numeric values.
- Actions are atomic, taking a fixed amount of time. Changes that occur during an action can effectively be treated as occurring instantaneously.
- Action effects are deterministic. This implies that no other agents can take actions that affect the environment, and that therefore the effects of an action are exactly the difference between the state before it was taken and the state immediately following it.

3.5 An Overview of TRAIL

In summary, TRAIL is based on the idea that the most practical method of learning to achieve a variety of goals in continuous and dynamic environments is through learning models of actions. As we argued earlier, learning complete models of the environment in realistic environments is too computationally difficult, while learning only a behavioral policy is insufficiently general. Therefore, as it acts in an environment, TRAIL builds models of the actions it takes and the effects they have, using methods that are discussed in the following two chapters.

The TRAIL agent architecture is shown in Figure 3.1, and is discussed in further detail in Benson and Nilsson (1995) and in Chapter 6 of this thesis. All of TRAIL's behavior follows from goals that are given to the system by a human user.

In the absence of learning, the TRAIL architecture functions essentially as a standard plan-execute architecture with the addition of the teleo-reactive execution mechanism presented in Section 2.1. Once a goal is given to TRAIL, it obtains a plan to achieve the goal, either by planning from scratch (as described in Section 2.4) or by retrieving a TR tree from the Plan Library (Section 2.4.2.) This plan is then inserted into the TR memory. The agent also has inputs that allow the computation of conditions needed by the active TR tree. The immediate sensory data required for these computations are provided more-or-less continuously by a separate perceptual module—not shown in Figure 3.1—and are stored in the World Model along with other information about the environment that cannot be immediately sensed but that can be remembered. (The perceptual module is an independent component of the architecture and will not be further discussed here.) Finally, the executor determines which node in the TR tree is currently the highest true node and activates the appropriate action. If there is more than one active tree in the memory, goal arbitration can be done using an algorithm described in Benson and Nilsson (1995).

The adaptive behavior of TRAIL arises from the interaction of the planner, the reactive executor, and the action-model learner. These three components can be considered as a cycle that interrelates the processes of planning, execution, and learning.

As described above, TRAIL's planner receives goals in the form of conditions to be made true in the environment and produces plans in the form of TR trees that it expects will achieve these goals. Since the planner relies on a set of action models that has been produced by the learner, any inaccuracies in the learned action models may lead to incorrect trees being produced. The execution of these trees may lead either to the achievement of the agent's current goal, or to a call to the learner or teacher if any of a variety of failures occurs. Plan failures are discussed more fully in Section 4.4.

Finally, the learner receives records of action execution from the executor and uses them to update its set of action models, correcting inaccuracies that caused plan

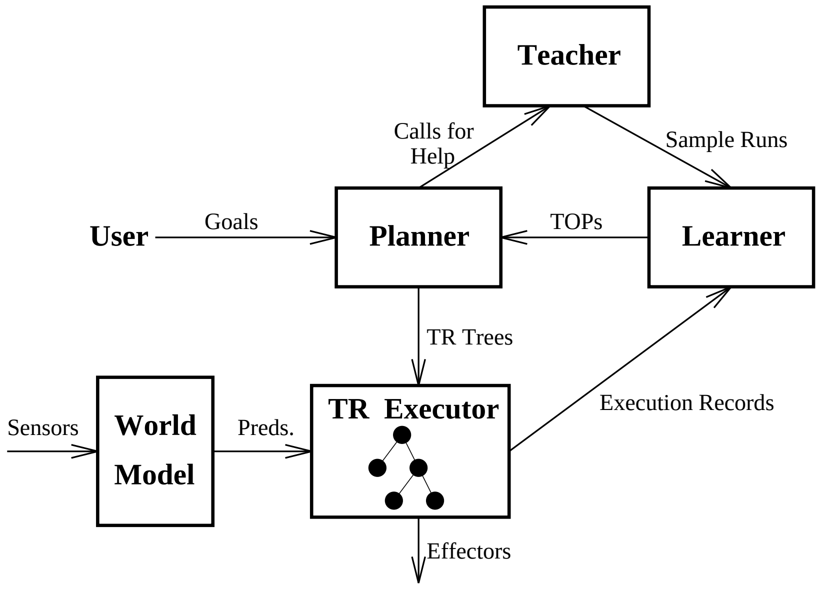

Figure 3.1: TRAIL's Agent Architecture

failures. The learning algorithm itself is based on a variant of the inductive logic programming algorithm DINUS (Džeroski et al. 1992) and is presented in Chapter 5. The updated action models are then used by the planner to create new trees, allowing the learning cycle to continue.

This cycle assumes that TRAIL has sufficient knowledge to create an approximately correct plan for the goal it has been given. However, if TRAIL's action models are insufficient to construct a reasonable plan, it must resort to some method of environmental exploration. As described above, the general problem of domain-independent exploration is still too hard to be solved without extensive domain knowledge. Therefore, TRAIL has access to an external teacher that it can call at any time to drive it through completion of its current goal.

Thus, once a goal is given to TRAIL, it uses a combination of planning, learning, and (when necessary) calls to the teacher to complete the goal. Each call to the teacher and each incorrect plan that is generated provides an opportunity for TRAIL to correct errors in its action models. Thus, over time, TRAIL manages to generate correct plans increasingly often. We will return to the overall behavior of the TRAIL system in Chapter 6.

Chapter 4

Instance Generation in TRAIL

In the last two chapters, we have presented our motivations for building an action-model learner and described the background of the TRAIL system, including the TR-tree control structure and the planning and learning architecture. In the next two chapters, we take a closer look at TRAIL's learning component as an inductive concept learner. This chapter describes the methods that TRAIL uses to generate examples, or training instances, from which it can learn, while Chapter 5 describes how TRAIL uses these learning instances to generate action models.

4.1 Inductive Concept Learning

The form of learning used in TRAIL is known as *inductive concept learning*, and is the most widely studied area of machine learning. (Other areas include reinforcement learning and learning to perform tasks more efficiently.) In the inductive concept learning framework, the learner is given a set of *training instances*, each of which is labeled with some *category*. The task of the learner is to produce a *classifier* that, given a new instance, will with high probability generate a correct label for the instance. This classifier can be a decision tree, a neural network, a set of rules, or any of a variety of other representations. Valiant's formal framework for inductive concept learning (Valiant 1984) and a number of different concept learning approaches are covered in a survey paper by Dietterich (1990).

In most of the existing inductive concept learning work, it is assumed that the set of training instances is given to the learner by some outside source. If the learner is attempting to learn medical diagnosis, it will be given patient records labeled with their diagnoses, while if the learner is doing behavioral cloning (see Section 3.1.1), the trainer will provide instances of situations labeled with the action that the trainer took in that situation. There is also a variety of work on query-based learning (Angluin 1988) in which it is assumed that the learner can query an oracle to obtain labels for particular instances. However, in TRAIL, we cannot rely on the training instances being in such a conveniently labeled format. Although it is difficult and computationally expensive for TRAIL to create directed "queries" in an environment (since it may not initially know how to achieve particular world states), the external teacher and TRAIL's own planner provide numerous examples of behavior. Unfortunately, the examples provided by the teacher and the planner are not in the form of action models. Thus, the problem we are focused on in this chapter is not the original generation of these experiences but in the conversion of the experiences into a set of labeled instances that can be used by a concept learner.

We can subdivide the process of learning action models into three separate phases. First, some module, either a teacher or an experimentation system, takes actions in the environment. Second, the learner observes these actions and categorizes them as positive or negative training instances for specific action models. Finally, an inductive concept learning algorithm takes these instances as input and outputs a description of the action model.

The first phase, that of selecting actions to take in the environment, has been studied in the context of several action-model learning systems, including LIVE (Shen 1994) and EXPO (Gil 1992). Both systems included directed experimentation modules that generate behavior useful for action-model learning. TRAIL, on the other hand, generates behavior by calling the teacher, by executing TR trees created by the planner, and by using certain limited experimentation strategies discussed later in this chapter.

The second phase, that of observing the behavior and producing labeled training instances for the learner, has been less thoroughly examined. The STRIPS environmental assumptions discussed earlier have made this phase essentially trivial for most other action-model learners, as we shall see in Section 4.3. However, for environments that contain continuous attributes and durative actions, producing labeled training instances is considerably more complicated. The remainder of this chapter discusses the difficulties involved in instance generation within TRAIL, and presents TRAIL's solutions to these problems.

The final phase of learning is the inductive concept learning itself. TRAIL performs inductive concept learning using an approach known as Inductive Logic Programming, which is intended to efficiently produce classification rules given instances that are described in first-order logic. TRAIL's concept learning algorithm is discussed in detail in Chapter 5.

4.2 Experience Records in Continuous Environments

If an agent is acting in an environment, whether it is under control of an external teacher or its own internal exploration or experimentation module, it is producing a series of experiences that a learner can observe. These experiences can be viewed as tracing paths through state space (see Section 2.3.2). Each experience consists of a sequence of states during which actions are executed. Durative actions may span several states in the sequence.

We can describe these experiences as state-action sequences:

$$S_1 \ \mathtt{a_1} \ S_2 \ \mathtt{a_2} \ S_3 \ldots S_{n-1} \ \mathtt{a_{n-1}} \ S_n$$

where each S_i is a description of a state s_i of the environment and each $\mathtt{a_i}$ is an action taken by the agent. Such a sequence is called an *experience record.* (As stated in the introduction, we use s_i to denote the state described by the state description S_i.) The state descriptions themselves, as usual, are conjunctions of ground predicate calculus literals, taken from the set of predicates provided by the domain designer.

If the environment is a STRIPS-like environment, it is clear what these experience records represent. Each $\texttt{a}_\texttt{i}$ is an atomic action that transforms the world state from s_i to s_{i+1}. (Of course, s_{i+1} may not actually differ from s_i, as $\texttt{a}_\texttt{i}$ might not have any effect in s_i.) Since the agent is the only entity capable of causing changes in the world, all of the differences between s_i and s_{i+1} are due to action $\texttt{a}_\texttt{i}$, and the entire effect of $\texttt{a}_\texttt{i}$ consists of these differences.

On the other hand, if the environment contains durative actions, or other agents that might affect the world state, the semantics of such a representation are less clear. Since the environment is continuously changing, we assume that the agent is sensing the environment at some sampling rate. By convention, we assume that state-action sequences in such environments represent a periodic sample of what the environment state was and what action the agent was taking at the time. Therefore, each action may span several states $s_i \dots s_{i+j}$ and effects may occur at several points during an action execution. Figure 4.1 represents several experience records in which action $\texttt{a}$ was used in attempting to achieve λ. In brief, the states labeled with a "+" in the figure will generate positive instances of the TOP $\xrightarrow{\texttt{a}}\lambda$, while the states labeled with a "-" will generate negative instances.

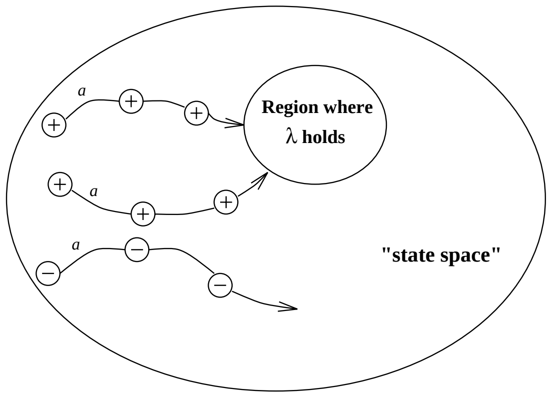

Figure 4.1: Experience Records, as Viewed in State Space

Observe that this execution record format corresponds naturally to the way in which an agent would execute a teleo-reactive tree: at some rate, the agent observes the environmental state (based on current sensor values and possibly also on a stored model) and computes an action based on it. If the agent is recording these

$(state, action)$ pairs as it is acting, then we are assured that the experience record provides a relatively accurate representation of the agent's behavior. In particular, if `ai` is a durative action, any changes that occured between states s_i and s_{i+1} are guaranteed to have occurred while the agent was executing `ai`. (Note that this does not imply that `ai` *caused* these changes.) If `ai` is a near-instantaneous atomic action, the changes may have occurred some time after `ai` was executed but before s_{i+1}. But still, the agent was not executing any *other* action during that time, so the effects can be at least tentatively attributed to `ai`.[1]

Throughout the remainder of this chapter, state-action sequences will be assumed to have been generated either by observing a teacher or by executing a teleo-reactive tree as described above. Since our agents may be executing with a high sampling rate, they can produce quite long experience records. Methods for shortening these records will be covered in Section 4.2.2.

4.2.1 Sources of Experience Records

If an experience record comes from an agent's observations of a trainer, then there is no information that can be recorded other than the state-action pairs. (Again, this is one of the main advantages of using a learning-from-training paradigm; the teacher does not need to give the learner any additional domain information.) However, if the experience record in question results from the execution of a teleo-reactive tree, the agent has additional information that can aid the learning process. Each node N_i in the TR tree (other than the root node) was generated by regressing the condition of its parent node through some TOP τ_i. Every action taken during the execution must result from some active node N_j in the TR tree, and thus must correspond to the TOP τ_j that was used in generating node N_j. Therefore, TRAIL records along with each action the TOP that was used to generate the node that was active at the time.

This information can be useful in several ways during learning. If the TOP was

[1]Note that this assumption prevents us from learning delayed effects that occur after the termination of the action. This problem is discussed further in Section 7.2.

successful (i.e. the parent node became true as a result of the action taken) then TRAIL can record a positive instance of that TOP directly, while if the TOP failed in some way, negative instances for the TOP can be easily generated. Both of these processes are covered in more detail in the remainder of this chapter.

4.2.2 Simplifying Experience Records

As observed above, if the agent is sensing the world at a high sampling rate, it will produce a very long experience record. This will greatly increase the computation time necessary to learn operators from the sequence. Therefore, we need some method of simplifying the state-action sequences to produce more manageable representations.

The obvious method for experience record simplification is to collapse sequences of state descriptions that are effectively identical. For instance, if a subsequence of the experience record is S_i $\mathtt{a_i}$ S_{i+1} $\mathtt{a_{i+1}}$ S_{i+2} where S_i and S_{i+1} are identical and $\mathtt{a_i}$ and $\mathtt{a_{i+1}}$ are the same action, this subsequence can be condensed to S_i $\mathtt{a_i}$ S_{i+2} with no loss of information. If this simplification is done, the state descriptions must also be annotated with the actual number of intermediate states that occurred since the previous state in the condensed record. This information will be needed later in estimating the average time an action takes to achieve an effect.

We can condense the experience record still further if the learner is given some domain information as to which state attributes are likely to be relevant to the actions we are learning. A real environment may contain many irrelevant attributes, and if these attributes are frequently changing, they can result in numerous irrelevant changes in state between the relevant changes. If the learner knows that a set of attributes is irrelevant, the state descriptions S_i and S_{i+1} can be considered identical and combined, as described in the previous paragraph, any time they differ only on the set of irrelevant attributes. Again, this process will considerably simplify the experience records, but at the cost of potentially discarding important information if the wrong attributes were selected as irrelevant. The detection of irrelevant attributes is an important problem in machine learning, but one that has not yet been examined in the context of action-model learning. We discuss this issue further in Section 7.2.

The above simplification methods work well when applied to discrete domains, but generally fail completely when applied to domains containing real-valued attributes. Consider the flight simulator domain. Once the plane has taken off, its altitude will be constantly changing. (This is pretty much true even if the plane is attempting to engage in level flight.) However, if we assume that the real-valued attributes are continuous, which is not an unreasonable assumption in most domains, then most of the intermediate states encountered during an experience are not relevant. If the altitude of the plane changed from 100 feet to 200 feet over some interval of time, we can assume that it actually took on all the intermediate values; it is not important for the learner to know which intermediate values actually occurred at the particular sampling times used during the interval. The algorithm used for simplifying experience records containing real attributes is closely related to the issue of positive instance generation in such domains, and will be described in the next section.

4.3 Generating Positive Instances

Once the agent has obtained a series of experience records, either by observing a teacher or from its own exploration and experimentation module, it needs to transform these state-action sequences into TOP instances for a concept learner. In a STRIPS-like environment, this instance generation process is relatively simple. For each S_i `ai` S_{i+1} triple, the learner looks at its existing database of operators to find the one corresponding to `ai`, or if it allows conditional operators, the one corresponding to the execution of `ai` in s_i. If an operator is found, the learner will make a prediction about the state after the operator is executed, which can be compared to the actual post-state description S_{i+1}. If the predicated and actual states match, the learner has observed a positive instance of that operator, otherwise it has observed a negative instance. (If no operator is found, one can be generated in the obvious manner.)

In environments that cannot easily be described using the STRIPS operator formalism, the instance generation task is more complex. The remainder of this section examines the generation of positive instances of TOP preimages from experience records in such non STRIPS-like environments.

4.3.1 Recording Positive Instances from Experience Records

We begin with the assumption that effects that occurred during the execution of an action are actually reliable effects of that action. This may not be an accurate assumption, as they may have been caused by other agents in the environment or simply by random chance. Such incorrect attributions can initially lead the agent to develop "optimistic superstitions" such as "going to Room131 will cause Door1 to be open when I return." These superstitions will initially produce operators that do not work in the real world. However, TRAIL is robust in the face of such operators: if TRAIL uses such an optimistic operator in a plan, the plan will fail and TRAIL will realize that the operator is in fact incorrect. (Details of this process are covered in Section 4.4.) This process is an inevitable part of learning in noisy environments, as the agent cannot usually determine from a single observation whether an effect was actually caused by the agent's actions. The only way to determine whether an effect is reliable is through repeated observations, and the use of the operator in later plans will produce such repeated observations.

Thus, given any state-action-state triple S_i a S_{i+1}, any difference δ between S_i and S_{i+1} is considered as an effect of the action a. We call this observation a *positive occurrence* of the TOP $\xrightarrow{\mathsf{a}}\delta$. Now, we note that s_i is a state such that application of action a led to the effect δ. Therefore, if we assume that the observation is reliable, s_i is in the preimage region of the TOP. Now suppose that a was also executed for some time prior to state s_i:

S_{i-k} $\mathsf{a} \ldots S_{i-1}$ a S_i a S_{i+1}

If δ does not hold in s_{i-k}, then we can also say that s_{i-k} is a state such that application of a caused δ to become true. Therefore, s_{i-k} is also in the preimage region of the TOP $\xrightarrow{\mathsf{a}}\delta$. By an identical argument, the descriptions of all states in the set $s_{i-k} \ldots s_i$ in which δ does not hold are recorded as *positive instances* of the preimage condition for the TOP.[2] Note that the *positive occurrence* of the TOP is the entire section of the execution record during which a was executed, ending at state s_{i+1}, while the *positive instances* are the individual state descriptions $S_{i-k} \ldots S_i$, which will later be

[2]If the TOP $\xrightarrow{\mathsf{a}}\delta$ did not already exist, a new TOP is created with action a and effect δ.

State	State Description	Action
s_0	$FacingBar(Bar2) \wedge BotHolding(Bot3, Bar4) \ldots$	`forward`
s_1	$FacingBar(Bar2) \wedge BotHolding(Bot3, Bar4) \ldots$	`turn`
s_2	$BotHolding(Bot3, Bar4) \ldots$	`turn`
s_3	$BotHolding(Bot3, Bar4) \ldots$	`turn`
s_4	$FacingDoorway(Door1) \wedge BotHolding(Bot3, Bar4) \ldots$	`turn`
s_5	$FacingDoorway(Door1) \ldots$	`turn`
s_6	$FacingDoorway(Door1) \wedge FacingBar(Bar1) \ldots$	`forward`

Table 4.1: A State-Action Sequence

used in learning.

As an example, consider the (simplified) state-action sequence shown in Table 4.1. The intended effect of the `turn` action in the sequence appears to be the condition $FacingBar(Bar1)$. Thus, the sequence of states $s_1 \ldots s_6$ is a positive occurrence of the TOP $\overset{\texttt{turn}}{\rightarrow} FacingBar(?x)$, and the state descriptions $S_1 \ldots S_5$ are positive instances for the preimage. Meanwhile, there are also a number of other effects that occur during the `turn` action. For instance, the predicate $FacingDoorway(Door1)$ becomes true while the bot is turning, which may produce another valid TOP. On the other hand, the predicate $BotHolding(Bot3, Bar4)$ becomes false during the turn. Therefore, TRAIL identifies the state descriptions $S_1 \ldots S_4$ as positive instances of the TOP $\overset{\texttt{turn}}{\rightarrow} \neg BotHolding(?x, ?y)$. These positive instances may cause TRAIL to develop a "superstition" about the possible effects of the `turn` action, but such a superstition will eventually be unlearned when TRAIL attempts to use the TOP (unsuccessfully) in some future plan.

When we observe a positive occurrence of a TOP, we also record the time that the action takes to achieve the postcondition. This is the elapsed time between the earliest state in which a was being executed and δ did not hold and the state s_{i+1} in which δ became true. As TRAIL observes multiple positive occurrences, it maintains a record of the mean and variance of these execution times. TRAIL will later need to know the average execution time for a TOP, in order to detect certain types of TOP failures, as discussed in Section 4.4.1.

The above process considers all effects that occur during the execution of `a` as candidates for positive instances of some action model. One might also consider only those effects of a that occurred during the last time step in which a was executed, since these are presumably the "most important" effects of a. After all, the agent stopped taking the action after these effects occurred. However, intermediate effects often occur during execution that may be useful in future plans. For instance, if a robot is turning to face a doorway, it might turn past an electrical outlet along the way. Thus, the predicate $FacingOutlet$ would be false in s_0, would become true in some intermediate state s_i, and would be false again in s_n. If the learning mechanism within the robot were to examine the intermediate states, it might notice the power outlet and be able to find it more quickly next time it is in the room.

In the current implementation of TRAIL, there is one case in which it does not consider all of the effects of an action a. If TRAIL has planned a TR tree for some goal, and this tree is successfully executed without any unexpected transitions in the tree, then it will often be the case that no relevant surprises occurred during execution. Therefore, in the interest of efficiency, TRAIL only generates positive occurrences of the TOPs that were actually used in planning the tree nodes that were executed. Since these TOPs are recorded along with the actions in the experience record, TRAIL can reinforce the correct TOPs without searching all pairs of states encountered during the execution of a, as would be done by the positive instance generation process described above. We believe that this heuristic speeds up the learning process, but at present, the overall effect of the heuristic is still a subject for further investigation.

4.3.2 Positive Instances in Real-Valued Domains

As was observed in Section 4.2.2, the definition of a state difference may need to be refined if the agent is in an environment containing real-valued attributes. Suppose the agent is executing some action a in the flight simulator that causes the altitude to increase by one foot each time step for some interval of n time steps. Then each new value of the altitude will lead to a separate positive occurrence of the TOP $\xrightarrow{\mathsf{a}} Altitude(?x)$. Since all previous states in the execution of a will be positive instances of this TOP, there will be a total of $n(n-1)/2$ positive instances generated. This will generally be far more instances than we actually need for learning.

Therefore, if the effect of a TOP is to change the value of a real-valued attribute (i.e. a real-valued argument of some domain literal), TRAIL only creates a positive occurrence of the TOP when either (i) the agent stops taking the action or (ii) the sign of the derivative of the attribute in question changes. Thus, if an action a causes the plane's altitude to change smoothly from 100 feet to 150 feet to 120 feet, TRAIL will produce two positive occurrences of the TOP $\xrightarrow{a} Altitude(?x)$, one expressing the change in $?x$ from 100 feet to 150 feet and one expressing the change from 150 feet to 120 feet. This simplification considerably reduces the number of instances generated, without losing significant information.

In cases where the positive occurrences themselves occur over long periods of time, the number of instances can be reduced still further by taking only a sampling of the intermediate states of a positive occurrence. In the case where the differences between these intermediate states are only small changes in numerical attributes, state sampling will also not result in the loss of significant information. TRAIL makes use of this sampling strategy to reduce the number of instances generated when learning in the flight simulator domain.

Finally, we may also wish to have some mechanism to differentiate between significant and insignificant changes in a real-valued attribute. If a boolean condition such as $Holding(Bar1)$ changes from true to false as a result of an action, this is likely to be a significant change. However, if the altitude of the plane changes from 200 feet to 202 feet during an action a, this is most likely an insignificant, coincidental change. In particular, we probably do not want the agent to plan to use action a in some future situation in which the plane needs to climb!

Therefore, we have allowed the domain designer to specify optional "tolerance values" for each real-valued attribute V. These tolerance values have two effects on the generation of positive instances for any TOP that has the effect of changing V:

- If the value of V changes by less than the tolerance value, no positive occurrence of the TOP is generated.

- If the value of V is increasing over an interval, and during some subinterval it decreases by an amount less than the tolerance, this decrease is ignored (and vice versa if the value of V is decreasing over the interval.) For instance, suppose the altitude of the plane increases from 100 feet to 150 feet, then down to 143 feet, then up to 200 feet. Unless the tolerance value of the altitude attribute is less than 7 feet, only one positive occurrence of the TOP $\xrightarrow{\mathsf{a}} Altitude(?x)$ will be generated, with $?x$ bound to the value 200 feet.

Both of these heuristics serve to eliminate many of the extraneous positive instances that can be generated during learning.

4.4 Generating Negative Instances

Just as positive instances of TOP preimages are generated from observations of action successes, negative instances of TOP preimages are generated from observations of cases where the action did not have some anticipated effect. There are two ways in which a learner can observe such failures. First, it may observe the execution of a teleo-reactive tree that was planned using a set of TOPs. If any node in the tree does not achieve its intended purpose, the TOP used in the generation of that node has failed. (Recall that each node in the tree was generated by the regression of its parent condition through some TOP.) Second, the learner may observe the actions of a teacher and attempt to predict the effects of the teacher's actions, according to its set of TOPs. If some action does not have its expected effect, then the TOP used to predict the effect may be incorrect. Most of this section considers the first case; the second case will be covered in Section 4.4.5.

4.4.1 Failure in Teleo-Reactive Trees

Given a plan, in the form of a TR tree, consider the problem of detecting failures in the execution of the plan. In the case of STRIPS operators, after each operator is executed, we can simply observe whether the expected result has occurred. However, execution of a plan node that uses a durative action is more complicated. The procedure for executing a node in a TR tree can be stated as "Continue executing the action a until some goal is achieved, unless the activating condition becomes false first." Given this definition, there are two distinct ways in which the execution of a node can fail.

First, the activating condition may become false before the goal condition becomes true. We refer to this case as an *activation failure*. Second, the activation condition may remain true indefinitely while the goal condition never becomes true. This case is referred to as a *timeout failure* as it is detected when the action is executed for significantly longer than the average execution time of the TOP without success.

More formally, consider the general form of a TR node pair, as shown in Figure 4.2. Let G be the condition that the action a is intended to achieve and P be the current estimated preimage condition for the TOP $\xrightarrow{\mathsf{a}} G$. There may also be other conditions that the planner expects to be true throughout execution (which were achieved earlier in the plan and will be necessary later.) Call this set of conditions M (for *maintenance* conditions.) Now the goal of the action is precisely $G \wedge M$, as the purpose of the action a is to achieve G while maintaining M. Similarly, the activating condition for the node is $P \wedge M$. If M becomes false during execution of a, the planner's independence assumption has failed, while if P becomes false before G becomes true, the agent is no longer in the preimage region of the TOP and so should not expect to be able to successfully achieve the TOP goal condition G.

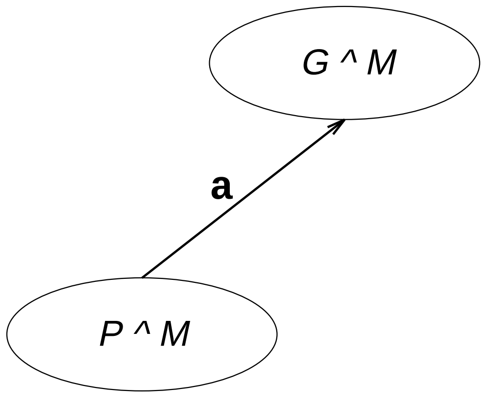

Figure 4.2: The General Case of a TR Node Pair

For any world state s and condition λ, define $s \models \lambda$ to hold iff λ is satisfied in s.[3] Now, for any execution of a durative plan node, let s_e be the state in which execution began. Due to the properties of the TR execution mechanism, we are guaranteed

[3]We borrow the $\models$ symbol for this nonstandard usage.

effect	#	*actual condition*	*cause*
$s_f \not\models P$ (activation failure)	1	$s_e \models \pi$, $s_f \not\models \pi$	random execution failure
	2	$s_f \models \pi$	estimated preimage too specific
	3	$s_e \not\models \pi$, $s_f \not\models \pi$	estimated preimage too general
$s_f \not\models M$ (activation failure)	4	$s_e \models \pi \wedge M$, $s_f \not\models \pi$	random execution failure
	5	$s_e \models \pi \wedge M$, $s_f \models \pi \wedge \neg M$	$\neg M$ a side effect
	6	$s_e \models M \wedge \neg\pi$	estimated preimage too general
$s_f \models P \wedge M$ (timeout)	7	$s_e \not\models \pi$	estimated preimage too general
	8	$s_e \models \pi$, $s_f \not\models \pi$	random execution failure
	9	$s_e \models \pi$, $s_f \models \pi$	timeout detected too soon

Table 4.2: Possible Explanations For a TR Node Failure

that $s_e \models P \wedge M$. Finally, let s_f be a state in which a failure was detected. As stated above, this failure detection could occur either because the activation condition $P \wedge M$ became false in s_f, or because the executor decided that the node had been active for too long without success.

Given each case of plan node failure, we can analyze the possible causes of the failure, as summarized in Table 4.2. Using the notation defined above, an activation failure corresponds to the situation where $s_f \not\models P \wedge M$ and $s_f \not\models G \wedge M$, while a timeout failure occurs when the plan node would still be active ($s_f \models P \wedge M$) but the executor has been executing a for too long without success.

For the remainder of this chapter, we will let π denote the true preimage of the TOP $\xrightarrow{a} G$. π may or may not be equal to P. Recall that the *preimage region* for the TOP is the set of states for which the preimage condition π holds.

4.4.2 Analyzing Activation Failures

We first consider the case of an activation failure (the first six lines of Table 4.2). Since $s_f \not\models P \wedge M$, there are two general cases: either $s_f \not\models P$ or $s_f \not\models M$ (or both). If $s_f \not\models P$, one of the following three specific cases must hold:

1. $s_e \models \pi$, $s_f \not\models \pi$. Since $s_e \models \pi$, the TOP should have worked, but a random execution failure apparently prevented the action from succeeding.

2. $s_f \models \pi$. The agent has reached a state that is not in the estimated preimage region of the TOP but is in the true preimage region. Thus, the current TOP preimage estimate P is over-specialized and needs to be generalized to include s_f.

3. $s_e \not\models \pi$, $s_f \not\models \pi$. Here the agent began in a state that was not in the true preimage region. Since by definition $s_e \models P$, the current TOP preimage is over-generalized and needs to be specialized to exclude s_e.

On the other hand, if $s_f \not\models M$, one of the following holds:

4. $s_e \models \pi \wedge M$, $s_f \not\models \pi$. Since $s_e \models \pi$, the TOP should have worked, but again a random execution failure apparently took the agent outside the true preimage region. M may or may not be a legitimate delete list element of the TOP.

5. $s_e \models \pi \wedge M$, $s_f \models \pi \wedge \neg M$. The agent is still in a state from which G can be achieved, but the maintenance condition M has become false. Thus M is a legitimate delete list element of the TOP.

6. $s_e \models M \wedge \neg\pi$. In this case the agent began in a state outside the true preimage region, so the estimated preimage P is over-generalized. As in case 4, we cannot accurately determine the true effect of the action on M.

Once an activation failure has occurred, it can be split into one of the above two groups based on the conditions that hold in s_f. However, the appropriate correction to the TOP depends on which specific case applies. In case 2 of the above list, the preimage needs to be generalized, while in case 3 it needs to be specialized. Similarly, case 6 requires the preimage to be specialized, while case 5 requires a side effect to be added to the TOP. In the noise cases, 1 and 4, the preimage ideally should not be changed at all.

Thus, an activation failure in a TR tree suggests that the agent experiment in order to attempt to discover the cause of the failure. A simple form of experimentation can be used to decide whether the true preimage condition π holds in the final state s_f. All the agent needs to do is continue executing a from s_f to see if G eventually becomes true. If it in fact does, then the agent knows that it is in either case 2 or case 5, depending on whether P or M holds.[4]

In the case where π does not hold in s_f, it is more difficult to determine whether this is the result of π not holding in the starting state s_e or the result of a noisy execution failure. The obvious solution is to repeatedly execute the TOP in s_e to determine whether the failure is reliable. However, even aside from the obvious computational expense of such experimentation, perceptual aliasing in the domain may cause some difficulty: simply duplicating the situation may duplicate some hidden condition that caused the failure but that is not common to all states equivalent to s_e. Thus, in order to accurately determine whether s_e belongs in the preimage region of the TOP, the TOP must be executed in a series of representative situations equivalent to s_e. We have not yet developed a satisfactory general mechanism for doing this experimentation. Therefore, TRAIL simply considers all cases in which π does not hold in s_f as evidence that the TOP needs to be specialized. Thus, all such cases will result in additional negative instances for the concept learner, which is not unreasonable as it reflects the fact that the TOP is not completely reliable.

There is one case of activation failure that deserves special mention here. That is the case where the agent gets itself into a state from which it is clear the goal cannot be achieved, for instance if the simulated plane being controlled by the learner crashes into the ground. Inspired by the airplane example, we call such states *crash* states. Assuming that there is some way in which the agent can recover from such a situation, it is useful to record the states that led up to the crash state. First, these states are very high-confidence negative instances, compared to others that might be incorrectly labeled as a result of incorrect preimages or insufficiently long timeout periods. Second, if possible we would like the learner to put extra weight on excluding these states from the preimage, since we wish our agents to avoid such states even if

[4]Since experimentation is computationally expensive, TRAIL actually experiments only when a node fails multiple times without leading to a successful revision of the plan. The exact criteria for experimentation depend on the estimated reliability of the TOP in question.

they only lead to crash states with low probability.

4.4.3 Analyzing Timeout Failures

Next, we examine the other failure case, that of a timeout failure. Timeout failures are normally detected by comparing the current execution time of an operator to the average execution time for that operator. If the time used is more than three deviations above the average, TRAIL detects a timeout condition.

Given a timeout failure, there are three possible causes:

7. $s_e \not\models \pi$. Thus, execution may well continue indefinitely without success if a timeout is not imposed.

8. $s_e \models \pi$, $s_f \not\models \pi$. This is a random execution failure similar to cases 1 and 4 above.

9. $s_e \models \pi$, $s_f \models \pi$. In this case, execution is proceeding correctly, but failure has occurred because the timeout period is insufficiently long.

Unfortunately, the only way to distinguish case 9 from the other cases is to continue execution, which is equivalent to simply lengthening the timeout period. Since the agent cannot continue lengthening the timeout period arbitrarily, it must for the moment simply assume that the timeout period is sufficiently long and accept such cases as sources of noise. However, we have provided a mechanism for TRAIL to recover from false negative instances caused by overly short timeout periods. Suppose that TRAIL has observed a timeout failure of length t for a TOP τ. If it later observes a sufficient number of longer executions of τ such that the mean execution time for τ increases above t, it will assume that the earlier failure was actually an instance of failure case 9, and should be removed from the list of negative instances for τ.

The other two timeout failure cases, cases 7 and 8, could be distinguished by repeated experimentation of the same sort described earlier, but as in the earlier cases, the possibility of perceptual aliasing would require the system to execute the TOP in a variety of situations representative of s_e. Therefore, TRAIL does not do experimentation in the case of a timeout failure, but rather automatically considers all timeout failures as sources of negative training instances.

4.4.4 Generating Negative Instances from Plan Failures

Once a TOP failure is detected, it must be converted into a set of negative instances for the learner. In the case of a timeout failure, this process is very similar to the positive instance generation process described earlier. The action is described in an experience record consisting of a state-action sequence S_e $\mathsf{a} \ldots \mathsf{a}$ S_f. Since this sequence represents a failure of the TOP, each state description in the sequence is recorded as a negative instance of the TOP preimage.

If the failure was an activation failure, the learner is faced with a more complex task. As described earlier, when an activation failure is detected, the agent will sometimes experiment by continuing to execute the action. If the goal node of the TOP eventually becomes true, the agent can simply identify a positive occurrence of the TOP as described in Section 4.3, recording delete list elements where appropriate. If the goal node of the TOP does not become true, then the agent must consider whether the preimage condition P and the maintenance condition M hold. If the preimage condition P is false, the TOP has failed and the states in the experience record can be recorded as negative instances. If the maintenance condition M is false, then some literal in M has become false and should be recorded as a delete list element of the TOP. Thus, delete list elements can be identified even if the TOP has not succeeded.

A simple example in Botworld should make clear the need for identifying delete list elements even in the case of TOP failures. Suppose that TRAIL is trying to achieve the condition $AtGrabbingDist(Bar1) \wedge OnMidline(Bar1)$ and constructs the plan shown in Figure 4.3. When this plan is executed, TRAIL will achieve $AtGrabbingDist(Bar1)$, turn until it is $ParallelTo(Bar1)$, and then begin moving forward in hopes of achieving $OnMidline(Bar1)$. Of course, this will fail immediately. Therefore, TRAIL needs to learn from this failure that $AtGrabbingDist(?x)$ is a delete list element for the TOP $\xrightarrow{\texttt{forward}} OnMidline(?x)$.

In the interest of performance efficiency, we place one limit on the identification of

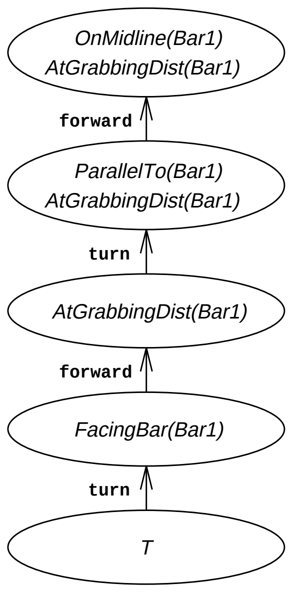

Figure 4.3: A Plan That Fails Due to a Side Effect

activation failures. If a TOP is known to be unreliable, it is inefficient to do relearning, and possible experimentation, every time the TOP fails. Instead, we allow TRAIL to continue executing the plan, while maintaining a count of the number of times each node has failed. If a node fails repeatedly, we can then assume that the failures are likely to be due to a planning error rather than an execution error, and thus should be further examined. We can use the plan node success rate, as described in Section 2.4.1, as an estimate of the reliability of the TOP, and thus call the learning module only if we are sure to some confidence level that the failures are not merely execution errors.

4.4.5 Generating Negative Instances from a Teacher

Finally, let us return to the case where the agent is learning by observing some training agent rather than acting on its own. Here, learning is constrained by the fact that the agent has no access to the plan, if any, that is being executed. Also, since the agent is only observing the training agent, it will not be able to perform experiments immediately upon failure detection.

Given these limitations on the learner, we approach the negative instance generation problem by defining a *negative occurrence* of a TOP $\xrightarrow{\mathsf{a}}\lambda$ as any state-action sequence in which action a was executed and the expected effect λ did not occur during execution of a. Even without access to the plan being executed by the teacher, TRAIL can observe such negative occurrences while the teacher is achieving some goal.

As described in Section 4.4.1, TOP failures can be divided into two categories, activation failures and timeout failures. Given a state-action sequence, activation failures can only be defined with respect to the estimated preimage of a particular TOP $\xrightarrow{\mathsf{a}}\lambda$. If the estimated preimage of the TOP holds at some point during the execution of a but does not hold at some later point, the TOP has failed. Timeout failures for the TOP, however, can be defined more generally: any sequence in which a is executed for a sufficiently long time without achieving λ is a timeout failure.

However, we do place one limitation on the identification of timeout failures. Observe that for most operators, the preimage region only covers a very small portion of the state space. Therefore, if the learner generates a negative occurrence of the TOP $\xrightarrow{\mathsf{a}}\lambda$ every time a is executed and λ does not occur, it will quickly generate a large number of negative instances. Instead, TRAIL will identify a timeout failure only if it expected λ to occur when it did not, i.e. some state in the execution record is within the current estimated preimage region of the TOP.

Thus, when the training agent begins executing an action a, the learner examines all TOPs that involve action a, and labels as *active TOPs* the subset for which the preimage condition P is currently satisfied. Each of these active TOPs indicate that the learner expects the postcondition literal for that TOP to become true if a is executed for a sufficiently long time. If this expectation fails, the learner can identify a negative occurrence of the TOP.

In general, there are four possible outcomes for each active TOP $\xrightarrow{\mathsf{a}}\lambda$ during a teacher observation:

- The postcondition literal λ may become true while a is executed. In this case, the intermediate states will be identified as positive instances as described in Section 4.3.

- The estimated preimage condition P may become false during execution, without λ ever becoming true. The learner can thus assume that the TOP has failed and record a negative occurrence.

- The action a may be executed for longer than the timeout period for the TOP, at which point the learner can again identify a negative occurrence of the TOP.

- The teacher may switch to a different action before any of the other three cases has occurred. In this case there is little useful we can learn about the preimage of the TOP $\xrightarrow{\mathsf{a}}\lambda$ from the observation, although we may be able to identify delete list elements of the TOP.

Once the learner has observed a negative occurrence of a TOP $\xrightarrow{\mathsf{a}}\lambda$, it will record negative instances in the same way in which positive instances are recorded from positive occurrences. Given a state-action sequence

S_i a S_{i+1} $\mathsf{a} \ldots \mathsf{a}$ S_{i+k}

in which a failure was detected in state s_{i+k}, and $s_{i+j} \models P$ for all $0 \leq j < k$, then each s_{i+j} is a state in which P held but the expected effect λ did not occur. Therefore, each of the state descriptions S_{i+j} can be recorded as a negative instance for the preimage condition of $\xrightarrow{\mathsf{a}}\lambda$. This process is otherwise identical to the positive instance generation process discussed in Section 4.3.

The above instance generation process assumes that the learner has an approximation P of the preimage condition at the time at which it is generating negative instances. This assumption presents a problem when a TOP is initially generated, as it will not have any initial preimage estimate. Rather than assume that the preimage condition is T, TRAIL instead uses a two-step learning process to learn the preimage when it initially creates a TOP from an experience record. TRAIL first generates positive instances as described in Section 4.3, then learns a preimage condition based on those positive instances, and finally uses this condition as an initial preimage estimate for generating negative instances. This method of course assumes that the learner can generate a useful concept from positive instances only, a problem that will be discussed in Section 5.5.5.

4.5 Instance Generation in Other Systems

Most of the previous work on action-model learning has focused on the generation of positive instances. LIVE, EXPO, and OBSERVER all used STRIPS-like action representations, which largely avoid the need for the type of failure analysis discussed in this chapter. Instead, this earlier work concentrated on exploration and experimentation, two areas that were largely ignored in TRAIL. DeJong's continuous domain planner uses a more expressive model of actions which addresses some of the same issues addressed by TRAIL, but his representation is very different from that used in TRAIL. This section discusses the instance generation process in these four systems and their differences from TRAIL.

4.5.1 Instance Generation in LIVE

In Shen's LIVE system (Shen 1994), instance generation can be roughly divided into three sources: exploration, execution, and experimentation. Since the LIVE environments are generally STRIPS-like, action traces correspond exactly to the STRIPS-like execution traces discussed in Section 4.2, and learning is done based on the differences between successive states.

Exploration in LIVE arises from situations in which LIVE is unable to find an applicable action model to achieve some goal it has been given. If this situation arises, LIVE tries random actions according to several heuristics, particularly focusing on those actions that do not yet have any known effects. If one of these random actions has an effect in the environment, a positive instance is generated, otherwise no instances are generated.

Execution refers to situations where LIVE's planner has constructed a plan and the plan fails to have its intended effect. Since actions in LIVE are atomic units that have a set of predicted effects, each action in the plan either succeeds or fails. If an action does not have its expected effects, LIVE detects a "surprise" and essentially generates a negative learning instance.

Experimentation in LIVE, like exploration, arises from planning failures. If LIVE's planner encounters either an infinite regression or a situation where two subgoals are mutually interfering, it assumes that one of the models used in planning must be incorrect. Since planning failures arise from over-specialized preconditions, LIVE in this situation creates an experiment in which the action can be applied in a state where it is not expected to succeed. If the action does succeed, LIVE again detects a surprise, this time a positive instance that can be used to generalize the model's precondition. Note that TRAIL does not have any similar mechanism for doing experimentation to correct overly specific operator preconditions.

4.5.2 Instance Generation in EXPO

The example generation process in Gil's EXPO system (Gil 1992) can also be divided up into exploration, execution, and experimentation. When EXPO executes an operator, it observes the effects of the operator. If the operator has the expected set of effects, it is not changed. If one or more of the expected effects does not occur, EXPO generates a negative instance of that operator. EXPO then uses its experimentation process, described below, to correct the precondition. Finally, if an unexpected effect of an existing operator is observed, a similar experimentation process is used to construct a model for conditional effects of that action.

Experimentation is the primary emphasis of the EXPO system. The main use of experimentation in EXPO is in correcting operator preconditions. Once EXPO has observed a negative instance, it is compared with previous successful applications of the operator to generate a set of differences. EXPO then experiments to determine which of these differences should be added to the existing precondition. In each experiment, EXPO plans to achieve a state in which some of the differences hold, then applies the operator and observes the results. Thus, a positive or negative instance is generated from the experiment and eventually used to update the precondition.

Only a few of EXPO's learning methods could really be classified as exploration, and even those methods are described as another form of experimentation. A typical example is the generation of new operators by direct analogy: once EXPO has seen an operator have some effect, it can try the same action using different, but related, objects. Gil gives an example where EXPO initially has an operator to drill with a high helix drill bit, then experiments to determine what happens if it applies the same action using a twist drill bit instead.

4.5.3 Instance Generation in OBSERVER

Wang's OBSERVER system (Wang 1995*b*) is probably the most similar to TRAIL in terms of learning methodology. OBSERVER also learns both from problem-solving traces provided by a domain expert and from its own planning and execution. Each of these processes generate instances, which are used to generate preconditions using a learning algorithm similar to version spaces (Mitchell 1982). However, the instance generation process in OBSERVER is considerably simpler than the instance generation process in TRAIL.

OBSERVER generates only positive instances from its observations of the expert. Each state in which an operator was applied by the expert corresponds to a positive instance of that operator. The partially-learned operators are then used to plan solutions to a set of practice problems. During the execution of these plans, OBSERVER generates further instances for learning. If an operator changed the value of any predicate in some state, then that state is identified as a positive instance of the operator, otherwise it is identified as a negative instance. These instances are then used to update the operator precondition, creating conditional effects if necessary in order to explain states where the same operator had different effects.

4.5.4 Instance Generation in DeJong's Continuous Domain Planner

DeJong's (1994) work on learning to plan in continuous domains does not exactly fall under the category of action-model learning, but it does involve learning models of processes, models that are later used in planning. DeJong's explanations are a combination of qualitative models of processes and quantitative models of specific variables. Thus, his explanations are very different from the action models used by LIVE, EXPO, OBSERVER, and TRAIL.

DeJong's system learns explanations from experience traces, which are very similar to the continuous experience records discussed in Section 4.2. The actual learning is done in two phases. First, the learner hypothesizes a qualitative model of the process that matches the observed behavior of the system. Second, it uses curve-fitting techniques to build a quantitative model of the variables involved. This is a very different process from the framework of instance generation and concept learning discussed in this chapter.

As in TRAIL, experience records arise from two sources, observations of a domain expert and execution of plans. The initial experience record serves as a "positive instance" in that explanations are hypothesized based on the record. Once a plan is created from the explanations, there are two types of failures that can occur. A plan will predict values for the variables in the environment. If these predictions are off by small amounts, the learner generates new numeric data points, which the curve fitting algorithm uses to produce a new quantitative model. If a prediction is qualitatively wrong (for instance, a value is increasing when it should be decreasing,) the learner rejects the current qualitative model and generates a model that is consistent with the newly observed behavior as well.

Chapter 5

Learning in TRAIL Using ILP

We now turn to examining the learning problem that results from the instance generation process described in the previous chapter. Recall that the instances generated are descriptions of states of the world. These descriptions are in the form of conjunctions of positive ground literals. For learning purposes, we make the Closed World Assumption, so a number of negated literals are also implied by each description. Each described state is labeled as a positive or negative instance of a particular TOP, indicating whether TRAIL considers it to be part of the preimage region or not. TRAIL can then use various concept learning techniques to estimate the preimage of the TOP. The exact format of these instances, and the methods that TRAIL uses to learn from them, are discussed beginning in Section 5.2.

5.1 Introduction to ILP

Inductive Logic Programming (Lavrač & Džeroski 1994, Muggleton 1992), or ILP, is a specialized subarea of machine learning aimed at learning concepts in domains that are too complicated for traditional concept learning algorithms. Most concept learning algorithms assume that each instance can be described in an *attribute-value* format, using a set of attributes that are either binary or ordinal. In many cases, learning algorithms have been extended to handle continuous attributes as well, but the more complicated cases, where the instances are described in first-order logic, are still beyond attribute-value machine learning algorithms.

A few examples should make this point clear. First, consider a a database of family relationships, such as $Father(John, Mary) \wedge Mother(Ann, John) \wedge \ldots$ Suppose a learner is attempting to learn the concept of $Grandmother(?x, ?y)$. (This problem is obviously quite artificial, but it is a standard conceptual example found in the ILP literature.) The learner is given the database of relationships, a set of positive instances, such as $Grandmother(Ann, Mary) \wedge \ldots$, and a set of negative instances, such as $\neg Grandmother(John, Mary) \wedge \ldots$. From this set of instances, it needs to learn a description of the concept $Grandmother$. An attribute-value learner would only be able to induce concepts in terms of the arguments of $Grandmother$, which is clearly insufficient for this problem. In order to learn a description of $Grandmother$, the learner needs to introduce a new variable $?z$ that represents the intermediate generation between grandmother and grandchild. Furthermore, it must also be able to specify relations among arguments and variables, such as $Mother(?x, ?z)$, which is impossible in most decision tree inducers or neural network learners.

Our second example is more directly related to action-model learning. Suppose we have a number of states from the Office Delivery domain described in first-order logic. Each state is a case where an operator is known to have succeeded or failed. A plausible scenario consisting of several such instances is shown in Table 5.1. We wish to learn a concept that covers only the successful instances and none of the failures. In a way, this problem is similar to the standard concept learning problem, as each instance is an independent list of literals. But as in the $Grandmother$ example, we cannot simply give the problem shown in Table 5.1 to an attribute-value concept learner and expect it to learn. The literals in each instance cannot easily be converted to propositional or numeric features. The problem is that the literals are *structured* - the arguments of a literal refer to objects in the world, and an object can be an argument of several different literals, all of which may be relevant to the learning problem. Such instances can, in many cases, be transformed into propositional instances, but this process is non-trivial, and will be examined further later in this chapter.

Inductive Logic Programming is a technique that is designed to solve this type of structured learning problem. A very readable introduction to ILP can be found

Success: Robot was able to make 7 copies of Article1
$At(Robot, Room20) \wedge At(Xerox2, Room20) \wedge Has(Robot, Article1) \wedge$
$Mode(Xerox2, Copy) \wedge Paper(Xerox2, 100) \wedge \ldots$

Success: Robot was able to make 5 copies of Article2
$At(Robot, Room30) \wedge At(Xerox3, Room30) \wedge Has(Robot, Article2) \wedge$
$Mode(Xerox3, Copy) \wedge Paper(Xerox3, 150) \wedge \ldots$

Failure: Robot was not able to make 4 copies of Article2
$At(Robot, Room27) \wedge At(Xerox2, Room20) \wedge At(Xerox3, Room30) \wedge$
$Has(Robot, Article2) \wedge \ldots$

Failure: Robot was not able to make 5 copies of Article1
$At(Robot, Room30) \wedge At(Xerox3, Room30) \wedge Has(Robot, Article1) \wedge$
$Mode(Xerox3, Copy) \wedge Paper(Xerox3, 1) \wedge \ldots$

Table 5.1: Instances For Learning an Operator for `copy`

in Lavrač & Džeroski (1994), while a collection of ILP-related papers is found in Muggleton (1992).

5.2 Limited ILP for Action-Model Learning

The basic definition of ILP is as follows: given a set of labeled training instances E (called the *foreground knowledge*) and some set of background knowledge B, find a hypothesis H, expressed in some hypothesis language, that correctly predicts the labels for most or all of the training instances.

A hypothesis normally consists of one or more *clauses*, each of which has the target concept as a consequent. We say that an instance $e \in E$ is *covered* by a clause C with respect to background knowledge B if and only if $B \cup C \models e$. An instance is covered by a hypothesis if and only if it is covered by at least one of the clauses in the hypothesis.

In the *Grandmother* example from the previous section, a hypothesis that might be learned is the following:

C_1: $Grandmother(?x, ?y) \leftarrow Mother(?x, ?z) \wedge Mother(?z, ?y)$

C_2: $Grandmother(?x, ?y) \leftarrow Mother(?x, ?z) \wedge Father(?z, ?y)$

The positive instance $Grandmother(Ann, Mary)$ is covered by the clause C_2 in the above hypothesis. Note that the two background facts $Mother(Ann, John)$ and $Father(John, Mary)$ together with the clause C_2 logically entail the positive instance, $Grandmother(Ann, Mary)$.

In ILP, then, our goal is to find a hypothesis H such that, given the background knowledge B, all of the positive instances E^+ are covered by the hypothesis H while none of the negative instances E^- is. If we allow the concept to be inexact (presumably due to the presence of noise in the domain), we will simply require that most of the positive instances and few of the negative instances are covered, using some appropriate measure of the correctness of the concept over the instances.

It is not immediately obvious how this framework of instances and background knowledge corresponds to the delivery domain problem shown in Table 5.1. Here, none of the available knowledge applies across multiple instances, unlike the family relationship database mentioned in the *Grandmother* example. Also, the instances themselves aren't directly in the format of positive and negative literals to be covered.

The conversion of the TOP preimage instances to the standard ILP framework is actually fairly straightforward. Figure 5.1, which shows the ILP conversion process for two of the delivery domain instances from Table 5.1, should help make the steps in this process clear. First, recall that each instance describes a state of the world. We reify this state, as in the situation calculus (McCarthy & Hayes 1970), adding a state argument ST_i to each literal in the description of instance S_i. The literals describing each instance (with state arguments included) form the background knowledge for the ILP learning problem.

Next, we need to introduce the predicate to be learned. This predicate is called $InPreimage(ST_i)$ and is labeled as either positive or negative for each ST_i depending on whether instance S_i was a success or a failure. This set of $InPreimage$ literals will form the foreground knowledge E for the Inductive Logic Programming problem.

Next, we note that by examining the action and intended effect of each instance, we can determine a set of bindings for the TOP variables of the TOP that is being learned (see Section 2.3). Consider the examples in Table 5.1, which describe the use of the `copy(?obj,?x)` action to achieve the predicate $Has(Robot, copies(?n, ?obj))$ in several situations.[1] The TOP variables for this TOP are $?n$, $?obj$, and $?x$, and each of these variables may be bound differently in each instance, according to the action taken and the intended effect in that instance. In order to be able to generalize over these instances, the induced concept for the *InPreimage* predicate must be able to refer to these TOP variables. Therefore, each *InPreimage* predicate contains as arguments the bound values of the TOP variables in the instance, as well as the state argument associated with the instance.[2] The intended semantics of the literal $InPreimage(X_1, X_2, \ldots, X_n, ST_i)$ is that if the TOP variables are bound to the values $X_1, X_2, \ldots, X_n$ then the state represented by the state argument ST_i is in the preimage region of the TOP. The learned concept for the *InPreimage* predicate can be converted to the preimage of the TOP by simply dropping the state argument from each literal and taking the disjunction of all of the clauses in the concept. Again, Figure 5.1 should help make this process clear.

The learning problem that results from this conversion differs in several ways from the general ILP learning problem. First off, the background knowledge (which describes the observed positive and negative instances) is partitioned by the introduced state argument; each background literal pertains to one and only one foreground *InPreimage* instance. We can use this fact to essentially disregard the state arguments while learning, as described more fully in Section 5.4. Furthermore, all of the background knowledge is guaranteed to consist of ground literals, since each instance describes one particular state of the world. We also observe that the learned concepts for *InPreimage* will not contain other *InPreimage* literals, so there is no need for the various techniques used in recursive ILP. On the other hand, it is important to

[1]Note that $?n$ is not an argument of the `copy` operator; `copy` is a durative action that can be continued until the proper number of copies are made. $?x$ refers to the Xerox machine on which the copies are being made.

[2]The order of the arguments is not relevant as long as they are consistent across instances.

The TOP:

Action: `copy(?obj,?x)`
Postcondition: $Has(Robot, copies(?n, ?obj))$

Instances seen by learner:

`copy(Article1,Xerox2)` achieves $Has(Robot, copies(7, Article1))$ in:
$At(Robot, Room20) \wedge At(Xerox2, Room20) \wedge Has(Robot, Article1) \wedge$
$Mode(Xerox2, Copy) \wedge Paper(Xerox2, 100) \wedge \ldots$

`copy(Article2,Xerox2)` fails to achieve $Has(Robot, copies(4, Article2))$ in:
$At(Robot, Room27) \wedge At(Xerox2, Room20) \wedge At(Xerox3, Room30) \wedge$
$Has(Robot, Article2) \wedge \ldots$

Background facts:

$At(Robot, Room20, ST_1)$	$At(Xerox2, Room20, ST_1)$	$Has(Robot, Article1, ST_1)$
$Mode(Xerox2, Copy, ST_1)$	$Paper(Xerox2, 100, ST_1)$	$\ldots$
$At(Robot, Room27, ST_2)$	$At(Xerox2, Room20, ST_2)$	$At(Xerox3, Room30, ST_2)$
$Has(Robot, Article2, ST_2)$	$\ldots$	

Foreground facts:

+ $InPreimage(7, Article1, Xerox2, ST_1)$
- $InPreimage(4, Article2, Xerox2, ST_2)$

Possible learned concept:

$$InPreimage(?n, ?obj, ?x, ?st) \leftarrow Has(Robot, ?obj, ?st) \wedge At(Robot, ?p, ?st) \wedge At(?x, ?p, ?st)$$

TOP preimage condition:

$Has(Robot, ?obj) \wedge At(Robot, ?p) \wedge At(?x, ?p)$

Figure 5.1: The Preimage Learning Process Using ILP

note that the learned concepts for *InPreimage* may need to include interval conditions on the variables used in the concepts. Thus, whatever ILP algorithm we use must be able to handle intervals.

One additional piece of domain knowledge that is often given to ILP learners should be mentioned here, as it turns out to be both practical and very useful for preimage learning. This is the idea of mode declarations of predicates, as used in GOLEM (Muggleton & Feng 1990) and FOIL2.0 (Quinlan 1991). In general, if a predicate is a one-to-one mapping, we can simplify the learning process by viewing it as a function and specifying that the arguments corresponding to the outputs of the function are *output arguments*, while the other arguments are *input arguments*.[3] For example, $At(?x, ?p)$ is a one-to-one mapping giving the location of object $?x$, so we can label $?x$ as an input argument and $?p$ as an output argument. On the other hand, $Near(?p, ?q)$ is a many-to-many mapping expressing whether two locations are near each other, so each argument must be viewed as an input argument. It is also possible to have functional predicates whose arguments are output-only; for instance, the predicate $CurrentTime()$ is clearly an output-only function.

For the purposes of TRAIL, as in a number of other ILP systems, input and output arguments are not precisely defined properties of a domain, but rather a method of providing useful information to the learner. The domain designer can give the system a set of declarations expressing the mode of the arguments of each predicate, which TRAIL uses to simplify the learning process, as described in Section 5.4. TRAIL's learning system will in fact function regardless of the predicate mode declarations. However, it generally learns more effectively if the mode declarations correspond to the intuitive usage of the predicates in the domain.

5.3 Overview of Existing ILP Algorithms

Over the past few years, the field of Inductive Logic Programming has grown significantly. The field is much too large to give a complete overview here, so we instead

[3]Mode definitions can also be used for the purpose of assigning types to variables in Inductive Logic Programming systems. TRAIL's learning system does not do this at present.

present a brief summary of three classic ILP systems. A good textbook introducing the field is Lavrač & Džeroski (1994). Muggleton's (1992) book provides a useful collection of papers, although there have been considerable developments in the field since then, particularly in the area of theory (Frazier & Page 1993, Džeroski, Muggleton & Russell 1993, Džeroski 1995).

5.3.1 FOIL

The first widely publicized ILP learning algorithm was Quinlan's FOIL (1990). FOIL attempts to cover the positive instances in the foreground knowledge by generating a series of clauses via an approach similar to the AQ systems (Michalski 1983). It begins with a set of labeled positive and negative instances and an empty hypothesis. In each cycle of the algorithm, a clause is generated that covers some of the positive instances and as few of the negative instances as possible. After a clause has been added to the hypothesis, the covered positive instances are removed from the instance set. This process repeats until the algorithm decides it has covered sufficiently many of the positive instances. TRAIL uses a similar covering algorithm, discussed in Section 5.5.1.

The added clauses themselves are generated using a greedy information-based search similar to that used in other machine learning algorithms, particularly ID3 (Quinlan 1986). The learner begins the clause search process with an empty clause, which by definition covers all of the instances. It then repeatedly specializes the clause by adding new literals, until either the clause covers no more negative instances or a length restriction is violated (thus preventing the learned clause from becoming too complex and *overfitting* the data.) The new literals are drawn from one of the following categories:

- $L_i(V_1, V_2, V_3, \ldots)$ where L_i is some predicate in the domain and the V_j are variables, including at least one variable that is already included in the head or body of the clause.

- $X_i = X_j$ where X_i and X_j are both input variables, i.e. variables in the head of the clause.

- Negations of one of the above.

The metrics that are used for deciding which literal to add to the clause and for deciding when to stop adding literals to the clause are discussed in more detail in Section 5.5.3. Briefly, literals are chosen that reduce the entropy of the instances covered by the clause, according to a metric known as *weighted information gain.* The process is terminated any time the *encoding length* of the clause becomes longer than the number of bits required to explicitly indicate which positive instances are covered by it.

There are several difficulties with FOIL from the point of view of action-model-preimage learning. First, the expressive power of the learned clauses is limited; FOIL does not allow the learner to include a literal that specifies that a variable must be bound to a particular value (although this feature has been added to a later version of FOIL (Cameron-Jones & Quinlan 1993).)

Second, the various metrics used in FOIL for deciding which literal to add to a clause, when to stop adding literals to a clause, and when to stop adding clauses to a concept do not provide very satisfactory noise handling. This issue is examined in more detail in Section 5.5.3.

Finally, FOIL's greedy search strategy makes it very difficult for it to learn concepts that involve a conjunction of connected literals. For example, consider the precondition of the `copy(?obj, ?x)` TOP described in Figure 5.1. One of the conditions needed for the TOP to succeed is that the robot and the copier must be in the same location. Since the copier is included as an argument of the $InPreimage$ literal, this condition can be expressed within the limitations of the FOIL hypothesis language as something like $At(Robot, ?p) \wedge At(?x, ?p)$. However, it would be very difficult for FOIL to actually learn this conjunction: the literal $At(Robot, ?p)$ holds in every instance (presumably) and so does not provide any discrimination between positive and negative instances. The literal $At(?x, ?p)$ similarly does not provide any useful information. Therefore, neither literal would be added to the clause at any point unless both could somehow be added together. Adding both literals to the clause would require some form of lookahead in the search, greatly increasing the search cost. (This difficulty, known as the *connected literal* problem, is a general problem that plagues many Inductive Logic Programming systems.)

5.3.2 GOLEM

The ILP system GOLEM (Muggleton & Feng 1990) adopts a considerably different approach to Inductive Logic Programming. Like FOIL, GOLEM is based on the idea of creating covering clauses. The learner repeatedly generates single clauses that cover some positive instances and as few negative instances as possible, continuing until a satisfactory number of the positive instances are covered.

Where the algorithms differ is in the method used to construct the covering clauses. Instead of starting with the concept T and specializing by adding literals, GOLEM works by starting with specific positive examples and generalizing them by computing the *Relative Least General Generalization* of the instance and other instances. This is known as a *bottom-up* approach to ILP, as opposed to the *top-down* approach of systems such as FOIL.

In order to explain GOLEM, we need to explain the concept of Relative Least General Generalization (or *rlgg*) in more detail. The concept of *lgg*, or Least General Generalization (Plotkin 1969), is well known in logic. Intuitively, the lgg of two clauses is the most specific clause such that for each clause, there is a substitution that transforms the lgg into a subset of that clause. For instance, the lgg of the two clauses $F(2) \leftarrow G(1,2) \wedge H(2,3)$ and $F(3) \leftarrow G(1,3) \wedge I(2,3)$ is $F(?x) \leftarrow G(1,?x)$.

Given two instances e_1 and e_2 and background knowledge B, the rlgg of the instances is then defined as the least general generalization of the instances relative to the background knowledge. Formally, we have the following:

$rlgg(e_1, e_2) = lgg((e_1 \leftarrow B), (e_2 \leftarrow B))$

Roughly, the rlgg of two instances is an implication in which the head of the implication is a generalization of the instances themselves, and the body consists of all facts that hold for both instances in the background knowledge. In the case where each instance describes a situation, the body of the rlgg thus expresses those facts that are common to the two situations described by the instances. Rlggs tend to be very large in general, especially in cases where there is a large amount of background knowledge. Therefore, they are usually pruned to contain only those facts that are considered to be relevant to the instances. Rlggs are explained in more detail by Muggleton and Feng (Muggleton & Feng 1990); the exact definitions will not be needed here.

As an example of computing an rlgg, consider once again the `copy` operator described earlier. Suppose we have instances describing two states s_0 and s_1 in which the action succeeded. In s_0, the $?x$ variable was bound to $Xerox1$, while in s_1 it was bound to $Xerox2$. The situations are reified and assigned names ST_0 and ST_1, respectively. As previously mentioned, the bindings of the TOP variables are also included in the instance predicates. So the instance predicates are something like:

- $InPreimage(Robot, Xerox1, ST_0)$
- $InPreimage(Robot, Xerox2, ST_1)$

A set of background knowledge describes each situation, something like the following:

- $At(Robot, R1, ST_0)$
- $At(Xerox1, R1, ST_0)$
- $At(Robot, R2, ST_1)$
- $At(Xerox2, R2, ST_1)$

As it turns out, in computing the rlgg, we can limit the background knowledge for each instance to those literals that use the same situation argument. Therefore, the rlgg of the two instances above is exactly the lgg of the following two clauses:

- $InPreimage(Robot, Xerox1, ST_0) \leftarrow At(Robot, R1, ST_0) \wedge At(Xerox1, R1, ST_0)$
- $InPreimage(Robot, Xerox2, ST_1) \leftarrow At(Robot, R2, ST_1) \wedge At(Xerox2, R2, ST_1)$

We leave it as an exercise for the reader to show that the lgg of these two clauses is in fact

$InPreimage(Robot, ?x, ?st) \leftarrow At(Robot, ?p, ?st) \wedge At(?x, ?p, ?st)$

(After removing a few extraneous clauses such as those containing the variable that generalizes $Robot$ and $Xerox1$.)

When generating a new clause, GOLEM begins by randomly choosing pairs of positive instances and computing their rlggs with respect to the background knowledge. Since it is computing a generalization, both instances will be covered by the learned concept. Since it is computing the least general generalization, the concept will not cover any more negative instances than necessary. Among all the clauses generated from pairs of instances, one is chosen that maximizes coverage while not covering too many negative instances. This clause is further generalized by randomly choosing other positive instances and computing the rlgg of the clause and the chosen instance, attempting to find a useful extension that again does not cover too many negative instances. Once no further extensions can be made, GOLEM adds the clause to its concept definition and begins searching for other clauses.

GOLEM is in many ways a natural way of computing action-model preimages. The notion of an rlgg intuitively fits the preimage learning problem - finding a generalization that covers those situations where the action works. As was shown in the example above, GOLEM does not suffer from the connected literal problem discussed in the previous section. Furthermore, the fact that the background knowledge is partitioned by the state argument makes the rlgg computation process considerably simpler. An earlier version of TRAIL was in fact based on a learner that computed rlggs of instances and learned in a manner very similar to GOLEM.

However, there is one main problem with GOLEM that prevented it from being very useful for preimage computation. It turned out to be very difficult to get the GOLEM-based version of TRAIL to handle noise effectively. First, unlike in the case of FOIL, information-based search is very expensive, since each step involves computing a new rlgg. Second, we were unable to find any principled method of deciding when to stop expanding the rlgg and begin a new one. (Note that this problem is very similar to the stopping problem in FOIL.) Therefore we elected to switch to a top-down learning algorithm for the latest version of TRAIL.

5.3.3 LINUS and DINUS

The ILP system LINUS (Lavrač, Džeroski & Grobelnik 1991) is based on the observation that there is already a large and well-developed literature on concept learning using attribute-value representations of instances. If the ILP learning problem can be transformed so that it is expressible in terms of boolean features, then standard machine learning algorithms such as C4.5 (Quinlan 1992) and CN2 (Clark & Niblett 1989) can be used to learn a set of rules that express a concept. These rules can then be transformed back into first-order logic. (Sub-symbolic learning techniques such as neural networks are less appropriate for this application since it is difficult to convert the learned concept back into a symbolic preimage.)

The transformation process in LINUS is fairly simple. It makes use of predicate modes for literals in the domain, as introduced in Section 5.2. For the following example, let $G(?x, ?y)$ be an input-only predicate and $H(?x, ?y, ?z)$ be a predicate in which the first two arguments are inputs and the last argument is an output. Now, given that we are attempting to learn a concept $F(V_1, V_2, V_3, \ldots)$, the following "features" are constructed:

- Bindings of input variables to values, e.g. $V_i = a$
- Equalities among input variables, e.g. $V_i = V_j$
- Literals from the domain with input arguments from among the input variables, e.g. $G(V_i, V_j)$
- Literals from the domain with input arguments from among the input variables and output arguments that are constants, e.g. $H(V_i, V_j, a)$

Each instance provides a binding of the variables $V_1 \ldots V_n$. LINUS uses these bindings, together with the background knowledge, to compute truth values for each of the above literals for each instance in the obvious manner. Therefore, it has a set of features and values for each instance. These are then given to an attribute-value learning system, which induces a concept in terms of the features. LINUS then transforms this concept back into first-order logic, using the above feature definitions.

The LINUS system is quite limited in its usefulness for real ILP problems. In most ILP domains, including preimage learning, a learner needs to be able to include new variables in the learned concepts, and LINUS is unable to do so. However, there is an extension of LINUS known as DINUS (Džeroski et al. 1992) that uses the same framework but does allow the introduction of new variables, using *determinate literals* with specified predicate modes. Determinate literals, which will be explained in Section 5.4, introduce new variables that have uniquely defined values for each instance. These new variables can then be used in the creation of new features. DINUS is quite similar to the TRAIL learning system, which will be discussed in detail in the remainder of this chapter.

5.3.4 Applications

Aside from its applicability to preimage learning, Inductive Logic Programming has a number of other potential applications that are interesting to examine here. Most of the examples given in the ILP literature are trivial demonstration examples, such as family relationship databases and simple recursive LISP functions. However, there are a number of more realistic applications where ILP may prove useful. A more detailed overview of these applications is found in Bratko and Muggleton (1995).

One commonly cited application of ILP is the design of finite element meshes for analyzing stresses in physical structures (Dolvsak, Bratko & Jezernik 1994). Given some physical structure that is to be modeled, the modeler needs to decide on how finely meshed the model will be in each part of the structure. If the model is too coarsely meshed the simulation will be inaccurate, while if it is too finely meshed it will be too computationally expensive. The process of deciding on an appropriate resolution for each part of the structure is complicated and depends on the shape of the structure, the stresses applied, and the boundary conditions. The problem is very difficult for traditional attribute-value learners because the answers depend on relations between the primitive components of the structure, which fit more naturally into a first-order representation.

The problem of predicting secondary structure of proteins is another widely studied problem, with features similar to the finite element mesh design problem. The secondary structure of a protein (its three-dimensional spatial structure) depends on the elements of the protein and the relationships between them in complex ways, which are suitable for representation as an Inductive Logic Programming problem. GOLEM was applied to this problem (which is an important unsolved problem in molecular biology) and achieved 81% success on their testing set (Muggleton, King & Sternberg 1992), better than the previous best known result, which had used a neural network approach.

A related problem is concerned with predicting the activity of molecules given their structure. King, Sternberg & Srinivasan (1995) have investigated this problem, and found that an ILP algorithm outperformed both linear regression and decision trees when applied to complex compounds. Furthermore, the theories constructed by ILP were more understandable than those created by attribute-value methods.

5.4 An Indexical-Functional View of ILP

The concept learning algorithm in TRAIL is based on applying the concept of indexical-functional variables (Agre & Chapman 1987, Schoppers & Shu 1990) to the preimage instances. Recall that the $InPreimage$ literals have two distinct types of arguments: the reified state argument and the set of bindings of the TOP variables. (Again, the intended semantics of the literal $InPreimage(X_1, X_2, \ldots, X_n, ST_i)$ is that if the TOP variables are bound to the values $X_1, X_2, \ldots, X_n$ then the state represented by the state argument ST_i is in the preimage region of the TOP.)

In the remainder of this chapter, we will see that by viewing the TOP variable arguments and other variables as indexical-functional variables, we can easily re-represent the first-order instance descriptions in propositional form. This allows TRAIL to learn TOP preimages using a method very similar to the method used in the ILP system DINUS that we described in the previous section.

5.4.1 Indexical-Functional Variables

An indexical-functional variable, or IFV, is used to refer to an object that fills a particular role in a situation without knowing the specific identity of the object. Typical IFVs are variables such as `Location-of-Robot` and `Object-held-by-arm`. Although an IFV may change value across situations, we can refer directly to the IFV in literals that describe the situations.

?n	`Number-of-copies-made`
?obj	`Object-copied`
?x	`Copier`

Table 5.2: Indexical-Functional Representations of `copy` TOP Variables

Observe that the TOP variables for a TOP take on different values in each described situation, and thus can be represented using IFVs. Since the non-state arguments of the $InPreimage$ literals correspond to the set of TOP variables, these arguments can naturally be represented using IFVs as well. For instance, in the copying example from Figure 5.1, the arguments of the literal $InPreimage(?n, ?obj, ?x, ?st)$ have the indexical-functional meanings shown in Table 5.2.

Thus, the first literal of the induced preimage for the `copy` TOP can be viewed in indexical-functional terms as $Has(Robot,$`Object-copied`$)$. Of course, the learner does not have any such English-ized representation of the IFVs, but using these indexical-functional names is an intuitive way of conceptualizing what the learner is doing.

The most interesting result of representing the variables in an induced ILP concept as IFVs comes in analyzing the other variables that are included in the concept. Consider the second literal in the induced preimage for the TOP in Figure 5.1, $At(Robot, ?p, ?st)$. The variable $?p$ corresponds to the IFV `Location-of-Robot`, meaning that the literal is the vacuous condition $At(Robot,$`Location-of-Robot`$)$. However, the third literal, $At(?x, ?p, ?st)$ becomes $At($`Copier`, `Location-of-Robot`$)$, which is of course a crucial precondition for the action to succeed. This ability to refer to variables such as `Location-of-Robot` across multiple literals is one of the main advantages of using ILP for learning.

The formal definition of induced IFVs follows from the predicate mode definitions mentioned earlier. We begin by adopting the notion of *determinacy* from the system GOLEM (Muggleton & Feng 1990). For our purposes, a literal is determinate if, given any binding of its input arguments in a situation, each of its output arguments has at most one possible binding. (The situation argument in each literal is always treated as an input variable and does not actually get used.) In other words, each output argument must be a function of the input arguments and the state argument. For instance, the literal $At(?x, ?y)$ is determinate if we consider $?x$ as an input and $?y$ as an output, since each binding of $?x$ in a situation will produce only one binding for the location argument $?y$. However, if we consider $?y$ as an input and $?x$ as an output, $At(?x, ?y)$ is not determinate because each location may contain several objects or people. Note that by the definition above, a literal that has only input arguments is always determinate. A literal that has only output arguments, such as the literal $Altitude(?x)$ in the Flight Simulator domain, may or may not be determinate.

Indexical-functional variables, then, can be used to represent the output arguments of determinate literals as well as the TOP variables. A determinate output-only literal produces a single IFV for each argument. An input-only literal produces no IFVs at all. And for determinate literals containing both input and output arguments, each possible assignment of a set of variables to the input arguments produces a new IFV for each output argument. Consider the literal $Has(?x, ?y)$, in which $?y$ is an input argument and $?x$ is an output argument. If $?y$ is set to `Object-copied`, the output argument corresponds to the new IFV `Possessor-of-Object-copied`, while if $?y$ is set to `Copier`, the output argument is `Possessor-of-Copier`. The input arguments can be derived IFVs as well; consider `Location-of-Possessor-of-Object-copied`. Finally, the input arguments can also be special *domain constants*, such as `Robot`, that are likely to be useful in learning several operators. For instance, a variable such as `Location-of-Robot` is probably relevant to many different TOPs. These domain constants are included in the specification of the domain by the designer.

The process for creating IFVs given a particular state description is an iterative cycle, which is illustrated by an example in the next section. At any point during the IFV creation process, the learner has a set of domain objects it is presently focusing on, known as *active referents*. Each active referent is either a domain constant or the current value of one or more IFVs, annotated appropriately with the names of the variables that are bound to it. The initial set of active referents consists of the domain constants plus the value of each TOP variable. (Recall that each TOP variable corresponds to an IFV.)

Now, on each step of the cycle, the learner finds all determinate literals in the state description such that all of the input arguments are objects in the set of active referents, including at least one active referent introduced in the previous cycle (to avoid generating the same IFV multiple times.) A new IFV is assigned to each of the output arguments for such a literal, and its binding is added to the list of active referents. This process is repeated as long as new IFVs are created. Note that we are making the *unique names assumption* here; if a literal refers to an object constant we assume that mentions of that object constant in other literals are in fact referring to the same object in the world. (Actually we allow for exceptions to this rule in the case of multiple indistinguishable objects, i.e. several copies of a book. The exact semantics of this exception are complicated and not really relevant here.)

There is one limitation that we place on the IFV generation process. Suppose that a database contains the literals $Father(Bob, Adam)$ and $Father(Adam, Bob)$. (Clearly this is either a mistake or a violation of the unique names assumption!) This pair of literals allows for the creation of an infinite chain of created literals:

```
Father-of-Adam

Father-of-Father-of-Adam

Father-of-Father-of-Father-of-Adam

...
```

In order to prevent such an infinite chain, we add an "occurs-check" that makes sure that no indexical-functional variable has the same value as any of the IFVs used to define it. Observe that each IFV is defined by a literal, a position, and (possibly) some set of other IFVs that serve as the input arguments for the literal. These input IFVs are recursively defined in terms of other IFVs, eventually grounding out in TOP variables or domain constants. Therefore it is easy to perform the occurs-check by simply comparing the value of the newly created variable to the bindings of all the variables that are directly or indirectly used in defining it.

$At(Robot, Room101)$	$At(Xerox1, Room101)$	$Mode(Xerox1, Copy)$
$Author(Article5, Nilsson)$	$Size(Room101, Large)$	$Has(Robot, Article5)$
$CurrentTime(1200)$		

Table 5.3: Description of a State s_0

5.4.2 An Example of IFV Computation

Now we will look at an illustration of the IFV computation process in a specific situation, based loosely on the copy-making example introduced in Section 5.1. Suppose that the action `copy(Article5,Xerox1)` achieves $Has(Robot, Copies(3, Article5))$ in a state s_0. TRAIL records the description of state s_0 as a positive instance of the TOP $\xrightarrow{\texttt{copy(?obj,?x)}} Has(Robot, copies(?n, ?obj))$ with the TOP variables bound as follows: ($?n/3$, $?obj/Article5$, $?x/Xerox1$). A simplified description of state s_0 is shown in Table 5.3. (Note that for simplicity we have not included the *Paper* predicate in this example.)

The definition of the domain specifies modes for each of the predicates in the domain. The predicates At, $Mode$, $Author$, and $Size$ are all determinate predicates with the first argument as an input argument and the second argument as an output argument. The predicate $CurrentTime$ is a determinate output-only predicate. The predicate Has we use as an input-only predicate; for any agent and object it tells whether the agent possesses that object. (Clearly a person can possess more than one object, and an object can be possessed by more than one person, such as two people having copies of the same book.)

Formally, the predicate modes for the literals in the domain are:

- At(:input :output)
- $Mode$(:input :output)
- $Author$(:input :output)
- $Size$(:input :output)
- Has(:input :input)

Object	**Variable**
3	`Number-of-copies-made`
Article5	`Object-copied`
Xerox1	`Copier`
Robot	*(domain constant)*

Table 5.4: An Initial Set of Active Referents

- $CurrentTime$(:output)

We start with four initial active referents: the bindings of the three TOP variables (`Number-of-copies-made`, `Object-copied`, and `Copier`) and the domain constant *Robot*. The active referents are shown in Table 5.4. Note that of course the learner itself does not use these mnemonic names for the IFVs; we use them merely for ease of reference. (The learner actually uses arbitrarily named variables.)

In each iteration of the cycle of indexical-functional variable creation, we look at the literals in our state description and find ones for which the input variables match active referents. Examine the first literal, $At(Robot, Room101)$. Its input argument *Robot* matches one of the active referents, so the output argument of *At* defines a new IFV. Since the input argument in this case is the domain constant *Robot* the new IFV does not refer to any other IFVs and is simply `Location-of-Robot`.

The next literal is similar: $At(Xerox1, Room101)$. Again the input argument $Xerox1$ corresponds to an active referent so the output is a new IFV. Since the input is the IFV `Copier`, the output variable is `Location-of-Copier`.

The next two literals introduce new variables in the exact same way; the literal $Mode(Xerox1, Copy)$ leads to the creation of `Mode-of-Copier` while the literal $Author(Article5, Nilsson)$ leads to the creation of `Author-of-Object-copied`.

Neither of the next two literals in the state description introduce any new variables at all. $Size(Room101, Large)$ is a determinate literal, but the input argument $Room101$ does not correspond to any active referent. The literal $Has(Robot, Article5)$ is an input-only literal and cannot introduce any new IFVs.

Finally, we come to $CurrentTime(1200)$. $CurrentTime$ is clearly determinate, as

Object	**Variable**
3	`Number-of-copies-made`
Article5	`Object-copied`
Xerox1	`Copier`
Robot	*(domain constant)*
Room101	`Location-of-Robot`
	`Location-of-Copier`
Copy	`Mode-of-Copier`
Nilsson	`Author-of-Object-copied`
1200	`The-time`

Table 5.5: Set of Active Referents After the First Cycle

it can have at most one value in each situation. Therefore the value of this output-only literal is a well-defined IFV, in this case corresponding to `The-time`.

The current set of active referents is shown in Table 5.5 (note that the first active referent, 3, corresponding to `Number-of-copies-made`, has not been used.)

In the second cycle of the variable creation process, we look again at the literals describing the situation and see if any new IFVs can be created. In this case the first four literals do not allow anything new to be created, as none of the variables created on the previous cycle have values that are equal to their input arguments. However, we can now introduce new variables based on the literal $Size(Room101, Large)$. The input argument for this literal now corresponds to an active referent, and each variable that is bound to $Room101$ allows a new IFV to be created. Thus the learner creates two new variables, `Size-of-Location-of-Robot` and `Size-of-Location-of-Copier`. No further IFVs are created on this cycle, as the input-only predicate $Has(Robot, Article5)$ still produces nothing and the predicate $CurrentTime(1200)$ introduces nothing new.

One might wonder why it is that we create two distinct `Size-of` IFVs given that there was only one active referent used. The reason is that in other instances, it may not be the case that `Location-of-Robot` and `Location-of-Copier` are bound to identical values. Therefore, in other instances, `Size-of-Location-of-Robot` and

Object	**Variable**
3	`Number-of-copies-made`
Article5	`Object-copied`
Xerox1	`Copier`
Robot	*(domain constant)*
Room101	`Location-of-Robot`
	`Location-of-Copier`
Copy	`Mode-of-Copier`
Nilsson	`Author-of-Object-copied`
1200	`The-time`
Large	`Size-of-Location-of-Robot`
	`Size-of-Location-of-Copier`

Table 5.6: Set of Active Referents After the Second Cycle

`Size-of-Location-of-Copier` might have been created from distinct active referents, and thus could have different values. In order to have the use of IFVs consistent across instances, we need to have them be distinct variables in every instance, even when the location that is the input for `Size-of` is the same.

The full set of active referents is now as shown in Table 5.6. The reader will note that if a third stage of computation is done, no new variables will be introduced, so the indexical-functional variable computation is complete for this instance.

5.4.3 Indexical-Functional Instance Representations

The purpose of computing an indexical-functional representation of a situation (as described in the previous sections) is to allow the learner to generate concepts that have meaning in multiple situations but may refer to different objects in each of these situations. For instance, the learner may need to reason about some property of the TOP variable `Object-copied`. Although this variable may refer to a different physical object in each situation, the function of `Object-copied` is the same across all of them. The same holds true for derived IFVs, such as `Location-of-Robot`, which are computed according to the chaining process described above.

Using the set of TOP variables and derived IFVs, the learner can construct a representation of each situation entirely in terms of facts about IFVs. This representation consists of five parts.

First, we need a literal that defines the value of each derived variable. For instance, consider the literal *At*(`Copier`,`Location-of-Copier`). Given that the TOP variable bindings produce a unique value of `Copier` in each situation, this literal defines a unique value of the derived IFV `Location-of-Copier`, assuming, of course, that the literal $At(?x, ?y)$ is determinate. These definitions are constant across all situations, although not all of them may apply in every situation; it is possible that no literal in a situation description provides a value for some derived IFV.

Second, the learner must record the value in the situation of each of the defined IFVs. The learner will sometimes need to reason about specific constants in the domain. For instance, the value of `Mode-of-Copier` must be *Copy* in order for the copying TOP to work. Reasoning about the values of IFVs is also essential in inducing interval conditions, as described in Section 5.5.6. We represent this knowledge by including a new literal *Bound* whose first argument is always an IFV and whose second argument is always an object constant. For instance,

Bound(`Mode-of-Copier`, *Copy*)

One such *Bound* literal is recorded for each IFV that has a value in the situation, including those that correspond to TOP variables.[4]

Third, we must consider a special case that arises if one of the TOP variables is bound to a numeric value. In this case, the learner may need to have information about the relationship between the current value of this variable and its target value. For instance, suppose we are learning a TOP that uses the `up` action to achieve $Altitude(?x)$. This TOP will obviously only work if the plane is currently lower than the target altitude. The preimage must thus include a literal of the form $\leq$(`The-altitude`, `Target-altitude`). Therefore, if an IFV representing a TOP variable from the the goal of the TOP is bound to a numeric value, and some other IFV

[4]We could also imagine a general *BoundP* predicate that is true for an indexical-functional variable iff the variable has a defined value in the situation. This does not appear to be very useful for action-model learning.

in the domain represents the current value of that goal variable, a $\leq$ or $\geq$ literal is included that indicates the relation between the two IFVs. This is the only circumstance in which TRAIL includes inequalities in the indexical-functional situation representations, as other inequalities are likely to be meaningless or irrelevant, e.g. $\geq$(`The-altitude`, `The-time`).

Fourth, if there are any input-only literals in the domain, any instances of such a literal provide information about the situation. Suppose an instance contains the literal $Has(Scott, Book1)$. The two object constants $Scott$ and $Book1$ may both be the values of several IFVs. Suppose that $Scott$ is the value of the variables `Owner-of-Robot` and `Nearest-person-to-Robot` and that $Book1$ is the value of the variable `Desired-object`. Then the two literals

- Has(`Owner-of-Robot`, `Desired-object`)
- Has(`Nearest-person-to-Robot`, `Desired-object`)

both need to be added to the state description. Every possible combination of active referents that corresponds to the arguments of the actual literal will result in a literal in the representation. Note that the learner only creates these new literals from input-only literals, not from any of the other literals in the domain. This distinction is discussed further below.

Finally, note that so far we have only included two types of domain literals in our representations: the defining literal for each IFV, and any input-only literals. However, there is still much more information contained in a situation description. Consider the literal $At(Xerox1, Room101)$ from our earlier example. The literal At(`Copier`, `Location-of-Copier`) is included in our description already since it is the definition of the variable `Location-of-Copier`. However, we note that the object $Room101$ is also the value of `Location-of-Robot`. This fact can be expressed in one of two ways: either by including the predicates At(`Copier`, `Location-of-Robot`) and At(`Robot`, `Location-of-Copier`) or by including a predicate indicating that `Location-of-Copier` is the same as `Location-of-Robot` in this situation. We choose to adopt the second approach and include special $Equal$ predicates for each pair of IFVs that have the same bindings in a situation. (If there are n derived IFVs that

Object	**Variable**
3	`Number-of-copies-made`
Article5	`Object-copied`
Xerox1	`Copier`
Robot	*(domain constant)*
Room101	`Location-of-Robot`
	`Location-of-Copier`
Copy	`Mode-of-Copier`
Nilsson	`Author-of-Object-copied`
1200	`The-time`
Large	`Size-of-Location-of-Robot`
	`Size-of-Location-of-Copier`

Table 5.7: Active Referents to be Used in IFV Representation

have the same binding in a situation, we would need to include at least $n(n-1)$ domain predicates in order to cover all the possible substitutions of other variables into the original definitional literals, while there are only $n(n-1)/2$ pairwise equality statements, assuming that there is some fixed ordering on variables to avoid the need to include both $Equal(A, B)$ and $Equal(B, A)$.)

5.4.4 The Example Continued

Let us return to our example from Section 5.4.2. Recall that at the end of the indexical-functional computation process we had created the variables shown in Table 5.7:

There are seven new variables that were introduced during the computation process. Our representation of the instance must include the definitional literals for each of these:

- $At(Robot,$`Location-of-Robot`$)$
- $At($`Copier`, `Location-of-Copier`$)$
- $Mode($`Copier`, `Mode-of-Copier`$)$

- $Author$(`Object-copied`, `Author-of-Object-copied`)
- $CurrentTime$(`The-time`)
- $Size$(`Location-of-Robot`, `Size-of-Location-of-Robot`)
- $Size$(`Location-of-Copier`, `Size-of-Location-of-Copier`)

Next, the bindings of each indexical-functional variable must be expressed using *Bound* literals. This applies to both the seven created IFVs and the three that came from the TOP variables:

- $Bound$(`Number-of-copies-made`, 3)
- $Bound$(`Object-copied`, $Article5$)
- $Bound$(`Copier`, $Xerox1$)
- $Bound$(`Location-of-Robot`,$Room101$)
- $Bound$(`Location-of-Copier`,$Room101$)
- $Bound$(`Mode-of-Copier`,$Copy$)
- $Bound$(`Author-of-Object-copied`,$Nilsson$)
- $Bound$(`The-time`,1200)
- $Bound$(`Size-of-Location-of-Robot`,$Large$)
- $Bound$(`Size-of-Location-of-Copier`,$Large$)

One of the TOP variable IFVs, `Number-of-copies-made`, is bound to a numeric value, but there is no other IFV expressing the current value of a corresponding state attribute. Therefore, there is no need to include an inequality between `Number-of-copies-made` and some other IFV.

We have only one input-only literal in the situation, $Has(Robot, Article5)$. There is only one active referent matching each argument, so there is only one literal that needs to be included based on the input-only literal:

- *Has*(*Robot*, `Object-copied`)

Finally, we check to see whether any pairs of IFVs are equal in this situation. There are two such pairs, `Location-of-Robot` and `Location-of-Copier` and the corresponding `Size-of` variables. They are expressed in the following two literals:

- *Equal*(`Location-of-Robot`, `Location-of-Copier`)
- *Equal*(`Size-of-Location-of-Robot`, `Size-of-Location-of-Copier`)

This completes our indexical-functional re-representation of the state description. All the relevant information contained in the instance is now expressed in these indexical-functional literals.

5.4.5 Using Indexical-Functional Representations for Learning

The main advantage of using indexical-functional representations as described in the previous two sections is that we have considerably simplified the learning instances. The original representation is fundamentally first-order, in that objects must be reasoned about explicitly. Suppose that we were to try to represent the world state propositionally, using propositions such as $RobotAtRoom101$. The precondition for the `copy` operator would then include some clause such as $RobotAtRoom101 \wedge XeroxAtRoom101$, a concept that does not generalize at all to new situations.

In the indexical-functional representation, on the other hand, the instances are truly propositional. For instance, the two literals *Has*(*Robot*, `Object-Copied`) and *Equal*(`Location-of-Robot`, `Location-of-Copier`) are both propositions; they are either true or false in any situation. And they are fully general; the literal *Has*(*Robot*, `Object-Copied`) has the same meaning relative to the copying TOP in every situation regardless of the binding of `Object-Copied`. Therefore, the relevant facts in each situation can be described by a finite set of boolean features, consistent across situations.

The representation is actually slightly more complicated than a strictly boolean representation due to the presence of *Bound* literals. These can be viewed in machine learning terms as non-boolean feature inputs; the value of each IFV is either a continuous numeric feature or a discrete non-numeric feature. Numeric features are found in many concept learning applications, and are handled in TRAIL using methods discussed in Section 5.5.6. The discrete non-numeric features may seem problematic since the values of an IFV are not limited to one of a small number of possible values. However, it is normally the case that either the value of the variable is irrelevant (most of the time) or the variable must have one particular value, such as copy-mode *Copy*. In our test domains, it is never necessary to construct complex disjunctions over the possible values of a discrete IFV. We suspect that this is probably true of most preimage learning problems.

As we have seen, once we have constructed the set of IFVs for a set of instances, we can convert each of the instances to representations consisting entirely of propositions and non-boolean *Bound* features. The learning problem is thus much easier, and also more familiar as most of the existing concept learning work has focused on domains with propositional or numeric features. The way in which TRAIL actually learns preimages given the propositional representations is covered in the next section.

5.5 ILP Learning Strategies

5.5.1 The Generic Top-Down Learning Algorithm

As discussed in Section 5.3.2, we have elected to learn in TRAIL using a top-down learning strategy.[5] Such a strategy allows us to consider the whole dataset in order to make intelligent decisions on how to specialize a proposed concept. TRAIL's learning mechanism is based on the covering algorithm used in AQ (Michalski 1983) and FOIL (Quinlan 1990). A pseudo-code sketch of the algorithm is shown in Figure 5.2.

[5]It is interesting to note that the IFV representation discussed in the previous section would also be quite usable for a bottom-up learner such as GOLEM, as the pruned rlgg of two instances is simply the intersection of the IFV representations of the instances.

```
$H = F$
repeat
    $C = T$
    repeat
        Select a literal $L$ and add it to $C$: $C = C \wedge L$
    until $ClauseStoppingCriterion$
    Add $C$ to $H$: $H = H \vee C$
    Remove all positive instances covered by $C$
until $ConceptStoppingCriterion$
```

Figure 5.2: The Generic Covering Algorithm

There are four questions that must be answered in implementing a top-down covering algorithm based on the above pseudo-code:

- What is the set of literals from which L is chosen?
- What criterion is used to decide among the possible literals to add?
- What criterion is used to decide when to stop adding literals to a clause? ($ClauseStoppingCriterion$)
- What criterion is used to decide when to stop adding clauses to the hypothesis? ($ConceptStoppingCriterion$)

As a simple example, consider the artificial data set shown in Table 5.8. Suppose we wish to learn a concept that covers only the positive instances. We are given the three literals L_1, L_2, and L_3 and the boolean value for each literal in each instance. Our top-down learning algorithm begins with the clause T. It now applies some literal selection criterion, searching for a literal that will make the clause cover mostly positive examples. (The exact selection criterion is not important here; literal selection methods will be covered further in Section 5.5.3.) Examining the possible literals that could be added, the learner selects L_3, as that literal causes the clause to cover three positive instances and only one negative instance. L_3 is thus added to the clause. The clause does not yet satisfy the clause stopping criterion, so the learner searches for another literal to add. This time it selects L_1, so the clause becomes $L_3 \wedge L_1$. This clause covers only positive instances, so the learner adds it to H, and removes the two positive instances I_6 and I_8 covered by the clause.

Instance	L_1	L_2	L_3	**Label**
I_1	0	0	0	+
I_2	0	0	1	+
I_3	0	1	0	-
I_4	0	1	1	-
I_5	1	0	0	-
I_6	1	0	1	+
I_7	1	1	0	-
I_8	1	1	1	+

Table 5.8: A Simple Concept Learning Problem

The hypothesis does not yet cover all the positive instances, so the learner searches for another clause. This time it begins with T and adds $\neg L_2$, as $\neg L_2$ causes the clause to cover both remaining positive instances and only one negative instance. Finally, it adds $\neg L_1$ to the clause, as this literal excludes the last negative instance. The second clause is now added to the hypothesis, resulting in $(L_1 \wedge L_3) \vee (\neg L_2 \wedge \neg L_1)$. This hypothesis covers all of the positive instances and no negative instances, so the concept stopping criterion is satisfied, and the learner has produced a successful hypothesis.

The set of literals from which L can be chosen in TRAIL follows directly from the indexical-functional representation discussed above. Any literal that is found in the indexical-functional representation of any positive instance is a candidate for inclusion in the concept, as is the negation of any literal found in the representation of some negative instance. (Actually, we might also want to include interval conditions, which do not correspond directly to any single literal. This issue is discussed further in Section 5.5.6.) The literal selection and stopping criteria used by TRAIL are discussed beginning in Section 5.5.4.

5.5.2 Accuracy and Coverage

In evaluating the correctness of a learned concept we make use of two distinct measures of goodness, *accuracy* and *coverage*. Accuracy corresponds to the fraction of instances covered by the concept that are positive. Coverage is the fraction of the positive instances that are covered by the concept. Accuracy and coverage of the clauses within a concept are defined analogously. Clearly we desire concepts that are high in both accuracy and coverage.

Deciding which literal to add to a partially constructed clause (and in fact, deciding whether to add a literal at all) is a question of trading off accuracy against coverage. Each literal added can exclude negative instances, and can thus potentially increase the accuracy of the concept, but will generally also reduce the coverage of the clause by excluding some positive instances.

It is important to note that for a covering algorithm, the considerations of accuracy and coverage are not equivalent. If a positive instance is not covered by a clause, the learner may later induce some other clause that does cover the instance. But if a negative instance is covered, then it is guaranteed to be covered by the concept as a whole, reducing the overall accuracy of the concept. Therefore, we can claim that the accuracy of a clause is more important to a covering algorithm than the coverage. However, it is important not to take this claim too far. In addition to accuracy and coverage, simplicity is also important in learned preimages. Since TRAIL will be using the preimages to do backchaining, it is important that the learned preimages be relatively simple. Therefore, we must avoid including too many clauses in induced concepts. As an extreme example, suppose that we included one clause for each positive instance. Assuming there were no instances with conflicting labels, the resulting concept would have 100% accuracy and 100% coverage. However, not only will this concept be so complex as to be useless for planning, but it will also generalize very poorly to new situations, a condition known as overfitting the data. It is a well-known result in machine learning that shorter concepts tend to have better generalization performance, in addition to simplifying the planning process.

5.5.3 Literal Selection Criteria in FOIL and CN2

The ILP system FOIL uses a literal selection criterion known as *weighted information gain* that is indirectly based a simple accuracy estimate. Let $n(C)$ be the number of instances covered by a concept C, and let $n^+(C)$ and $n^-(C)$ be the number of positive and negative instances, respectively, covered by C. The *informativity* of a concept refers to the amount of information needed to signal that an instance covered by the concept is positive, and is given by

$I(C) = -\log_2 \frac{n^+(C)}{n(C)}$

Clearly the informativity of a concept decreases as the accuracy increases; a correct concept has informativity zero. Therefore, the inverse of the informativity is a reasonable measure of concept quality. The *information gain* of a literal L is given by

$IG(L) = I(C) - I(C \wedge L)$

Of course, the information gain metric does not take into account the coverage of the concept at all; if L reduces the concept to covering a single, positive instance, it will have maximal information gain. FOIL handles this problem in a rudimentary way by simply multiplying the information gain by the number of instances covered by the resultant concept, resulting in a formula known as *Weighted Information Gain*:

$WIG(L) = n^+(C \wedge L) * (I(C) - I(C \wedge L))$

This heuristic does have the right general properties for literal selection, but has the disadvantage that it is only a local measurement, and provides no way of comparing two concepts. In particular, any literal that increases the accuracy even slightly has a positive weighted information gain, so it is not possible to directly evaluate the possibility of not adding any more literals at all. (One could of course select an arbitrary threshold as a minimum weighted information gain needed to add a literal.) Instead, FOIL uses a clause stopping criterion based on an *encoding length restriction*; the number of bits used in a clause cannot exceed the number of bits that would be needed to explicitly encode the labelings of all the instances remaining in the data set. This heuristic is problematic for a number of reasons: if there are few instances it may prevent the system from learning a correct clause, while if there are many instances it can use the extra bits to overfit the data. Lavrač & Džeroski (1994) cite the following example: suppose some region of a dataset has 1023 negative instances and one positive instance. The one positive instance is most likely noise, but FOIL will have 20 ($\log_2 1024 + \log_2 1024$) bits with which to build a clause covering the instance.

The attribute-value learning system CN2 (Clark & Niblett 1989) uses a covering algorithm essentially similar to FOIL, but does search using beam search rather than hill climbing. Literals are added to clauses according to an accuracy estimate known as the Laplace estimate. Before literals are added, the resulting clause is tested to see whether it would be statistically significant. This helps to avoid including clauses with low coverage, but has no preference for very general clauses over ones that barely exceed the significance threshold. In addition, if there are few instances it may be impossible to find any clauses that exceed the significance threshold.

5.5.4 Strategies for Learning Preimages

In developing a learning algorithm for action-model preimages, it is important to consider the desired form of the learned preimages. We argued above that it is important that preimages be simple and symbolic, in order to be able to do useful backward chaining on them. This simplicity bias corresponds to our own observations about preimages. Both in our experimental domains and in other planning domains found in the literature, most operator preconditions tend to be either monomials (conjunctions of literals) or small disjunctions of monomials. Thus, our learner should be biased toward learning such preimages.

There has been little work within the machine learning community on learning small disjunctions of conjunctions. HILLARY (Iba, Wogulis & Langley 1988) is an incremental algorithm that explicitly considers both simplicity and coverage in developing disjunctive concepts. HILLARY would be a promising algorithm for use in action-model learning, but we were unaware of it at the time we were developing TRAIL's learning algorithm. Aside from this system, most concept learning research has focused on learning decision trees or sets of rules, which do not generally translate well to action-model preimages. Shen's CDL algorithms (Shen 1990) appear to learn small preconditions effectively within his LIVE system, but do not provide any noise handling mechanism. Covering algorithms such as FOIL produce disjunctions of conjunctions, but the learning heuristics used in FOIL are unsatisfactory for a number of reasons, as discussed above. Therefore, in designing TRAIL, we needed to develop our own covering-based algorithm for learning TOP preimages.

Monomials can be learned by using a straightforward hill-climbing search with a metric that is a simple weighted average of the coverage and accuracy of each proposed concept. Given the notation introduced in the previous section, we can estimate the accuracy and coverage by simply counting the number of instances covered by the concept, as follows:

$Acc(C) = \frac{n^+(C)}{n(C)}$
$Cov(C) = \frac{n^+(C)}{n^+(T)}$

An outline of the algorithm is shown in Figure 5.3. Note that the search metric itself, defined as $M(c)$ in the first line of Figure 5.3, is somewhat arbitrarily chosen, but does embody the desired criteria discussed in the previous section. The inclusion of coverage as well as accuracy in the metric avoids the difficulty of overvaluing literals that make the concept cover only a small number of positive instances. It also provides a way to compare the various potential added literals to the option of not adding a literal at all. If none of the potential literals increases the accuracy enough to compensate for the (possible) decrease in coverage, then the clause is complete. This provides a natural clause stopping criterion for the search process. (Of course, we could also apply a lookahead search with the same literal selection and stopping criterion. We expect that this would probably improve performance.)

Learning small disjunctions requires a slightly different approach. Single disjuncts tend to be regions of the search space that have high accuracy and low but not insignificant coverage, perhaps in the range of 20% – 50%. In many domains, hill-climbing search with a metric such as Coverage + Accuracy will have difficulty finding such regions. However, significantly increasing the weighting of the accuracy term makes it too likely that the learner will identify regions that cover only a few instances. Therefore, in conjunction with the monomial search described above, TRAIL's learner also does a search for disjuncts using a slightly different measure. We (rather arbitrarily) selected $n^+(C) - k * n^-(C)$ with $k = 2$ as the metric for the learner. The learning

```
$M(c) = Acc(c) + k * Cov(c)$

$C = T$
$EXIT = F$
repeat
    Find literal $L$ that maximizes $M(C \wedge L)$
    if $M(C \wedge L) > M(C)$ then
        $C = C \wedge L$
    else
        $EXIT = T$
until $EXIT$
return($C$)
```

Figure 5.3: Algorithm For Learning Monomials

algorithm itself is fairly simple, as shown in Figure 5.4.

Note our use of the encoding length principle as a concept stopping criterion - if the number of bits needed to add a disjunct that covers some of the remaining instances is less than the number of bits needed to encode the labels of the instances themselves, the growth of the disjunction is halted; otherwise, the procedure is called recursively. Encoding length is also used in the same way in deciding whether to use a monomial generated by the first algorithm or a disjunction generated by the second algorithm.

We have used the algorithms described above for all of our action-model learning work, and the combination appears to produce satisfactory results. First off, learning is relatively fast, important in an incremental domain. Second, it works well even for small numbers of instances. This is essential in action-model learning, where we need to get reasonable performance as soon as possible, without waiting for hundreds of instances. Finally, although it is a simple solution that is unlikely to be competitive with modern decision tree algorithms on large data sets, it appears to have good performance in practice when applied to preimage learning. The limiting factors in TRAIL's performance at present appear to be instance generation and domain noise, rather than the learning algorithm. We will discuss this issue further in Chapter 6.

```
$M(c) = n^+(c) - k * n^-(c)$

$C = T$
$EXIT = F$
repeat
    Find literal $L$ that maximizes $M(C \wedge L)$
    if $M(C \wedge L) > M(C)$ then
        $C = C \wedge L$
    else
        $EXIT = T$
until $EXIT$
$E'$ = instances not covered by $C$
if $EncodingLength(C + exceptions) < EncodingLength(instances)$ then
    $C' = LEARN(E')$
    return$(C \vee C')$
else return$(T)$
```

Figure 5.4: Algorithm For Learning Disjunctions

5.5.5 Heuristics for Learning from Positive Instances

We noted in Section 4.4.5 that TRAIL's method of generating negative instances from a teacher is dependent on an ability to learn a useful approximation to a TOP preimage when given only positive instances. In addition, it was found during the development of TRAIL that learning is sometimes speeded by the addition of certain probably relevant literals to preimage conditions. Therefore, TRAIL's learning system sometimes generates preimage conditions that are more specific than would be necessary simply to exclude all the given negative examples.

In general, the most relevant literals for a TOP preimage are those that refer to the TOP variables. If a condition on a TOP variable has been true in all the states in which the TOP has succeeded, there is a good chance that the condition should actually be in the preimage condition of the TOP. For instance, if the $\overset{\texttt{grab}}{\rightarrow} Holding(?x)$ TOP has only succeeded when $FacingBar(?x)$ holds, the literal $FacingBar(?x)$ is likely to be an element of the preimage of the TOP. Of course, this heuristic does not always work, so once TRAIL observes a positive in which $FacingBar(?x)$ does not hold, the literal should no longer be included by default in the preimage condition.

Thus, upon completion of the basic preimage learning algorithm discussed above, TRAIL will attempt to specialize the preimage condition by adding literals that refer to the TOP variables. (These literals correspond to the *inner circle relations* in LIVE (Shen 1994).) However, it should only do so as long as these literals hold in all of the positive instances that are currently covered by the preimage condition. Therefore, TRAIL adds any literal that mentions a TOP variable and does not reduce the coverage of the preimage.[6] This mechanism produces accurate preimage estimates more quickly, and allows TRAIL to intelligently generate a preimage that is more specific than T even when given only positive instances as input.

5.5.6 Including Intervals in Preimages

Normally, TRAIL does not reason about specific objects in the world. It is often important whether two indexical-functional variables have equal values, but it is usually not important what these exact values are. There are exceptions to this rule, of course, such as the literal $Bound($`Mode-of-Copier`$, Copy)$ that we discussed above. However, most of the exceptions take the form of real-valued variables for which the actual value is important to the success of an action. This is particularly evident in the flight simulator domain, as it becomes important to express facts about these variables, facts such as $GreaterThan(\texttt{speed}, 55)$. Unless we are to include these facts as specific propositions in the state descriptions, the learner needs to be able to induce ranges on the values of numeric arguments of literals.

The range induction process is quite straightforward. Whenever the instance descriptions include a *Bound* literal that binds an IFV to a numeric constant, the learning algorithm will consider adding literals of the form $IntervalBound(varname, low-end, high-end)$ to the concept. The utility of adding these literals can clearly be evaluated using the same criteria used to evaluate other literals considered during learning. Of course, these literals greatly increase the branching factor of the search

[6]One could consider accepting small reductions in coverage in order to be able to exclude some false positive instances. TRAIL does not do this, since if the preimage is overly general, plan executions will soon generate negative instances.

space. If a variable takes on n different values in the set of positive instances, there are $n(n-1)/2$ possible interval literals that could be induced. Ideally, we would like to have some heuristic that provides only a few likely candidate interval literals from this set. For instance, we might simply force one endpoint of each candidate interval to be infinity or negative infinity. However, at present the system simply tries all possible intervals.

The technique of using the value of a numeric variable as an input to an attribute-value learner can also be applied in the DINUS framework, if DINUS is given a learner that can handle continuous features. This approach is applied to various problems in behavioral cloning by Džeroski, Todorovski & Urbančič (1995).

5.6 Converting from Concepts to TOPs

Once a concept has been learned, the process of converting it back into a TOP preimage is fairly straightforward. The predicate of each literal in the concept is either a predicate in the domain or one of the artificial *Equal*, *Bound*, or *IntervalBound* predicates. The arguments of each literal are either indexical-functional variables, constants corresponding to the values of IFVs, or domain constants such as *Robot*.

The first step in the conversion process is to assign a variable name to each IFV. If the IFV corresponds to a TOP variable, the TOP variable name can be used; otherwise the learner must assign a new name to the variable.

If a new name has been assigned to a variable, the learner must insure that the name is given a meaning within the context of the TOP. For instance, if the concept includes some literal that mentions the IFV `Location-of-Robot`, and the IFV is assigned a new variable name, say $?v_{17}$, the TOP must contain some indication of what $?v_{17}$ means, so that when the precondition of the TOP is matched against a situation (as would be done when executing a plan created using the TOP,) $?v_{17}$ can be bound to the current location of the robot. In order to do this, we simply need to include in the preimage the definitional literal (as mentioned at the beginning of Section 5.4.3) for the IFV `Location-of-Robot`. In this case, the definitional literal is *At*(`Robot`, `Location-of-Robot`), which would be converted to the literal *At*(`Robot`,

$?v_{17}$) and included in the preimage. This literal thus insures that $?v_{17}$ is correctly bound whenever the preimage is matched to a situation. Since each IFV (other than the TOP variables) has exactly one definitional literal, the learner can simply add all of the appropriate definitional literals to the preimage and substitute in the variable names for the IFVs.

The artificial literals are also relatively simple to deal with. A *Bound* literal simply requires substituting the value in for the variable throughout the preimage. For an *IntervalBound* literal, we can construct appropriate inequalities: the literal $IntervalBound(?x, v1, v2)$ produces the inequalities $\geq(?x, v1) \wedge \leq(?x, v2)$. Finally, an *Equal* literal merely requires the learner to choose one variable name and substitute it in for the other.

We will now apply this process to the copying TOP which we have been examining throughout this chapter. Once the instances have been converted to indexical-functional form as demonstrated in Section 5.4.4, the learner can generate a concept using the created literals. Suppose the learned concept was the conjunction of the following literals:

- *Bound*(`Mode-of-Copier`, *Copy*)
- *Equal*(`Location-of-Robot`, `Location-of-Copier`)
- *Has*(*Robot*, `Object-copied`)
- *IntervalBound*(`Number-of-copies-made`, 1, 20)

(Note that other literals, such as *At*(`Robot`, `Location-of-Robot`) would clearly also have been true in all of the positive instances, but were not included in the learned concept as they did not provide any discrimination between positive and negative instances.)

Recall from Figure 5.1 that the TOP has the action `copy(?obj,?x)` and the postcondition $Has(Robot, copies(?n, ?obj))$. The TOP variables were $?obj$, $?x$, and $?n$, corresponding to the IFVs `Object-copied`, `Copier`, and `Number-of-copies-made`, respectively. The other IFVs that are included in the concept are assigned new variable names. `Mode-of-Copier` is assigned to $?v_1$, `Location-of-Robot` to $?v_2$, and

`Location-of-Copier` to $?v_3$. We can now convert the IFVs in the four literals found in the concept to variables, resulting in the following literals:

- $Bound(?v_1, Copy)$
- $Equal(?v_2, ?v_3)$
- $Has(Robot, ?obj)$
- $IntervalBound(?n, 1, 20)$

Each of the three variables $?v_1$, $?v_2$, and $?v_3$ needs to be defined, so the IFVs in the three definitional literals

- *Mode*(`Copier`, `Mode-of-Copier`)
- *At*(`Robot`, `Location-of-Robot`)
- *At*(`Copier`, `Location-of-Copier`)

are converted as well, resulting in:

- $Mode(?x, ?v_1)$
- *At*(`Robot`, $?v_2$)
- $At(?x, ?v_3)$

Now, we remove the literal $Bound(?v_1, Copy)$ from the concept and substitute the constant *Copy* for $?v_1$ throughout the concept. Next, we remove the literal $Equal(?v_2, ?v_3)$ and substitute $?v_2$ for $?v_3$ throughout the concept. (Since both are new variables, it does not matter which one we choose to use as a substitute. But note that if both arguments of an *Equal* predicate are TOP variables, the substitution needs to be done throughout the entire TOP, including the action and postcondition.) Finally, the literal $IntervalBound(?n, 1, 20)$ is replaced with two appropriate inequalities. The resulting concept is thus the conjunction of the following literals:

- $Has(Robot, ?obj)$
- $Mode(?x, Copy)$
- $At(\texttt{Robot}, ?v_2)$
- $At(?x, ?v_2)$
- $\geq (?n, 1)$
- $\leq (?n, 20)$

5.7 Computational Complexity of TRAIL's ILP Algorithms

We now briefly examine the computational complexity of the indexical-functional representation stage of the learning process. In the worst case, the number of IFVs that can be introduced is exponential in the size of the state description. To see this, suppose we have a predicate F(:input :output :output) and our state description is the following:

$F(C_1, C_2, C_2) \wedge F(C_2, C_3, C_3) \wedge F(C_3, C_4, C_4) \wedge \ldots$

If C_1 is a domain constant in this domain, there will be 2 IFVs with value C_2 (representing the two output arguments of $F(C_1, C_2, C_2)$), 2^2 with value C_3 (two for each one with value C_2), 2^3 with value C_4, and 2^{n-1} with value C_n.

Are there limitations that we can place on the computation process that allow us to state more useful bounds on the complexity? It turns out that there are. We merely need to limit the length of the variable creation cycle. If we only do a few cycles of the process, the number of variables created is much more manageable. This restriction is not unreasonable, as real domains rarely contain useful IFVs that are even as far as three steps from the set of input variables.

Let D be this maximum depth of derived IFVs. (We say a variable has depth i if it was created on the ith cycle through the algorithm. TOP variables are of depth

0.) Let L be the number of different literals in the domain that contain at least one output argument. Let M be the maximum arity of any literal.

Now, define the number of IFVs created on the *ith* cycle through the algorithm as N_i. Assuming that all the literals are determinate, each IFV potentially allows for the creation of a new IFV for each output argument of each distinct literal in the domain, or a possible $L(M-1)$ literals. Therefore, we know that:

$N_{i+1} < L(M-1)N_i$

Therefore, for a fixed maximum depth D, the number of derived IFVs is polynomial in the number of TOP variables, the number of distinct literals in the domain, and the arity of these literals.

Now, we examine the process of representing a state description using IFVs. Suppose that N_D IFVs have been created, as defined above. Following the five-part process described in Section 5.4.3, each derived IFV produces one definitional literal and one *Bound* literal describing its value in the situation. This results in at most $2N_D$ literals. At most one inequality literal is generated for each goal variable, so at most N_0 inequality literals are included. Each pair of IFVs may potentially generate an equality literal, so as many as $\frac{{N_D}^2}{2}$ *Equal* literals may be generated. Finally, each input-only literal with M arguments can generate as many as ${N_D}^M$ literals. If we consider the maximum arity of an input-only literal as a constant, the resulting expression is still polynomial in N_D.

Although the worst-case size of the IFV representations of state descriptions is significantly larger than the original representation, in practice the sizes of the representations are quite manageable. There are at least two significant reasons for this discrepancy. First, the worst case analysis assumes that the number of IFVs produced from each TOP variable grows exponentially with depth. In practice, this turns out not to be the case, as each TOP variable generates only a few new IFVs describing its properties. Second, the worst-case number of *Equal* literals and input-only literals occurs only if all of the generated IFVs have identical values. In practice, it is almost never the case that more than a few IFVs have identical values, resulting in far fewer literals than might otherwise be generated.

As evidence for the practicality of the IFV approach in our domains, we tested the

Number of TOP Variables	**Average Size**
1 Variable	4.0 literals
2 Variables	10.0 literals
3 Variables	13.8 literals

Table 5.9: Average Size of State Representations Using IFVs

IFV conversion algorithm on a set of 133 states that were generated during the course of problem solving in the delivery domain. The delivery domain was chosen because it was the only one of the three domains that is truly relational, and thus contains considerably more complicated state descriptions than the other two domains. Initially, the ground first-order state descriptions contained an average of 13.2 literals per description. Since each of these states occurred while learning some TOP, the IFV conversion algorithm was given as input the appropriate set of TOP variables and the domain constant *Robot*. Each state description was then converted into an IFV representation as described in Section 5.4. On average, the resulting state representations contained only 8.5 literals. The average size of the state representations varied considerably depending on the number of TOP variables used when producing the representation, as shown in Table 5.9.

It may seem odd that the IFV representations are on average smaller than the original state descriptions. This difference is easily explained by the fact that given our algorithm, all literals in the IFV representation must be directly or indirectly related either to one of the TOP variables or to one of the domain constants. For instance, in a TOP describing the process of carrying an object, a literal describing the mode of the copier is included in the original state description but not in the IFV representation. The key point, in any case, is the fact that although the IFV representations may theoretically be significantly larger than the original state descriptions, in practice they appear to be quite manageable in size.

5.8 Concept Learning in Other Action-Model Learners

Although no other action-model learning systems have explicitly used ILP in their learning methods, each of the other systems did have to deal with the fact that their state descriptions were structured instances rather than attribute-value instances. Therefore, each of these systems used a learning algorithm which could deal in some way with structured instances.

Shen's LIVE system (Shen 1994) uses his CDL algorithm (Shen 1990) to learn preconditions of operators. The CDL algorithm operates by examining each instance individually. If the current concept classifies it correctly, no learning is done. Otherwise, the algorithm finds some difference between the concept and the misclassified instance and appends this difference to the concept description. Differences are found by starting with a set of objects corresponding roughly to TRAIL's set of TOP variables, and looking for chains of literals built on these objects. This method is powerful enough to learn concepts such as the preimage of the copy TOP introduced in Section 5.2. However, CDL does not have any mechanism for handling noise, nor is there any obvious way of modifying it to do so.

The learning algorithm in Gil's EXPO system (Gil 1992) also operates by a method of difference-finding. EXPO compares an observed negative instance to the most similar previously observed positive instance and computes the set of differences between them. It then does experimentation in the environment to determine which difference should be added to the precondition. This is another example of a form of experimentation which has not been implemented in TRAIL. However, the learning mechanism in EXPO can only add one condition at a time, and thus is unable to learn concepts which depend on more than one literal, such as the copy preimage from Section 5.2. It is also unable to deal with overly specific preconditions or noise in the effects of actions.

Wang's OBSERVER system (Wang 1995*b*) learns preconditions by building a most general and most specific representation in a manner similar to that of the version space algorithm (Mitchell 1982). The most specific representation is built by taking conditions which are common to all of the observed positive instances and replacing constants with variables. Negated conditions are produced by observing differences between the learned concept and any negative instances which match it. The most general representation is learned only from negative instances obtained during practice. Although OBSERVER can recover from mistaken generalizations made during learning, like LIVE and EXPO it assumes that the observed instances are noise-free.

Chapter 6

Examples and Evaluation

This thesis is not primarily an experimental thesis. Unlike more thoroughly studied areas of machine learning such as decision tree induction, the area of autonomous learning agents does not yet have any standardized set of performance tasks on which to evaluate different systems. There are at least two main reasons for this. First, there as yet is no satisfying set of standardized environments in which agents can be tested. Second, the aims of the different agent learning systems are significantly different, so it is difficult to compare them directly. We discuss these issues further in Section 6.4.

This chapter examines the performance of TRAIL in the three experimental domains introduced in Section 1.2. We first illustrate TRAIL's problem-solving behavior through several detailed examples of TRAIL's operation in two different environments, the Botworld construction domain and the SGI flight simulator. Following this, we present some quantitative results on the behavior of TRAIL in the Botworld construction domain and the office delivery domain, demonstrating runs in which TRAIL begins with no knowledge of the domain and learns to behave independently (through the help of an external teacher). Finally, we discuss the behavior of TRAIL in the flight simulator domain and some of the difficulties that affected TRAIL's performance in this domain.

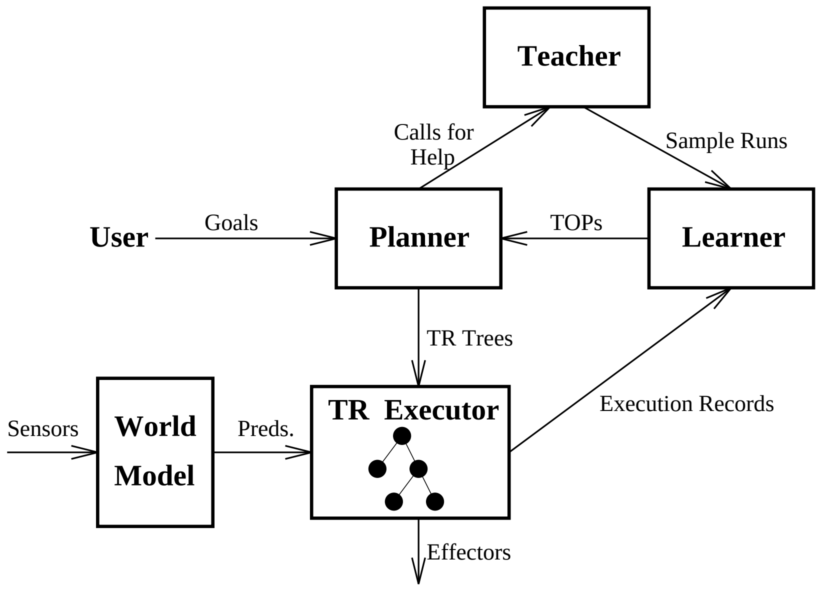

Figure 6.1: TRAIL's Agent Architecture

6.1 The Behavior of the Overall TRAIL Architecture

We begin this chapter by stepping back from the details of the TRAIL learning mechanism to describe the behavior of the integrated TRAIL system at a high level. For convenience, we reproduce the diagram of TRAIL's overall architecture, first shown in Section 3.5. For our purposes in this section, the important components are the planner, the TR executor, the action-model learner, and the teacher.

Once a goal is given to TRAIL by the user, one of the following four scenarios can occur:

- There may already be a tree in the plan library that, when executed, is sufficient to accomplish the goal. In this case, either there were no execution failures during the goal completion, or there were a small number of failures that could be handled by the existing tree. Once the goal is complete, the execution record is given to the learner, which updates its set of TOPs based on the record. Positive instances are generated for each of the TOPs that were successfully executed, while negative instances are generated for any action failures that may have occurred.

- If there is no applicable tree in the plan library, the planner may produce a tree that, when executed, is sufficient to accomplish the goal. From the perspective of TRAIL's learning system, this case is indistinguishable from the case above.

- The planner may not have sufficient knowledge to create a tree to achieve the goal from the current situation. (This is obviously the case the first time a goal is given to TRAIL.) In this case, TRAIL calls the teacher and the learner updates its action models from its observations of the teacher.

- The planner may produce an incorrect tree that is not sufficient to complete the goal. In this case, the tree will fail during execution, resulting in either a timeout failure or a series of action-effect failures. Once a failure is detected, the learner is called, and uses knowledge gained from the failure to modify its set of TOPs. This modified set of TOPs will allow the planner to replan the tree from the current situation (see Section 2.5). Once the corrected plan is complete, execution resumes from the current state of the agent.

  This process of iterated execution and replanning will be repeated until either TRAIL creates a modified tree that is sufficient to achieve the goal, or TRAIL reaches a planning impasse at which it is unable to construct a tree that it expects to complete the goal, and resorts to the teacher.

We can view this process as allowing TRAIL to construct successive approximations to the correct preimages of each TOP. Suppose that the current approximation π^* of the preimage condition π of a TOP τ is too general at some point during learning. Then there is at least one state s such that $s \models \pi^*$ and $s \not\models \pi$. Eventually, some tree will use τ in such a state s. The TOP will fail, and TRAIL will generate a negative instance of the preimage for τ. The learner will thus use this instance to specialize π^*, potentially excluding s.

Suppose on the other hand that the approximation π^* is too specific. Then there is at least one state s such that $s \models \pi$ and $s \not\models \pi^*$. Eventually, TRAIL will be called upon to achieve a goal from such a state s that can only be achieved using τ. TRAIL will be unable to generate a tree to do so, and thus will be forced to call the teacher. The teacher's actions will provide a positive instance of τ succeeding in state s, which the learner can use to generalize π^* to include s.

In this way, TRAIL uses its experience to converge on a useful set of action models. As it sees more goals, it corrects errors in its TOP representations, and thus learns to construct correct plans increasingly often.

6.2 An Extended Example in Botworld

We will now illustrate the overall behavior of the TRAIL architecture, through an example that shows TRAIL learning to grab a bar in the Botworld construction domain. This example is a typical example of TRAIL's performance in this domain, illustrating the process of learning by observing a teacher, the use of the learned teleo-operators in planning, the detection of plan failures, and the process of learning and plan modification that results from the plan failures. It also illustrates a number of the difficulties that TRAIL faces in dealing with an environment where actions take varying lengths of time and have effects that are not always predictable from the current world state.

The bot's possible actions include the atomic actions `grab` and `ungrab` and the durative actions `turn`, `forward`, and `backward`. TRAIL knows the names of these actions and can execute each of them. It also has a set of percepts by which it senses its environment, but has no further domain model. More details on the percepts and actions in Botworld, and on the process of grabbing a bar, are found in Section 1.2.1.

We begin by placing the bot in the state shown in Figure 6.2 and telling TRAIL to grab the bar, which the bot identifies as $Bar1$. This initial state is described as $TooFar(Bar1) \wedge FacingMidline(Bar1)$.

TRAIL currently has no valid TOPs, so it is forced to call the teacher and ask for help. The teacher for this example is a program that runs a simple bar-grabbing routine. In this particular task, the bot turns (left, the default direction) until it is facing directly away from the midline, backs up to the midline, turns to face the bar, moves forward until it is close to the bar, and then successfully executes the `grab`

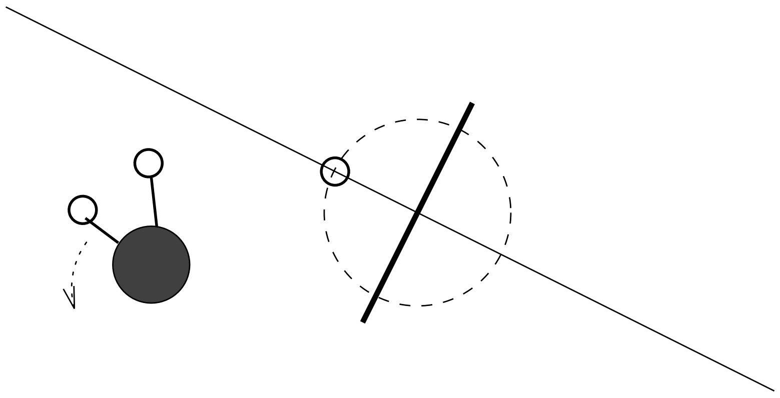

Figure 6.2: Initial State For First Botworld Task

action.

This results in TRAIL learning a set of TOPs from its observation of the teacher's execution record. A partial list of these TOPs is shown in Table 6.1.

Action	**Effect**	**Preimage**
`turn`	$ParallelTo(?x)$	$TooFar(?x)$
`turn`	$\neg FacingMidline(?x)$	$FacingMidline(?x) \wedge TooFar(?x)$
`backward`	$OnMidline(?x)$	$ParallelTo(?x) \wedge TooFar(?x)$
`turn`	$FacingBar(?x)$	$OnMidline(?x) \wedge TooFar(?x)$
`forward`	$AtGrabbingDist(?x)$	$FacingBar(?x) \wedge OnMidline(?x) \wedge$ $TooFar(?x)$
`grab`	$Holding(?x)$	$FacingBar(?x) \wedge OnMidline(?x) \wedge$ $AtGrabbingDist(?x)$

Table 6.1: Partial List of Learned Construction TOPs

This is not a complete list of the learned operators; TRAIL also creates a number of other TOPs such as $\xrightarrow{\texttt{turn}} \neg ParallelTo(?x)$ and $\xrightarrow{\texttt{forward}} \neg TooFar(?x)$ that are included in TRAIL's operator set but do not turn out to be useful for planning in bar-grabbing tasks. TRAIL in fact has learned eight TOPs at this point; the six above are the only ones that are used in subsequent tasks.

Also note that the preimages of a number of these operators are overly specific. One of TRAIL's learning heuristics (discussed in Section 5.5.5) is that if a TOP achieves a postcondition $F(?x)$, then any literals in the state description that refer to $?x$ will be included in the preimage so long as they do not decrease the number

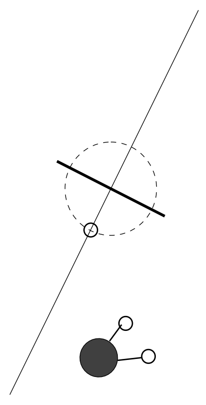

Figure 6.3: Initial State For Second Botworld Task

of positive instances covered. Since TRAIL has only seen the action `turn` achieve $FacingBar(?x)$ when the bot was on the bar midline and too far from the bar, the literals $OnMidline(?x)$ and $TooFar(?x)$ are included in the preimage for the $\xrightarrow{\texttt{turn}}FacingBar(?x)$ TOP.

Now we give TRAIL another bar-grabbing task, shown in Figure 6.3. Using the TOPs learned in the first run, TRAIL creates the plan shown in Figure 6.4. However, the execution of this plan does not work as TRAIL expected. The bot turns around to the left until $ParallelTo(Bar1)$ holds, at which point node N_1 becomes active and TRAIL begins executing the TOP $\xrightarrow{\texttt{backward}}OnMidline(?x)$. However, the bot at this point is facing the midline, so it continues to back up until TRAIL detects a time-out failure on this TOP. This failure produces a negative instance $ParallelTo(?x) \wedge TooFar(?x) \wedge FacingMidline(?x)$. Since TRAIL had earlier observed the positive instance $ParallelTo(?x) \wedge TooFar(?x)$, it changes the preimage for the TOP to $ParallelTo(?x) \wedge TooFar(?x) \wedge \neg FacingMidline(?x)$.

Given this updated preimage, TRAIL no longer expects node N_1 to execute successfully, as the preimage condition for the $\xrightarrow{\texttt{backward}}OnMidline(?x)$ TOP does not hold. Therefore, the planner revises the plan by changing the condition on node N_1 to include the literal $\neg FacingMidline(?x)$. Since this new condition does not hold in the current situation, the planner is forced to extend the plan using the $\xrightarrow{\texttt{turn}}\neg FacingMidline(?x)$ TOP to achieve the condition on N_1, resulting in the modified plan shown in Figure 6.5.

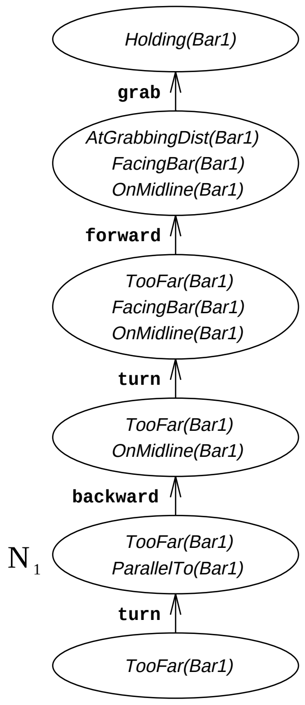

Figure 6.4: An Initial Plan For Bar-Grabbing

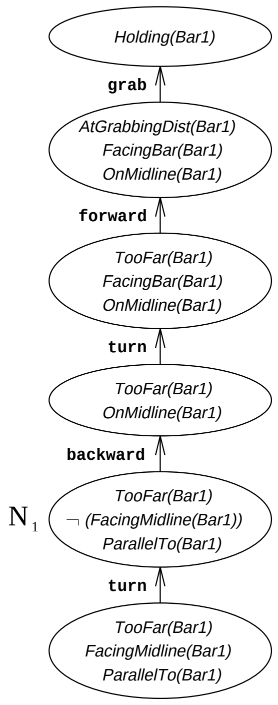

Figure 6.5: Modified Plan For Bar-Grabbing

However, this plan does not work either. The bottom node in the plan is activated, and the bot begins turning, which soon results in the condition $ParallelTo(?x)$ becoming false. This is an activation failure of the second type discussed in Section 4.4.2, in which the maintenance condition of the node has become false. Therefore, $ParallelTo(?x)$ is added as a delete list element of the $\xrightarrow{\texttt{turn}}\neg FacingMidline(?x)$ TOP. Since the delete list element initially has a probability of 1, this TOP can not be used to achieve $\neg(FacingMidline(Bar1)) \wedge ParallelTo(Bar1)$. The alternative of first achieving $\neg(FacingMidline(Bar1))$ and then using the TOP $\xrightarrow{\texttt{turn}}ParallelTo(?x)$ cannot be planned either, as TRAIL has at present included $\neg(FacingMidline(?x))$ in the delete list of the $\xrightarrow{\texttt{turn}}ParallelTo(?x)$ TOP.[1]

At this point, the planner cannot revise the plan to cover the current situation, so TRAIL is forced to call the teacher in order to complete the goal. The teacher does so, and TRAIL once again learns from its observations. In the process, the teacher ends up backing up to be on the midline, producing another positive instance of the TOP $\xrightarrow{\texttt{backward}}OnMidline(?x)$. Unfortunately, this instance takes much longer than the first positive instance, which raises the mean completion time of the TOP. As we saw in Section 4.4.3, TRAIL has a heuristic that causes it to discard timeout failures for which the timeout interval is shorter than the current average execution time of the TOP. (This heuristic allows TRAIL to discard false negative instances that were collected early in the learning process.) Thus, after the teacher has achieved $OnMidline(?x)$, TRAIL observes that the time taken by the earlier timeout instance is considerably less than the new timeout period, and thus discards the negative instance. When the TOP preimages are recomputed at the end of the run, the learner resets the preimage of the $\xrightarrow{\texttt{backward}}OnMidline(?x)$ TOP, for the moment, to $ParallelTo(?x) \wedge TooFar(?x)$. Since there is no currently active plan, TRAIL's planner does not need to do any updating.[2]

[1]TRAIL will eventually learn that achieving $ParallelTo(?x)$ by turning deletes the literal $\neg(FacingMidline(?x))$ only about 50% of the time.

[2]If the plan from Figure 6.5 were stored in a plan library, it would need to be modified. Plan caching in the plan library is not used in this example.

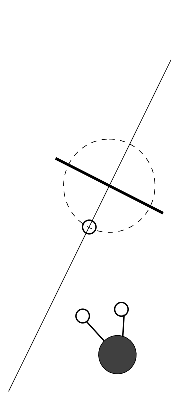
Figure 6.6: Initial State For Fourth Botworld Task

For our third run, we restart the bot in a state similar to the state shown in Figure 6.2 from the first run. The planner constructs the same plan shown in Figure 6.4. This run is completely straightforward, as the plan succeeds with no TOP failures.

Now, we try starting the bot in the state shown in Figure 6.6. Again, the planner constructs the same plan, but this time, as in the second run, the bot turns until it is parallel to the bar and facing the bar midline, at which point it begins backing up. Again, a timeout failure is detected for the $\xrightarrow{\texttt{backward}} OnMidline(?x)$ TOP, and as before, the preimage is revised to $ParallelTo(?x) \wedge TooFar(?x) \wedge \neg FacingMidline(?x)$.

At this point, the planner does not know how to revise the plan, so the teacher is called. But this time, the bot is already parallel to the bar, so the teacher simply moves the bot forward until it reaches the midline. Thus, TRAIL learns the new TOP $\xrightarrow{\texttt{forward}} OnMidline(?x)$ with preimage $ParallelTo(?x) \wedge FacingMidline(?x) \wedge TooFar(?x)$.

For the fifth run, we try something different by putting the bot very close to the bar, as shown in Figure 6.7. The condition $TooFar(Bar1)$ no longer holds in the initial state. However, almost every TOP learned so far has included $TooFar(?x)$ in its precondition, so the planner is unable to produce a plan for this situation.

TRAIL again calls the teacher, which successfully grabs the bar by turning so it is parallel to the bar, moving forward until it is on the midline, turning to face the bar, and executing `backward` until it is at the correct distance. (This sequence actually has to be repeated several times, as the $FacingBar$ predicate is not very precise at

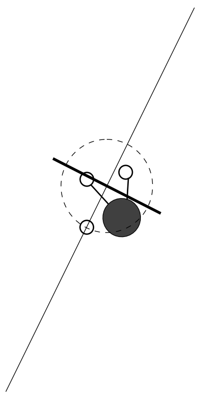

Figure 6.7: Initial State For Fifth Botworld task

short distances. Therefore, the bot may not be facing the bar exactly, and backing up sometimes causes $OnMidline$ to become false. TRAIL is quite capable of dealing with such inconsistency.[3]) From this training run, TRAIL learns a new TOP for the action $\xrightarrow{\texttt{backward}} AtGrabbingDist(?x)$, with preimage $FacingBar(?x) \land OnMidline(?x) \land \neg FacingMidline(?x)$. (The $\neg FacingMidline(?x)$ literal is essentially random noise due to the action failures mentioned above.) Meanwhile, new positive instances are generated for a number of other TOPs, allowing the preimages of these TOPs to be generalized. In particular, the literal $TooFar(?x)$ is removed from the preimages of the TOPs $\xrightarrow{\texttt{forward}} OnMidline(?x)$, $\xrightarrow{\texttt{turn}} ParallelTo(?x)$, and $\xrightarrow{\texttt{turn}} FacingBar(?x)$, as well as a number of other TOPs.

Finally, we position the bot once again in the same position as in the fourth run, as shown in Figure 6.6. Now, the planner comes up with the somewhat more complex plan shown in Figure 6.8. (Note that several of the `turn` nodes in the tree are annotated with probabilities, due to the possible side effects of the turn operations.)

Execution of this plan proceeds as normal until the bot is on the bar midline and facing the bar. At this point, the top nodes in both branches of the tree are active. The arbitrary tie-breaking rule used in TRAIL is to choose the rightmost of equivalent nodes, so the bot begins backing up hoping to achieve $AtGrabbingDist(Bar1)$.

[3]Again, while we could easily change the $FacingBar$ predicate to make it more reliable, we continue to use the predicate as it was originally defined. Real environments may well have predicates that are not completely accurate.

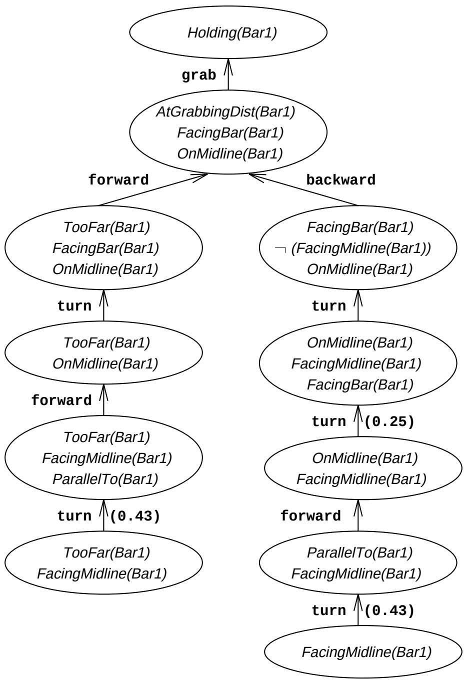

Figure 6.8: A More Complex Bar-Grabbing Plan

Naturally, TRAIL detects a timeout failure fairly quickly on the recently learned $\xrightarrow{\texttt{backward}} AtGrabbingDist(?x)$ TOP, causing the learner to generate a new negative instance and add the condition $\neg(TooFar(?x))$ to the TOP preimage. Now, TRAIL calls the planner with the new set of TOPs to revise the plan. The planner cannot find any way to make the right branch work in the current situation, so the branch is pruned, and the plan is left as shown in Figure 6.9. This plan now succeeds.

At this point, TRAIL has learned much of the information it needs in order to grab a bar. It still needs to make several modifications to its TOPs, however, in order to be able to produce correct plans. For instance, at some point it will learn that the TOP $\xrightarrow{\texttt{turn}} ParallelTo(?x)$ does not always delete the condition $\neg FacingMidline(?x)$, and it will also learn that the TOP $\xrightarrow{\texttt{forward}} AtGrabbingDist(?x)$ is not always reliable (due to imprecision in the set of Botworld predicates.) A summary of the performance of TRAIL over a long series of bar-grabbing tasks is given in Section 6.5.

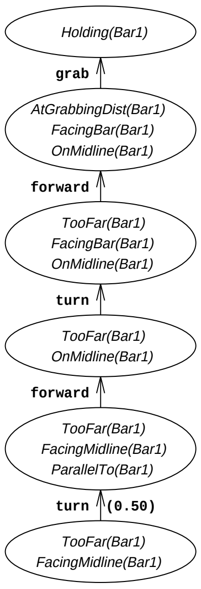

Figure 6.9: A Simplified Version of the Bar-Grabbing Plan

6.3 An Extended Example in the Flight Simulator

In this section, we will observe the performance of TRAIL in the flight control of a Cessna aircraft in the Silicon Graphics Inc. Flight Simulator.[4] As we discussed when we introduced the domain in Section 1.2.3, the flight simulation domain is considerably more difficult to learn from than the Botworld domain. As a result, this section only examines a small set of fairly simple flight tasks. Nevertheless, these examples illustrate TRAIL's mechanism for learning in continuous domains, a series of plan failures and replanning episodes, and the interaction between learning and continuing plan execution. We will return to the high-level issues raised by TRAIL's performance in the flight simulator domain in Section 6.7.

[4]Again, we gratefully acknowledge the help of Seth Rogers of the University of Michigan in providing code and assistance with the simulator interface.

6.3.1 Learning to Take Off

TRAIL's first task is to learn to get the plane off the ground. We express this goal to TRAIL by simply asking it to achieve a particular altitude. In this case, the goal we give it is $Altitude(150)$, which is converted by our interface into the interval goal $Altitude([125..175])$ (The interval is necessary since the altitude sensor might never read exactly 150.)

In the first test run, naturally the agent cannot make a plan to achieve the goal, so it calls on the teacher. The teacher takes off using a simple hand-coded flight control routine that successfully gets the plane off the ground and to 150 feet. This flight control program increases the throttle to 75%, waits until the plane's on-ground speed reaches 60 knots, then begins controlling the stick through the `up` PID controller. (As discussed in Section 1.2.3, most of TRAIL's flight actions are calls to PID controllers. These calls are durative actions that attempt to keep the climb rate of the aircraft at a particular value.) This continues until the plane reaches approximately 100 feet, at which point control is given to the `up-slow` controller. (The teacher uses `up-slow` to complete any climb since it makes it easier to level the plane off at a particular altitude. This is a case where Fuzzy TR Trees (see Section 7.2) would make the flying task considerably simpler.)

From this test run, TRAIL learns the set of TOPs shown in Table 6.2.

Action	**Effect**	**Preimage**
`inc-throttle`	$Throttle(?x)$	$Bound(?x, 75)$
`wait`	$Speed(?x)$	$Bound(?x, 60)$
`up`	$Speed(?x)$	$Bound(?x, 65)$
`up`	$ClimbRate(?x)$	$IntervalBound(?x, 1, 6)$
`up`	$Altitude(?x)$	$Bound(?x, 114)$
`up`	$\neg OnGround$	T
`up-slow`	$ClimbRate(?x)$	$Bound(?x, 3)$
`up-slow`	$Altitude(?x)$	$Bound(?x, 144)$

Table 6.2: Learned TOPs in the Flight Simulator

Note that, except for the *Bound* constraints on the goal variables, the preimages in this example are all very general. In the bar-grabbing example in the previous section, the bar served as a target object, and literals relating to the target bar were included in the preimage by default. In contrast, in this example there is no target object, so TRAIL does not have any literals that it can include by default. However, since each of the TOPs (except for $\xrightarrow{\text{up}}\neg OnGround$) involves setting a numerical parameter, the binding of each goal variable is included using a *Bound* literal. Also, during this run, the `up` action set the value of *ClimbRate* to a number of different values, so the binding on the TOP variable $?x$ for the $\xrightarrow{\text{up}}ClimbRate(?x)$ TOP is actually the interval [1..6].

Once learning is complete, we get the plane back to the initial state by giving TRAIL the goal $OnGround \wedge Speed([-5..5])$. The process of learning to land in TRAIL is described in detail in Section 6.3.2.

Now, with the plane at a stop on the ground, we again give TRAIL the goal $Altitude([125..175])$. TRAIL's initial plan is as shown in Figure 6.10.

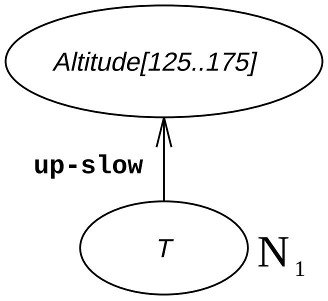

Figure 6.10: Initial Plan For Taking Off

Obviously TRAIL has overgeneralized the preimage of the TOP that uses `up-slow` to achieve $Altitude(?x)$. The `up-slow` PID controller controls the stick to set the climb rate to approximately 300 feet per minute, but it clearly can only do this if the plane is in the air, or on the ground and moving sufficiently fast to take off. But TRAIL begins executing the plan shown above in a state in which the plane is stopped and on the ground. Thus, the `up-slow` action has no effect, so TRAIL detects a timeout for the TOP $\xrightarrow{\text{up-slow}}Altitude(?x)$ and generates a negative instance for the learner. The learner now has several learning instances, which look something like the following:

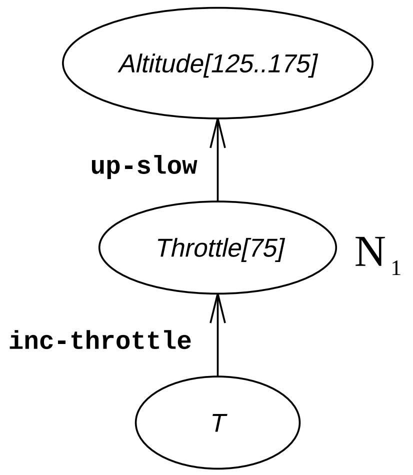

Figure 6.11: Revised Plan For Taking Off

$+ \quad Altitude(100) \wedge Speed(66) \wedge Throttle(75)$
$+ \quad Altitude(105) \wedge Speed(67) \wedge Throttle(75)$
$\quad \ldots$
$+ \quad Altitude(130) \wedge Speed(69) \wedge Throttle(75)$
$- \quad OnGround \wedge Altitude(0) \wedge Speed(0) \wedge Throttle(0)$

TRAIL's ILP learner then revises the preimage for the TOP, selecting $Throttle(75) \wedge Bound(?x, 141)$ from several equally plausible candidates. Thus, the condition on node N_1 in Figure 6.10 is changed from T to $Throttle(75)$. Since $Throttle(75)$ does not hold in the current state, TRAIL's planner extends the tree as shown in Figure 6.11.

The `inc-throttle` command soon resets the throttle to 75, and now TRAIL begins to apply `up-slow` once again in hopes of increasing the altitude. However, at this point something odd occurs. Recall that the teacher did not apply the `up-slow` action until the plane was already 100 feet above the ground. The actual time during which `up-slow` was being applied was relatively short. Therefore, TRAIL detects a timeout failure in the $\xrightarrow{\texttt{up-slow}} Altitude(?x)$ TOP before the aircraft has even left the ground. Thus, TRAIL now has the following instances for the TOP:

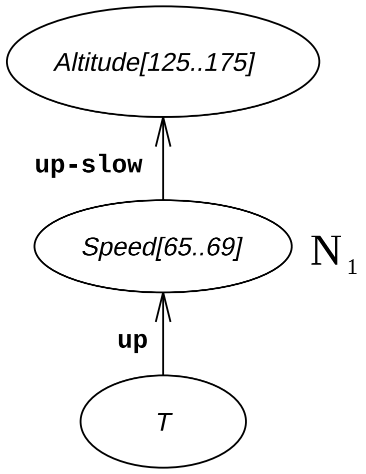

Figure 6.12: Third Plan For Taking Off

$+\ \ Altitude(100) \wedge Speed(66) \wedge Throttle(75)$

$+\ \ Altitude(105) \wedge Speed(67) \wedge Throttle(75)$

$\ldots$

$+\ \ Altitude(130) \wedge Speed(69) \wedge Throttle(75)$

$-\ \ OnGround \wedge Altitude(0) \wedge Speed(0) \wedge Throttle(0)$

$-\ \ OnGround \wedge Altitude(0) \wedge Speed(5) \wedge Throttle(75)$

$-\ \ OnGround \wedge Altitude(0) \wedge Speed(10) \wedge Throttle(75)$

$\ldots$

TRAIL's ILP learner again revises the preimage, this time to $Speed([65..69]) \wedge Bound(?x, 141)$. Thus, the condition on node N_1 is changed to $Speed([65..69])$. Now, TRAIL observed previously that the aircraft speed increased to 65 as the teacher was taking off. Therefore, it believes that the up action was responsible for the increase in speed, and constructs the revised plan shown in Figure 6.12.

Clearly, this is an incorrect plan, but in this case, due to unexpected interactions between replanning and the ongoing execution of the tree, it happens to work. Recall that earlier in this training run, TRAIL had set the throttle to 75. Therefore, at the point at which TRAIL begins executing the plan from Figure 6.12, the aircraft is already moving and building up speed. Eventually, the aircraft reaches a speed of 65 knots, takes off, and climbs to over 125 feet.[5] Thus, the interactions between the learning process, the replanning mechanism, and the continuing execution of the teleo-reactive plan have produced a result that differs from what would have been produced by either the learning or the execution alone. In this case, it has allowed TRAIL to take off without resorting to the teacher, despite the fact that the plan shown in Figure 6.12 is obviously incorrect. In general, the interaction between learning, replanning, and execution can be very complex, and is beyond the scope of this thesis to analyze. This interaction is also another reason why it is difficult to evaluate the performance of the individual components of TRAIL, as we will discuss in Section 6.4.

Meanwhile, TRAIL has still not learned a correct plan for taking off. Thus, once TRAIL has landed and stopped (as described in the next section), when we give it the goal for a third time, it constructs the same plan as was shown in Figure 6.12. Now, the lowest node in the tree is the active node, and the action `up` is applied. Once again, TRAIL sits on the runway, hoping that the `up` controller will cause the plane to accelerate to 65 knots. Once this does not happen, TRAIL detects a timeout failure, and revises the preimage for the TOP $\overset{\texttt{up}}{\rightarrow} Speed(?x)$ to be $Bound(?x, 65) \wedge Throttle(75)$. Note that since the PID controller does nothing while the plane is still too slow to take off, the `up` action is simply being used as a substitute for the `wait` action. In any case, the revised preimage results in the plan shown in Figure 6.13.

This plan causes TRAIL to increase the throttle to 75%, accelerate by applying the `up` controller until the plane actually takes off, and then apply the `up-slow` controller once the speed reaches 65 knots. The aircraft quickly reaches an altitude of over 125 feet and the goal is complete.

It would be nice if this were the end of the story, and that TRAIL had now completely learned how to take off. However, this success has a rather unexpected effect on one of the TOPs, namely $\overset{\texttt{up-slow}}{\rightarrow} Altitude(?x)$. Recall that earlier, the TOP was only executed for a short time before the goal became true. In this case, `up-slow`

[5]The PID controller is intelligent enough that even though it was called too early, it did not pull the stick all the way back while the plane was on the runway – this would have caused the plane to stall shortly after takeoff.

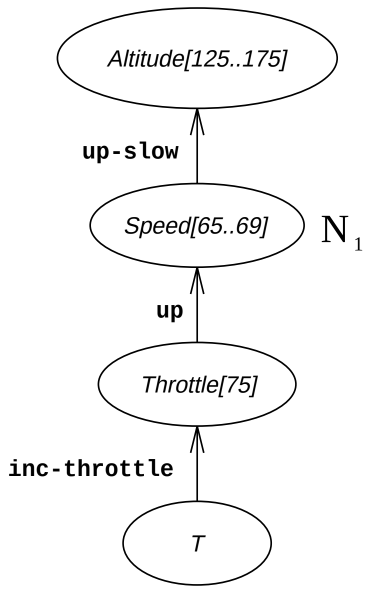

Figure 6.13: Successful Plan For Taking Off

was applied somewhat earlier, shortly after the plane left the ground. The length of the action was not long enough to cause a timeout failure, but TRAIL now has an execution success that is significantly longer than the earlier observed successes. As we saw in Section 6.2, this success causes TRAIL to lengthen the average completion time for the TOP, and thus prune the earlier negative example. Therefore, at the end of the run, the preimage for the TOP $\overset{\texttt{up-slow}}{\rightarrow} Altitude(?x)$ is reset to T.

The eventual effect of this overgeneralization is that the next time TRAIL takes off, it goes through what is essentially a repeat of the experience it had on its second run. However, this time the timeout failure is significantly longer, and will be retained as a negative instance. In this way, the preimage for the TOP is stabilized as $Speed([65..69])$.

The training described above is a fairly typical example of the learning behavior of TRAIL in the flight simulator domain. In most of the test runs we have done, TRAIL learns a plan similar to that in Figure 6.13 within four or five takeoffs. The condition on node N_1 varies somewhat from run to run, often including conditions on the climb rate or altitude instead of the speed. However, the overall effect of the plan is the same.

Unfortunately, in rare cases, the interaction between TRAIL's learning system and the simulator causes unexpected behavior to occur. Consider the following rather involved example. In one run, while landing the aircraft after a successful take-off, TRAIL observed that the altitude became 150 while the teacher was decreasing the throttle in preparation for landing. From this observation, TRAIL developed a TOP for $\xrightarrow{\texttt{dec-throttle}} Altitude(?x)$. After a futile attempt to use this TOP for take-off, TRAIL then tried using the previously learned TOP $\xrightarrow{\texttt{inc-throttle}} Altitude(?x)$ to achieve $Throttle(75)$ as was shown in Figure 6.11. Unfortunately, the `dec-throttle` operator had already set the throttle to a very negative value, so the `inc-throttle` action (normally very consistent) timed out before achieving $Throttle(75)$. Thus, the preimage for $\xrightarrow{\texttt{inc-throttle}} Altitude(?x)$ was reset to $Speed([10..15])$. This TOP created a number of obvious planning problems, and over the course of several plan revisions and teacher calls, several activation failure negative instances of the TOP were also generated. Thus, the TOP had a number of false negative instances associated with it, which proved to be extremely difficult for the learner to deal with. Even after 10 further training episodes, TRAIL was unable to learn to take off correctly until it was restarted with no domain knowledge.

Learning failures such as the one described in the previous paragraph occurred during approximately 10-20% of TRAIL's learning runs, almost always due to some TOP that was learned during landing. Thus, if we had turned off all learning behavior during the landing runs that separated each takeoff, the behavior of TRAIL might have been significantly improved. However, turning off the learning mechanism during parts of the training requires some rather arbitrary intervention by a human observer, and is clearly not in keeping with the spirit of a multi-domain autonomous learning agent.

6.3.2 Learning to Land

In general, the problem of landing an airplane is considerably more difficult than the problem of taking off. This is due to the fact that the pilot must coordinate the xy navigation of the plane with the descent such that the plane reaches altitude zero at the same time it arrives at the runway. In addition, the pilot must make sure that

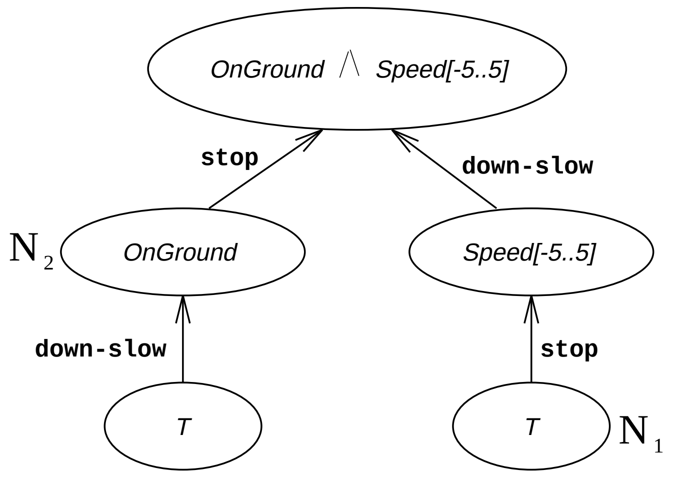

Figure 6.14: Initial Plan For Landing

the plane is not descending too fast, traveling too fast, or at too much of an angle to the ground. In an initial attempt to get TRAIL to land successfully, we simplified the problem considerably by extending the runway indefinitely from the startoff point. Since the takeoff examples discussed in the previous section did not cause the plane to change its direction of flight, no navigation is required in order to land the plane.

We express the goal of landing as a conjunction of two literals, *OnGround* and $Speed([-5..5])$, as we want the airplane to be (approximately) stopped as well as on the ground. When this task is initially given to TRAIL, it naturally calls the teacher, as it has no TOPs that are useful in planning the task. The teacher's landing process is fairly simple - after it resets the flaps and throttle, it uses the `down` PID controller to get the plane down to 50 feet, then uses the `down-slow` controller until the plane is on the ground. At this point, it uses another durative action called `stop` that turns off the throttle and puts on the brakes. The `stop` action is executed until the speed drops below 5 knots, and the goal is complete.

The two key TOPs that TRAIL learns from these observations of the teacher are $\xrightarrow{\texttt{down-slow}} OnGround$ and $\xrightarrow{\texttt{stop}} Speed(?x)$ where $?x$ is bound to the value 5. The second time TRAIL is told to land the plane, it uses these TOPs to construct the (rather amusing) plan shown in Figure 6.14.

Note that there is no information about either the throttle setting or the flaps in this plan, as TRAIL does not initially know that these values are relevant to the landing process. And in fact, for this simple task of landing the Cessna from low altitudes while flying straight, they are not terribly relevant. The Cessna's flight is sufficiently robust (at least in the SGI simulator) that it can land almost regardless of the flaps and throttle settings, as we shall soon see.

The truly remarkable thing about the plan in Figure 6.14 is that it actually works most of the time! Initially, the default node selection mechanism chooses node N_1 as the active node and begins executing the `stop` action. Thus, TRAIL turns off the throttle completely, and begins applying the brakes. The brakes have no effect while the airplane is in the air, of course, but the closed throttle causes the plane to gradually lose speed and altitude, generally reaching a stable point at around 40 knots and an altitude loss of 480 fpm. The dynamics of the Cessna are sufficiently stable that the plane neither goes into an uncontrolled dive nor stalls, even with the throttle turned off. Thus, the plane glides safely to the ground, where node N_2 becomes active, and the `stop` action continues until the plane comes to a stop.

However, there are rare occasions when the plan does fail, usually involving cases where the plane was climbing or descending steeply when execution began. So eventually, TRAIL will begin executing the plan and the `stop` action will cause the plane to stall before reaching the ground, usually resulting in a crash. TRAIL's instance generation mechanism will thus generate a negative instance for the $\overset{\texttt{stop}}{\rightarrow} Speed(?x)$ TOP. TRAIL's learner then changes the preimage for this TOP to *OnGround*. Once this is done, the right branch of the plan in Figure 6.14 no longer works and is pruned, leaving TRAIL with a generally successful landing plan.

6.3.3 Learning Level Flight

We have also tested TRAIL on the task of achieving level flight starting from the ground. For these experiments, we defined "level flight" as achieving a particular altitude with a climb rate of approximately zero. This goal is represented as $Altitude([125..175]) \wedge ClimbRate([-1..1])$. (Of course, this instantaneous goal is not equivalent to achieving a state of *stable* level flight; we will return to this issue in Section 6.7.)

The two subgoals of the conjunction $Altitude([125..175]) \wedge ClimbRate([-1..1])$ could conceivable be achieved in either order. However, TRAIL quickly learns that changing the value of *Altitude* using the `up` or `up-slow` action has the side effect of changing *ClimbRate*. Therefore, it learns that it must achieve $Altitude([125..175])$ and then achieve $ClimbRate([-1..1])$.

The process of learning to achieve the first subgoal, $Altitude([125..175])$, is essentially identical to the learning process described in Section 6.3.1. However, the process of learning to achieve the second subgoal, $ClimbRate([-1..1])$, is more complicated. When the teacher is achieving the goal, the sequence of actions it uses to level off is: decrease the throttle to 50%, lower the flaps to 0, and apply the `level` PID controller. This sequence of actions is necessary to allow for stable flight, but the climb rate usually becomes 0 while the teacher is lowering the throttle or the flaps. Therefore TRAIL learns TOPs such as $\xrightarrow{\texttt{dec-throttle}} ClimbRate(?x)$ and $\xrightarrow{\texttt{dec-flaps}} ClimbRate(?x)$, rather than the more accurate TOP $\xrightarrow{\texttt{level}} ClimbRate(?x)$. When applied during plan execution, these TOPs usually work, but they do not always lead to very robust plans. A typical successful plan learned during training is shown in Figure 6.15.

However, the task of learning to achieve level flight is significantly more difficult than simply learning to take off or to land. If the training examples and plan executions go smoothly, TRAIL quickly learns a reasonable plan such as the one in Figure 6.15. However, if anything goes wrong, the interactions between variables such as the altitude and the climb rate make it difficult for TRAIL to learn correct models of the actions. Thus, in many training runs, TRAIL has significant difficulty in learning a dependable plan for achieving level flight. The reasons for these difficulties are analyzed in much more detail in Section 6.7.

6.4 Evaluation Metrics for Autonomous Learning Systems

How can we evaluate the performance of TRAIL? Of course, the learning algorithm could be tested against a set of standard datasets as is commonly done in the concept

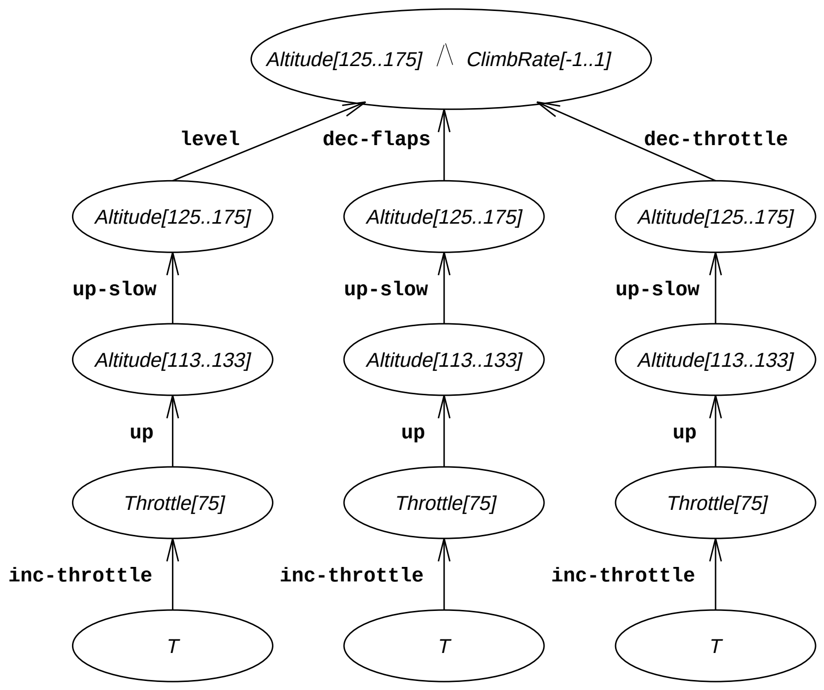

Figure 6.15: An Initial Plan For Level Flight

learning community (Murphy & Aha 1994), and more recently has begun to be done in the area of Inductive Logic Programming (Kazakov, Popelinsky & Stepankova 1996).

However, this type of testing would be misleading in at least two respects. First, the learning problems contained in the standardized databases may not be at all similar to the learning problems faced by TRAIL in computing action preimages. Accurate performance on external datasets does not necessarily imply that useful action models will be created. This is particularly true if the datasets used are small artificial ones, as have been used to demonstrate many ILP algorithms.

Secondly, since TRAIL is an integrated learning system, there is much more to it than the concept learning algorithm. Even if the concept learner is highly effective, we still need some way to evaluate the other components of the system - instance generation, experimentation, planning, and teleo-reactive execution. Furthermore, as we saw in Section 6.3.1, these components can interact in complicated and unexpected ways. Fortunately, there is a reasonable measure that can be used to test all of the components of TRAIL at once.

6.4.1 An Evaluation Metric for TRAIL

Recall that the objective of TRAIL is to learn to achieve goals autonomously in its environment, and that in order to do so, TRAIL is allowed to call upon a teacher during instruction. Therefore, we can consider that TRAIL has been successful if its learned action models allow it to complete goals without resorting to the teacher.[6] Suppose TRAIL is given a series of goals, drawn from some fixed underlying distribution. Define P_n as the probability that it will be able to solve the nth goal in the sequence without resorting to the teacher, given that it has solved the first n-1 goals (using the teacher when necessary.) Clearly, if the agent begins with no initial knowledge, $P_1 = 0$. If the agent is learning, this P_n should increase with n. And ideally, we would like P_n to approach 1 as n increases, thus showing that TRAIL has become independent of the teacher.

Our metric for evaluating TRAIL is based on this probability. In each run of the system, we start it out with no knowledge of the action models that describe the domain. We then give it a series of M goals $G_1 \ldots G_M$ drawn from some distribution. On each goal, TRAIL uses its planning, learning, and execution mechanisms, calling the teacher if it reaches a planning impasse. (Of course, on the first goal it *must* call the teacher, as it has no domain knowledge.) We then simply record whether TRAIL calls the teacher on each goal. Define W_i as 1 if TRAIL managed to complete goal G_i independently, and 0 otherwise.

Now, suppose we repeat K runs of the system as described above. In order to estimate P_n, we simply need to count the percentage of the time over the K runs where TRAIL succeeded independently on the nth goal:

$$P_n \approx \frac{\sum_{K runs} W_n}{K}$$

(In each of the K cases, the learner has successfully completed n-1 goals, and it

[6]Since we are primarily interested in measuring TRAIL's learning behavior, TRAIL's plan caching is turned off for these experiments.

completes the nth one independently $\sum W_n$ times.)

6.4.2 Choosing Domains for Evaluation

Once we have come up with a suitable evaluation metric, it still remains to choose an appropriate domain in which to use the metric to evaluate the system. Although a few testbeds for autonomous agents have been proposed, such as the Tileworld domain (Pollack & Ringuette 1990), there is not yet any widespread agreement on a standard domain in which to evaluate autonomous systems. First, the goals and assumptions of various autonomous systems differ, making it difficult to construct benchmarks on which to measure them. For example, it is difficult to compare a reinforcement learning system with, say, an autonomous map-learning robot. Second, there is an ongoing debate within the agents community over the relative merits of physical and simulated domains - physical domains such as mobile robots present formidable practical difficulties for experimentation, while it is often argued that simulated domains are "too easy" and remove most of the complexities of the real world.[7] Finally, the problems in any particular domain can often be solved more simply by a special-purpose program than by a general-purpose learning and planning agent. In order to show that our agent is useful, we must be able to evaluate it over several domains. In particular, we would like to show that a single agent, once trained, will be able to achieve goals successfully in multiple domains. (Several proposed test domains, as well as the more general issues relating to the selection of domains for evaluating agent architectures, are discussed in much greater detail in Hanks, Pollack & Cohen (1993).)

In this thesis, however, we are not really attempting to compare the performance of TRAIL with other agent architectures, as there do not seem to be any other architectures with which it can be directly compared. Instead, we are simply attempting

[7]Our position on this issue is that from a practical point of view, simulated domains will be necessary for doing any systematic agent experiments, and that the results from simulated domains can give us interesting and useful information about our agents, as long as we do not claim that these results will transfer directly to a physical system. We also note that a third possibility is to make use of domains such as virtual reality and the internet, in which the simulation itself *is* the real environment.

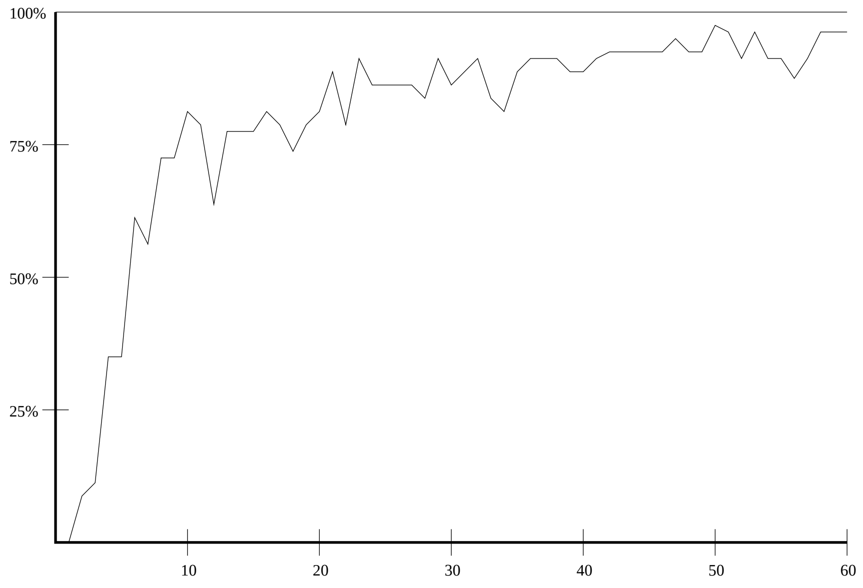

Figure 6.16: Mean Success Rate in the Construction Domain as a Function of the Number of Tasks Completed

to show that the TRAIL approach can be successful in a variety of domains. For that reason, we have implemented TRAIL in three fairly different domains: the Botworld construction domain, the office delivery domain, and the flight simulator. In the remainder of this chapter, we present a more systematic analysis of the performance of TRAIL in each of these domains. Section 1.2 covers each of the domains in more detail and discusses the differences among them from the perspective of action-model learning.

6.5 Performance in the Botworld Construction Domain

The first domain on which TRAIL is tested is the Botworld construction domain. Each task in these experiments is simply to grab a bar from a randomly selected starting position. As we saw in Section 6.2, the exact actions needed to grab the bar vary depending on the orientation of the bar relative to the bot, and on whether the bot is initially too far from the bar or too close to it.

Figure 6.16 displays the estimate of P_n, based on 43 runs of 60 tasks each. Learning is clearly occurring, with the success rate reaching about 94% by the 40th run. In some ways this task is not that difficult; probably most reinforcement learning systems would have an easy time with it.[8] However, it is significantly beyond the power of any of the other existing action-model learning systems. Most of the actions are durative, which would be difficult to represent in a straight STRIPS framework, and there is a significant amount of noise in the domain, primarily due to perceptual aliasing. (See Section 1.2.1 for more details.)

It is interesting to note that the success rate appears to level off around 94%. This appears to be due to one particular case, that of the TOP of using `backward` to achieve $AtGrabbingDist(?x)$. Normally, what happens when the bot is too close to the bar is that the teacher gets the bot to the bar midline, turns it to face the bar, and moves it backward until it reaches the correct distance. Thus, we would expect that the preimage of the TOP $\xrightarrow{\texttt{backward}} AtGrabbingDist(?x)$ should be $OnMidline(?x) \wedge FacingBar(?x) \wedge \neg(TooFar(?x))$. However, TRAIL occasionally observes situations where the bot moves backwards from some position where it is too close to the bar but not on the midline or facing the bar. Thus the preimage is generalized to include these new situations, and becomes just $\neg(TooFar(?x))$. Since this condition is sufficient to guarantee that `backwards` will achieve $AtGrabbingDist(?x)$, TRAIL never sees any negative examples of the TOP, so the preimage is never specialized.

So what is the problem with this? After all, $\neg(TooFar(?x))$ is in fact the correct preimage of the TOP. The problem lies in the delete list associated with the TOP. Consider the predicate $OnMidline(?x)$. If $FacingBar(?x)$ is true then the $\xrightarrow{\texttt{backward}} AtGrabbingDist(?x)$ TOP will not make $OnMidline(?x)$ false. However, if $FacingBar(?x)$ does not hold then it may well delete $OnMidline(?x)$. Now, suppose the agent is too close to the bar, and is attempting to achieve $OnMidline(?x) \wedge AtGrabbingDist(?x)$ (it knows it can achieve $FacingBar(?x)$ later.) If it achieves $OnMidline(?x)$ first, it will then try using the $\xrightarrow{\texttt{backward}} AtGrabbingDist(?x)$ TOP

[8]This comment assumes that the reinforcement learning system would be given a few sample training runs by a teacher; otherwise the exploration problem would probably preclude learning. Of course, it is unfair to do a direct comparison between unsupervised reinforcement learning and a system such as TRAIL that has access to a teacher.

to achieve $AtGrabbingDist(?x)$. This will fail immediately, as $OnMidline(?x)$ will become false. Since $OnMidline(?x)$ is only a delete-list element, TRAIL does not identify the fact that having $FacingBar(?x)$ true prevents it from being deleted. On the other hand, achieving $AtGrabbingDist(?x)$ first leaves the agent with no way to achieve $OnMidline(?x)$. Therefore neither way of achieving the goal appears to work.

So what is the solution? It would appear that what is really going on here is that there is one condition that needs to be achieved, namely $OnMidline(?x) \wedge AtGrabbingDist(?x)$. If there were a single predicate P corresponding to that conjunction, a new TOP $\xrightarrow{\texttt{backward}} P$ could be correctly learned, and would lead to a successful solution. However, it would be misleading to simply add this new predicate P to the Botworld representation. There is no a priori reason that a user would know to include P in the representation (clearly, we did not think to include P when we formalized the domain!) and it is unrealistic to expect that an expert will be around in all new domains to provide new predicates, especially if analysis is needed to determine which new predicates are necessary, as in this case.

Therefore, it seems that some sort of statistical analysis is needed to deal with this problem. If a conjunction of predicates is used very frequently, especially if it is a combination that seems to be difficult to achieve, we might want to have TRAIL create a new predicate expressing the conjunction. This could also form one of the components of a useful hierarchical learning system, as discussed further in Section 7.2. However, we clearly need to be careful in introducing such new concepts, in order to prevent the learning system from being overwhelmed with the newly created predicates.

6.6 Performance in the Delivery Domain

Now, we examine the learning behavior in the other Botworld-based domain, the office delivery domain, first introduced in Section 1.2.2. Since the robot is given the basic navigational routines, the domain is essentially a discrete one. Objects are held by one of a finite number of agents, each person is in one location, and the robot itself is either in a room or in transit between rooms. The operations in this task could

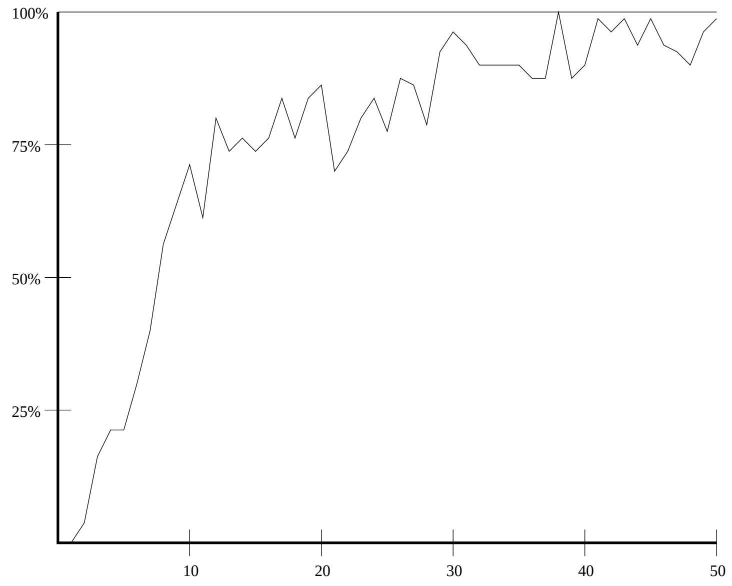

Figure 6.17: Mean Success Rate in the Delivery Domain as a Function of the Number of Tasks Completed

easily be learned by any action-model learner, such as LIVE or OBSERVER, as well as by TRAIL. However, note that the number of possible states, even in this simple domain, is actually very large. Ignoring for the moment the fact that the robot may have copies of objects, each of the 5 objects can be with each of 4 people or the robot, and there are at least 5 relevant rooms for the robot to be in. This alone accounts for 5^6 or 15625 possible states. The number of possible states and goals in this domain make the transfer of learning across tasks essential for learning. Given this need, the delivery domain appears to be completely out of reach of any existing reinforcement learning system.

The performance of TRAIL in the delivery domain, over 50 runs of 50 tasks each, is shown in Figure 6.17. In these experiments, there were four types of tasks that were given to the robot: delivering messages to people, fetching articles and delivering them to people, and delivering copies of articles on regular paper and on slides. (The last two tasks are distinguished only by the need for a different setting on the copy machine.) There are 64 possible goals in this domain, 424 if the number of copies to be made is considered as a distinctive feature.

Although this domain has many possible states, its structure is relatively straightforward, and as we said earlier, should be easily learnable by an action-model learner. Unlike the construction domain, there are no perceptual aliasing effects or unreliable actions to complicate learning. Therefore, it is somewhat surprising that TRAIL does not learn to complete the tasks more quickly than is shown in Figure 6.17. The reason for this slow learning is due to one of the heuristics used in TRAIL's learning mechanism. In inducing preimages, intervals are generalized only to the minimum extent necessary. Thus, if the copy action has been used to make 3 copies of one document and 6 copies of another document, TRAIL will assume that the action can only be used to make between 3 and 6 copies. Thus, if TRAIL is asked to make 8 copies of a document, it will not plan to use the copy action, and will need to call the teacher. Such teacher calls will continue to occur until TRAIL has seen copy tasks and slide-copy tasks involving both 1 copy and 10 copies (the maximum needed in any of the example problems.) This fact explains why even after seeing 50 example tasks, TRAIL has not yet converged to 100% performance on new tasks. The performance could be improved significantly if TRAIL generalized such intervals by default, but the selection of a good method for interval generalization is a subject for future research.

6.7 Learning Issues in the Flight Simulator

While TRAIL succeeds quite well at learning in Botworld and the delivery domain, the flight simulator domain is considerably more complicated. As we discussed in Section 6.3, it appears that some aspects of the flight simulator domain may be slightly beyond its current abilities. In particular, experimentation with the task of level flight (and a few experiments in navigation) revealed a number of issues which caused difficulty for TRAIL's learning system. Some of the most important of these issues are listed below.

- **The goal of level flight is really a maintenance goal.** A *maintenance goal* is one in which a condition, in this case a climb-rate near 0, must be maintained over some period of time. Since there is no way to represent maintenance goals within the current TRAIL formalism, the one-time goal $Altitude([125..175]) \wedge$

$ClimbRate([-1..1])$ is used. But this goal is achieved, at least for a short period of time, whenever the plane stops climbing, even if it has not reached a stable state. Therefore, TRAIL often sees action success cases in which actions such as `dec-throttle` cause the plane to level out.

- **A focus on immediate effects can cause TRAIL to learn the wrong operators.** As we saw earlier, when the teacher is achieving level flight, the sequence of actions it uses to level off is: decrease the throttle to 50%, lower the flaps to 0, and apply the `level` PID controller. This sequence allows for very stable flight, but the climb rate usually becomes 0 while the teacher is lowering the throttle or the flaps. Therefore TRAIL learns TOPs such as $\xrightarrow{\texttt{dec-throttle}} ClimbRate(?x)$ and $\xrightarrow{\texttt{dec-flaps}} ClimbRate(?x)$, rather than the more accurate $\xrightarrow{\texttt{level}} ClimbRate(?x)$. Clearly decreasing the throttle to zero is not a safe way to achieve stable level flight!

- **There are some situations in which it is difficult to correctly identify action failures.** For instance, consider the plan shown in Figure 6.18a. This plan was constructed by TRAIL in a situation where it was above the target altitude, and it in fact succeeds, since the `up` action causes the plane to begin leveling out once the condition $Altitude([125..175])$ becomes true. However, the true precondition for the TOP $\xrightarrow{\texttt{up}} ClimbRate(?x)$ must include the condition that the current climb rate is less than the target climb rate. TRAIL has a mechanism for including such conditions (see Section 5.4.3), but it only works if TRAIL can collect appropriate action failures.

  Consider the plan shown in Figure 6.18b. Clearly, it will fail once the altitude becomes higher than 175. In particular, an activation failure will be detected for the node $Altitude[125..175]$ since its activating condition is no longer true. Thus $\neg Altitude(?x)$ will be included as a side effect of the $\xrightarrow{\texttt{up}} ClimbRate(?x)$ TOP, and no negative instance will be generated to allow TRAIL to induce the correct preimage. The failure is in fact a failure of case 6 from Table 4.2 rather than case 5, and should thus generate a negative instance rather than a delete-list element. The true cause of the failure could be determined by

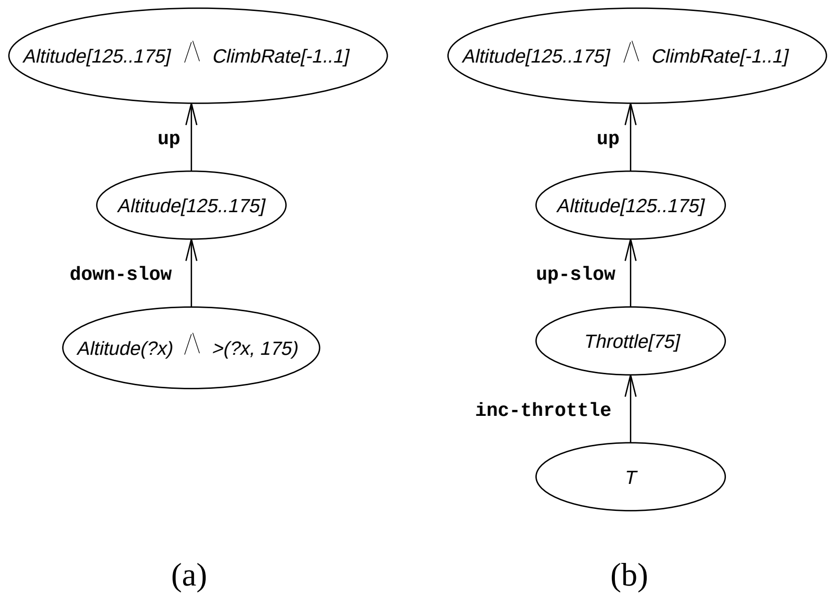
Altitude[125..175] /\ ClimbRate[-1..1]
up
Altitude[125..175]
down-slow
Altitude(?x) /\ >(?x, 175)
(a)
Altitude[125..175] /\ ClimbRate[-1..1]
up
Altitude[125..175]
up-slow
Throttle[75]
inc-throttle
T
(b)

Figure 6.18: Two Plans For Level Flight

repeated experimentation, but experimentation in the flight simulator has its own difficulties, as discussed below.

- **Experimentation in the domain is usually not very effective.** In some domains, such as the delivery domain or the part-machining domain used for experimentation in EXPO (Gil 1992), experiments can be carried out without drastically affecting the state of the system. However, in a real-time domain such as the flight simulator, the experiment itself can often have undesirable effects. For instance, experiments using the `up` action often take the plane so high that once the experiment is over, any `down` TOP suffers from a timeout failure before it can return the plane to the desired altitude. Or worse, if an experiment is done using an action such as `wait`, the plane may well crash before the experiment is completed. Finally, the results of the experiment itself may not be very consistent. Consider the failure of the TOP $\xrightarrow{\text{up}} ClimbRate(?x)$ mentioned above. If the agent experiments by continuing the `up` operator, the controller is sufficiently unreliable that the plane will occasionally level out during the climb. Thus, TRAIL will believe that the TOP usually does work and that $\neg Altitude(?x)$ is an appropriate side effect of the TOP. Clearly, some better method of experimentation is needed to deal with such domains.

- **TRAIL does not have any mechanism for handling conditional side effects.** Due to the dynamics of the simulator, the TOP $\xrightarrow{\texttt{level}} ClimbRate(?x)$ works much more reliably with the flaps down than with the flaps up. Therefore, if the flaps are up, the TOP will often result in an activation failure because the condition $Altitude([125..175])$ will become false before the climb rate becomes 0. However, this activation failure is due to a side effect rather than a failure of the TOP itself, and thus does not produce a negative instance. TRAIL has no mechanism for learning the fact that the side effect $\neg Altitude(?x)$ only occurs if $FlapsUp$ is true. Therefore, TRAIL instead simply learns the fact that $Altitude(?x)$ is sometimes a delete-list element of the TOP. There is no mechanism for it to realize that the flaps should be lowered before applying the `level` action.

- **It is difficult for TRAIL to coordinate multiple variables.** Consider the task of landing at a specific point on a runway. The pilot must make sure that the rate of descent is such that not only will the plane touch down safely, it will also reach altitude 0 at the appropriate xy location. Such goals require either a mechanism for explicitly reasoning about time or a new set of domain predicates designed specifically for such navigation tasks.

Chapter 7

Summary and Conclusions

In this thesis, we have presented a learning system for an autonomous agent. This thesis is intended to help extend the applicability of autonomous learning systems to domains that are dynamic, unpredictable, and continuous. A number of researchers have constructed autonomous learning systems, but few of these systems have any capability to operate in such domains. While the techniques introduced in this thesis are far from a complete solution to the problems of learning in such domains, we believe that the thesis is a significant step forward in that direction.

7.1 Summary of TRAIL

TRAIL is an architecture for an autonomous agent, with a primary emphasis on learning. The agent itself is based on the execution of symbolic, reactive control structures known as teleo-reactive trees. TR trees allow the agent to execute durative actions over indefinite time periods, while reacting appropriately to changes in the environment that might occur during execution. In addition, the symbolic nature of TR trees supports their construction through human coding, automated planning, and experience-based learning.

Learning in TRAIL is based on the assumption that the easiest method of learning for an autonomous agent is to build a partial world model that can be used to plan the agent's actions. This approach falls somewhere between the extremes of either building a complete world model or learning a complete behavioral policy in the form of a state-action mapping, both of which appear to be computationally difficult. In particular, TRAIL learns action models, known as teleo-operators, that can be used by an automated planner to construct teleo-reactive trees.

TRAIL learns these teleo-operators from two sources: the execution of TR trees that have been constructed by the planner, and the actions of an external teacher that can be called upon to complete goals when the planner is unable to construct a plan. Once TRAIL has built up an approximate set of action models, its behavior forms a cycle in which the planner generates TR trees based on the current teleo-operators, the execution mechanism executes the TR trees, and the learner observes the success or failure of the trees, and produces an updated set of teleo-operators. This cycle continues as goals are given to the agent, allowing TRAIL to successively approximate a correct set of teleo-operators.

The learning itself occurs in two separate phases. First, TRAIL's learner observes the actions of the external teacher or the results of the execution of a TR tree, and produces a set of state descriptions, each labeled as a positive or negative instance. In brief, the execution of a TR node can be classified as either a success, a timeout failure (in which the action apparently had no significant effects), or an activation failure (in which the action had an undesirable effect). Experimentation can also be used to assist in analyzing the causes of a node failure. Finally, in the second phase of learning, the positive and negative instances are given as input to a concept learning algorithm. This algorithm uses Inductive Logic Programming techniques to produce a symbolic precondition for the teleo-operator in question.

TRAIL has been successfully tested in three domains: a simulated construction domain, a simulated office delivery domain, and the Silicon Graphics, Inc. flight simulator. Experimental results show that TRAIL learns successfully in the construction and office domains, although not perfectly in the construction domain. TRAIL has also successfully learned to take off and land in the flight simulator, but the interrelated variables present in the flight domain have made it difficult for TRAIL to obtain satisfying performance on more complex aerial tasks. However, work with the flight simulator domain has led to the development of a number of ideas that are important to the TRAIL architecture.

7.2 Areas for Future Work

There are a wide variety of interesting research issues that are raised by the TRAIL work. The following lists a number of the areas for future work that follow naturally from the work described in this thesis. Some of them address specific limitations of the TRAIL system, while others are broader issues that were beyond the scope of the thesis.

- **Learning of Hierarchies** Any autonomous agent that hopes to be able to deal with complex real-world domains will have to be able to reason about actions at multiple levels. Plans must be constructed first at a high level, and then made more detailed through lower-level reasoning. Although the use of durative actions and teleo-operators can make this lower-level reasoning less computationally intensive in certain cases, most real-world problems will simply be too difficult to solve by reasoning only about base-level actions. Therefore, both our planning and our learning systems must be able to reason with hierarchies of actions. The teleo-reactive mechanism naturally allows the construction of hierarchies of trees (see Nilsson (1994) for more details), but TRAIL's learning mechanism does not currently have any support for hierarchical learning.

  One way of approaching the problem of learning higher-level actions is through the observation that the act of executing a TR tree is itself a durative action. Suppose we have constructed a tree that achieves $Holding(?x)$ where $?x$ is a bar in the Botworld construction domain. The act of executing this tree has a precondition (at least one node in the tree must be active), a durative action (continually find the highest true node in the tree and execute the corresponding action), and an effect. Therefore, it can naturally be described by a teleo-operator. This TOP would contain information on when the tree could be applied, what the effect of it would be, and what any side effects might be – exactly the sort of information that a hierarchical planner would need to know in order to use the tree as a single operator in a high-level plan.

A variety of work has been done on the subject of hierarchy learning for autonomous agents. Drescher's schema mechanism (Drescher 1991) develops complex actions for a *tabula rasa* autonomous agent, based on Piagetian learning theory. Ring (1991) combines actions to create operators using reinforcement learning. Triangle tables (Fikes, Hart & Nilsson 1972, Nilsson 1985) are an early method of macro-operator creation, while more recent work has focused on more flexible macro-operators involving iteration and recursion (Shell & Carbonell 1989, Shavlik 1990). However, it has been observed that macro-operators can often increase rather than decrease the complexity of search by increasing the branching factor (Minton 1989).

- **Predicate Invention** As we noted several times within this thesis, TRAIL is very dependent on the initial set of predicates that are given to it. Every concept that it learns, and every tree that it constructs, must be in terms of these predicates. Thus, if there is a hidden variable that is needed for learning, or if the agent wishes to use higher-level predicates for hierarchical planning, it must be able to make use of some form of predicate invention. Predicate invention is already used in a number of ILP systems (Stahl 1993) although most of these methods are not directly applicable to TRAIL.

  One obvious source of invented predicates lies in the learned preimages of TOPs. If an action appears to be unreliable, its preimage may be missing some hidden variable that is not described by the existing set of predicates. A new predicate can be hypothesized that holds only in those states in which the action is successful. (This idea is the basis for new feature construction within LIVE (Shen 1994).) However, unless the new feature can easily be explained in terms of past actions or world states, it can be difficult to determine whether the new feature holds in a given state, and thus difficult to use the new feature when planning or acting.

  As was mentioned above, predicate invention based on past actions of the agent is an important part of the action-model learning system LIVE. Drescher (1991) includes a mechanism for learning new predicates as a component of his schema mechanism for learning hierarchies of actions.

Predicate invention is also closely related to the learning of hierarchies, as higher-level predicates as well as higher-level actions will probably be needed. Again, preimages are an obvious candidate for the construction of new features. For instance, a predicate with the meaning "Holding(?x)-operator-can-be-applied" is likely to be of use in building higher-level plans that might make use of our hierarchical $Holding(?x)$ operator. This is similar to the idea of predicate relaxation used in the hierarchical planner PABLO (Christensen 1990). Predicate relaxation, in brief, is a systematic method for weakening a condition so that it holds in states in which it can easily be made true as well as states in which it is already true.

Finally, new features may be constructed from conjunctions of predicates that frequently appear together in plans. FOCL (Silverstein & Pazzani 1991) is a version of FOIL that attempts to solve the literal connectedness problem by inventing new predicates that are combinations of existing predicates. Conjunctions of predicates may have another important use in TRAIL as well, as TRAIL learns TOPs only for single postconditions. If it should need to have a TOP that achieves multiple conditions simultaneously, it cannot learn such a TOP using the current learning algorithm. (Such a situation was described in Section 6.5.) However, like all predicate invention methods, the construction of conjunctive features carries the risk of overwhelming the learner with newly created predicates.

- **Experimentation and Exploration** TRAIL at present uses only a simple form of experimentation, designed to determine whether an activation failure is due to an overly general or overly specific preimage (for more details on TRAIL's experimentation, see Section 4.4.2.) There are several other possible uses for experimentation in a learning system, however. Any time an action fails to have an expected effect, the agent might wish to experiment to determine whether the failure was a random execution failure or truly a case in which the action will not work. Since experimentation is computationally expensive, the agent may not wish to experiment upon every action failure, but rather only on those that directly contradict an earlier action success. Experimentation in this case would allow the agent to determine a better estimate for the actual probability of success.

Another form of experimentation would be useful in order to reduce dependence on the teacher in the case where the agent is unable to construct a plan for some goal. In this case, the agent may try actions in states that are somewhat similar to states in which the action has been observed to work.[1] If the action should succeed, a new positive instance has been observed and the TOP preimage can be generalized without having to resort to the teacher. This is only one approach to the more general problem of intelligent exploration of unknown environments.

Both LIVE (Shen 1994) and EXPO (Gil 1992) include significantly more experimentation than does TRAIL. LIVE does experimentation by applying operators that it believes to be faulty, using a new (essentially arbitrary) assignment of variables to objects, in hopes of discovering unexpected behavior. EXPO identifies missing preconditions for operators by hypothesizing a list of possibilities and then generating experiments in which some of the hypothesized preconditions apply. Both of these methods could potentially be added to the TRAIL architecture, and might well reduce its dependence on the teacher. A variety of other less related research work has also included methods of experimentation, including Mitchell's LEX system (Mitchell, Utgoff & Banerji 1983) and Christiansen's work on learning manipulation strategies for robotic graspers (Christiansen, Mason & Mitchell 1990, Christiansen 1989).

- **Learning from Delayed Effects** One of the main difficulties that plagued TRAIL in the flight simulator domain was an inability to reason about the delayed effects of actions. Recall that TRAIL assumes that effects are due either to whatever action TRAIL was taking at the time or to some unpredictable outside

[1]This would be particularly appropriate for interval preimages, as discussed in Section 6.6.

agency. Thus, TRAIL is unable to accurately represent actions that may have effects that persist after completion of the action. For instance, consider the `up` action in the flight simulator. After the action is complete, the aircraft will continue to climb for some time, even if the `level` action is applied. This effect could conceivably be modeled as a conditional effect of the `level` action that depends on the initial climb rate, but this is not really an accurate model.

What is needed in this case is probably a fundamental extension of TRAIL's action representation. While teleo-operators are better suited to representing processes than are traditional STRIPS operators, it appears that they are not sufficient to represent actions such as the flight simulator `up` action. An agent that can learn successfully in the flight simulator domain most likely will need to be able to reason explicitly about continuous processes. DeJong (1994) has done some excellent preliminary work in this direction. However, once actions are allowed to have delayed effects, the agent has a temporal credit assignment problem in deciding which action was responsible for a particular effect. Temporal credit assignment has been examined in much detail in the reinforcement learning community (Watkins 1989) but it is not clear how to extend most of this work beyond assigning credit for a single numerical reward or punishment.

- **Real-time Performance** There are many domains in which an autonomous agent does not need to worry very much about time spent learning and planning. Tasks such as office delivery, construction, and housework do not generally require split-second response, so it is acceptable if the agent pauses for several seconds in order to learn or plan. (Of course, much of the learning might be saved for "down time" when the robot is not needed for some more urgent task.) However, in order to be able to realistically handle domains in which real-time response is necessary, such as the flight simulator, the agent will need to be able to do its learning and planning in parallel with its other behaviors. For instance, the operation of the low-level PID controllers for our aircraft is independent of TRAIL's higher-level behaviors, and even if TRAIL is temporarily left without a high-level goal (due to a plan failure, for instance), it is fairly easy to program in a set of low-level behaviors that keep the aircraft stable, even if it is not achieving any navigational goals.[2] Any autonomous agent in a real-time domain will presumably need such parallelism, although the current state may change significantly while planning is occurring, resulting in a significantly more difficult time-dependent planning problem (Dean & Boddy 1988).

The current TRAIL system, unfortunately, does not include such parallelism. The implementation of a parallel processing system for planning and acting in LISP is a considerable project, and not really related to the main aim of this thesis. Instead, our flight simulator test runs all simply pause the simulator at appropriate times, simulating an instantaneous learning and planning computation. (The planning was in fact nearly instantaneous, as the plans constructed were all relatively small. The learning, however, was not nearly so quick.)

- **Improvements to the Learning Algorithm** There are at least three obvious improvements we could make to TRAIL's learning algorithm. First, the noise-handling mechanism of the algorithm could easily be improved. This issue was examined in more detail in Section 5.5.4.

  Second, the current algorithm is non-incremental, so the preimage must be completely recomputed every time a new instance is generated. There is little existing work on incremental ILP algorithms, although the HILLARY (Iba et al. 1988) incremental learning algorithm has been applied to relational as well as attribute-value problems. A few early ILP systems such as MIS (Shapiro 1983) were incremental but relied on an oracle to answer membership queries. Some of the more recent work on theory refinement has been in the direction of incremental algorithms (Mooney 1992).

  Finally, real world domains often have many more predicates than the domains described in this thesis. There is considerable work in the concept learning community on the elimination of irrelevant features (John, Kohavi & Pfleger

[2]This is not true, of course, in a less benign domain such as the dogfight environment. However, in such an environment, we can expect somewhat worse performance until learning has reached a reasonable level.

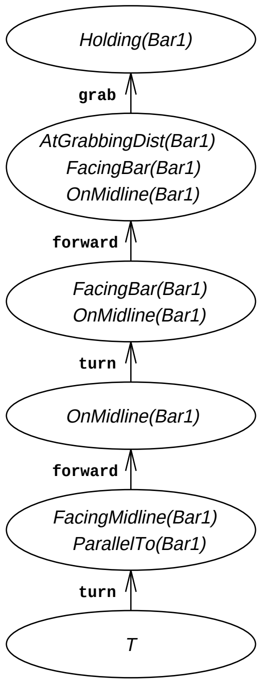

Figure 7.1: A Standard TR Tree For Bar-Grabbing

1994, Kohavi & John 1996). However, the problem for TRAIL is somewhat broader, as irrelevant predicates are a problem both in the induction of preimages and in the selection of TOPs to be learned. If there are many predicates that change in the environmental descriptions, TRAIL will construct at least one TOP for each of them, potentially leading to an overwhelming number of learned TOPs.

- **Fuzzy TR Trees** One weakness of the current TR tree formalism is that it conceptualizes the world as being entirely in a single state. For instance, consider a typical bar-grabbing tree as shown in Figure 7.1. As the bot approaches the bar midline, the $OnMidline(Bar1)$ node is false. At some point, the bot reaches the midline, and it suddenly becomes true. Therefore, the bot's behavior is to execute only the `forward` action until it reaches the midline, then execute only the `turn` action until it is facing the bar.

Instead, we could think of the actions in the TR tree as being determined by fuzzy predicates with varying degrees of truth. In a fuzzy TR tree (Shoykhet 1996), as the bot approaches the midline, the $OnMidline$ node gradually becomes true. Thus, the bot would initially be moving directly forward, but as the $OnMidline$ node became more and more true, the `turn` action would gradually become more activated, causing the bot to begin to turn as well as move. The strength of the `turn` action would continue to increase as the $OnMidline$ node became true, then would eventually decline again as the next higher node, $FacingBar$, began to become true.

7.3 Lessons Learned and Conclusions

This thesis presented TRAIL, an architecture for an autonomous agent that is capable of displaying goal-directed reactive behavior, creating plans for tasks through automated planning and replanning, and learning models of actions that can be used for planning. We have shown that this architecture is capable of learning successfully in simple simulated domains containing continuous variables, durative actions, structured state descriptions, and unpredictable action effects.

The development of the TRAIL architecture has also included a number of other advances, including the development of a new model of actions that is appropriate for reactive agents in continuous domains, an analysis of the possible causes of action failure for durative actions, a demonstration of the use of a DINUS-like Inductive Logic Programming algorithm for action-model learning, and the development of a metric suitable for evaluating the learning behavior of autonomous agents in new domains.

Section 7.2 highlighted some of the areas of future work that are needed to extend the ideas of the TRAIL architecture to more complicated and realistic domains, including the learning of hierarchies of predicates and actions, new methods of experimentation and exploration, and strategies for dealing with delayed and interacting effects. We are hopeful that future research in these directions will eventually point the way towards the development of a general-purpose learning architecture, with which autonomous agents will ultimately be able to learn in the real world.

Bibliography

Agre, P. & Chapman, D. (1987), PENGI: An implementation of a theory of activity, *in* "AAAI-87: Proceedings of the Sixth National Conference on Artificial Intelligence", AAAI Press / The MIT Press, pp. 268–272.

Angluin, D. (1987), "Learning regular sets from queries and counterexamples", *Information and Computation* **75**(2), 87 – 106.

Angluin, D. (1988), "Queries and concept learning", *Machine Learning* **2**(4), 319 – 342.

Bates, J. (1992), "Edge of intention", Presented at AAAI Arts Exhibition 1992, SigGraph 1993, Ars Electronica 1993, and the Boston Computer Museum.

Bates, J. (1994), "The role of emotion in believable agents", *Communications of the ACM* **37**(7), 122–125.

Baum, L., Petrie, T., Soules, G. & Weiss, N. (1970), "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains", *Annals of Mathematical Statistics* **41**, 164 – 171.

Bellman, R. E. (1962), *Dynamic Programming*, Princeton, NJ: Princeton University Press.

Benson, S. & Nilsson, N. (1995), Reacting, planning, and learning in an autonomous agent, *in* K. Furukawa, D. Michie & S. Muggleton, eds, "Machine Intelligence 14", Oxford: the Clarendon Press.

Bollinger, J. G. & Duffie, N. A. (1988), *Computer Control of Machines and Processes*, Reading, Massachusets: Addison-Wesley.

Bratko, I. & Muggleton, S. (1995), "Applications of inductive logic programming", *Communications of the ACM* **38**(11), 65 – 70.

Bratko, I., Urbančič, T. & Sammut, C. (1995), Behavioural cloning: Phenomena, results, and problems, Unpublished.

Brooks, R. (1986), "A robust layered control system for a mobile robot", *IEEE Journal of Robotics and Automation* **RA-2**(1), 14–23.

Brooks, R. A. (1989*a*), Engineering approach to building complete, intelligent beings, *in* "Proceedings of the SPIE – The International Society for Optical Engineering", Vol. 1002, pp. 618 – 625.

Brooks, R. A. (1989*b*), "A robot that walks; emergent behaviors from a carefully evolved network", *Neural Computation* **1**(2), 253 – 262.

Cameron-Jones, R. & Quinlan, J. R. (1993), Avoiding pitfalls when learning recursive theories, *in* R. Bajcsy, ed., "Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence", Morgan Kaufmann.

Cestnik, B. (1990), Estimating probabilities: A crucial task in machine learning, *in* "Proceedings of the Ninth European Conference on Machine Learning", pp. 147 – 149.

Christensen, J. (1990), A hierarchical planner that generates its own hierarchies, *in* "AAAI-90: Proceedings of the Eighth National Conference on Artificial Intelligence", AAAI Press / The MIT Press, pp. 1004 – 1009.

Christiansen, A. D. (1989), Automated acquisition of task theories for robotic manipulation, PhD thesis, Carnegie Mellon University.

Christiansen, A. D. (1991), Manipulation planning from empirical backprojection, *in* "Proceedings. 1991 IEEE International Conference on Robotics and Automation", pp. 762–768.

Christiansen, A. D., Mason, M. T. & Mitchell, T. M. (1990), Learning reliable manipulation strategies without initial physical models, *in* "Proceedings of the 1990 IEEE International Conference on Robotics and Automation", Vol. 2, IEEE Computing Society Press, pp. 1224–30.

Clark, P. & Niblett, T. (1989), "The CN2 induction algorithm", *Machine Learning* **3**(4), 261 – 283.

Connell, J. (1992), SSS: A hybrid architecture applied to robot navigation, *in* "IEEE Conference on Robotics and Automation", pp. 2719 – 2724.

Dean, T. & Boddy, M. (1988), An analysis of time-dependent planning, *in* "AAAI-88: Proceedings of the Seventh National Conference on Artificial Intelligence", American Association for Artificial Intelligence, AAAI Press / The MIT Press, pp. 49 – 54.

DeJong, G. F. (1994), "Learning to plan in continuous domains", *Artificial Intelligence* **65**, 71 – 141.

Dietterich, T. G. (1990), "Machine learning", *Annual Review of Computer Science* **4**, 255 – 306.

Dolvsak, B., Bratko, I. & Jezernik, A. (1994), Finite-element mesh design: An engineering domain for ILP application, *in* "Proceedings of the Fourth International Workshop on Inductive Logic Programming ILP-94", Bad Honnef/Bonn.

Drescher, G. (1991), *Made Up Minds: A Constructivist Approach to Artificial Intelligence*, MIT Press.

Džeroski, S. (1995), Learning first-order clausal theories in the presence of noise, *in* "Proceedings of the Fifth Scandinavian Conference on Artificial Intelligence", pp. 51 – 60.

Džeroski, S., Muggleton, S. & Russell, S. (1992), PAC learnability of determinate logic programs, *in* "Proceedings of the Fifth ACM Workshop on Computational Learning Theory", pp. 128 – 135.

Džeroski, S., Muggleton, S. & Russell, S. (1993), Learnability of constrained logic programs, *in* "Proceedings of the European Conference on Machine Learning", pp. 342 – 347.

Džeroski, S., Todorovski, L. & Urbančič, T. (1995), Handling real numbers in ILP: a step towards better behavioral clones, *in* "Proceedings of the Eighth European Conference on Machine Learning", pp. 283 – 286.

Etzioni, O. & Weld, D. (1994), "A softbot-based interface to the internet", *Communications of the ACM* **37**(7), 72 – 76.

Fikes, R. E. & Nilsson, N. J. (1971), "STRIPS: A new approach to the application of theorem proving to problem solving", *Artificial Intelligence* **2**, 189–208.

Fikes, R. E., Hart, P. E. & Nilsson, N. J. (1972), "Learning and executing generalized robot programs", *Artificial Intelligence* **4**, 251 – 288.

Flynn, A. M. & Brooks, R. A. (1988), MIT mobile robots – what's next?, *in* "IEEE International Conference on Robotics and Automation", pp. 611 – 617.

Frazier, M. & Page, C. (1993), Learnability in inductive logic programming: some basic results and techniques, *in* "AAAI-93: Proceedings of the Eleventh National Conference on Artificial Intelligence", American Association for Artificial Intelligence, AAAI Press / The MIT Press, pp. 93 – 98.

Galles, D. (1993), Map building and following using teleo-reactive trees, *in* "Intelligent Autonomous Systems: IAS3", Washington: IOS Press, pp. 390 – 398.

Gil, Y. (1992), Acquiring Domain Knowledge for Planning by Experimentation, PhD thesis, Carnegie Mellon University.

Hanks, S., Pollack, M. & Cohen, P. (1993), "Benchmarks, test beds, controlled experimentation, and the design of agent architectures", *AI Magazine* **14**(4), 17–42.

Hayes-Roth, B. (1995), Agents on stage: Advancing the state of the art of AI, *in* C. S. Mellish, ed., "Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence", Morgan Kaufmann, San Mateo, CA, pp. 967 – 971.

Hertz, J., Krough, A. & Palmer, R. G. (1991), *Introduction to the Theory of Neural Computation*, Santa Fe Institute Studies in the Sciences of Complexity, Reading, Massachusets: Addison-Wesley.

Iba, W., Wogulis, J. & Langley, P. (1988), Trading off simplicity and coverage in incremental concept learning, *in* J. Laird, ed., "Proceedings of the Fifth International Conference on Machine Learning", Morgan Kaufmann, pp. 73 – 79.

Jennings, N. & Wooldridge, M. (1996), "Software agents", *IEE Review* **42**(1), 17 – 20.

John, G. H., Kohavi, R. & Pfleger, K. (1994), Irrelevant features and the subset selection problem, *in* H. Hirsh & W. Cohen, eds, "Machine Learning: Proceedings of the Eleventh International Conference", Morgan Kaufmann, pp. 121–129.

Kaelbling, L. P. & Rosenschein, S. (1990), "Action and planning in embedded agents", *Robotics and Autonomous Systems* **6**, 35 – 48.

Kazakov, D., Popelinsky, L. & Stepankova, O. (1996), "Review of available ILP datasets", Available at `http://www.gmd.de/ml-archive/datasets/ilp-res.html`.

King, R., Sternberg, J. & Srinivasan, A. (1995), "Relating chemical activity to structure: an examination of ILP successes", *New Generation Computing* **13**(3-4), 411 – 433.

Kohavi, R. & John, G. H. (1996), "Wrappers for feature subset selection", *Artificial Intelligence*. To Appear.

Korf, R. E. (1985), "Depth first iterative deepening: An optimal admissible tree search", *Artificial Intelligence* **27**(1), 97–109.

Korf, R. E. (1988), Search: A survey of recent results, *in* H. Shrobe, ed., "Exploring Artificial Intelligence", San Mateo, California: Morgan Kaufmann.

Kuipers, B. & Byun, Y.-T. (1991), "A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations", *Robotics and Autonomous Systems* **8**(1-2), 47 – 63.

Langley, P. (1996), *Elements of Machine Laerning*, San Mateo, California: Morgan Kaufmann.

Lavrač, N. & Džeroski, S. (1994), *Inductive Logic Programming: Techniques and Applications*, Chichester, England: Ellis Horwood.

Lavrač, N., Džeroski, S. & Grobelnik, M. (1991), Learning nonrecursive definitions of relations with LINUS, *in* "Proceedings of the Fifth European Working Session on Learning", pp. 265 – 281.

Littman, M., Cassandra, A. & Kaelbling, L. P. (1995), Learning policies for partially observable environments: Scaling up, *in* A. Prieditis & S. Russell, eds, "Machine Learning: Proceedings of the Twelfth International Conference", Morgan Kaufmann, pp. 362 – 369.

Lozano-Pérez, T., Mason, M. T. & Taylor, R. (1984), "Automatic synthesis of fine-motion strategies for robots", *International Journal of Robotics Research* **3**(1), 3 – 24.

Mahadevan, S. (1992), Enhancing transfer in reinforcement learning by building stochastic models of robot actions, *in* D. Sleeman & P. Edwards, eds, "Machine Learning: Proceedings of the Ninth International Workshop", Morgan Kaufmann, pp. 290 – 299.

Mahadevan, S. & Connell, J. (1992), "Automatic programming of behavior-based robots using reinforcement learning", *Artificial Intelligence* **55**(2-3), 311–365.

McCarthy, J. & Hayes, P. (1970), Some philosophical problems from the standpoint of artificial intelligence, *in* B. Meltzer & D. Michie, eds, "Machine Intelligence 4", Edinburgh: Edinburgh University Press, pp. 463–502.

Michalski, R. (1983), A theory and methodology of inductive learning, *in* "Machine Learning, An Artificial Intelligence Approach, Volume I", Palo Alto, California: Tioga Press.

Minton, S. (1989), Selectively generalizing plans for problem solving, *in* "Proceedings of the Ninth International Joint Conference on Artificial Intelligence", Morgan Kaufmann, pp. 596 – 599.

Mitchell, T. M. (1982), "Generalization as search", *Artificial Intelligence* **18**, 203–266.

Mitchell, T. M., Keller, R. & Kedar-Cabelli, S. (1986), "Explanation-based generalization: A unifying view", *Machine Learning* **1**(1), 47 – 80.

Mitchell, T., Utgoff, P. & Banerji, R. (1983), Learning by experimentation: acquiring and refining problem-solving heuristics, *in* "Machine Learning, An Artificial Intelligence Approach, Volume I", Palo Alto, California: Tioga Press.

Mooney, R. J. (1992), Batch versus incremental theory refinement, *in* "Proceedings of AAAI Spring Symposium on Knowledge Acquisition".

Moore, A. W. (1990), Acquisition of dynamic control knowledge for a robotic manipulator, *in* B. Porter & R. Mooney, eds, "Proceedings of the Seventh International Conference on Machine Learning", Morgan Kaufmann, pp. 244 – 252.

Moore, A. W. & Atkinson, C. G. (1993), "Prioritized sweeping: Reinforcement learning with less data and less real time", *Machine Learning* **13**(1), 103 – 130.

Moravec, H. (1988), "Certainty grids for mobile robots", *AI Magazine* **9**(2), 61 – 74.

Muggleton, S. & Feng, C. (1990), Efficient induction of logic programs, *in* "Proceedings of the First Conference on Algorithm Learning Theory", pp. 368 – 381.

Muggleton, S., ed. (1992), *Inductive Logic Programming*, San Diego, California: Academic Press.

Muggleton, S., King, R. & Sternberg, M. (1992), Protein secondary structure prediction using logic, *in* "Proceedings of the Second International Workshop on Inductive Logic Programming".

Murphy, P. M. & Aha, D. W. (1994), "UCI repository of machine learning databases", Available by anonymous ftp to `ics.uci.edu` in the `pub/machine-learning-databases` directory.

Nilsson, N. J. (1980), *Principles of Artificial Intelligence*, San Mateo, California: Morgan Kaufmann.

Nilsson, N. J. (1984), Shakey the robot, Technical Report 323, SRI International, Menlo Park, California.

Nilsson, N. J. (1985), Triangle tables: A proposal for a robot programming language, Technical Report 347, SRI International, Menlo Park, California.

Nilsson, N. J. (1992), Towards agent programs with circuit semantics, Technical Report STAN-CS-92-1412, Stanford University Computer Science Department, Stanford, California.

Nilsson, N. J. (1994), "Teleo-reactive programs for agent control", *Journal of Artificial Intelligence Research* **1**, 139 – 158.

Peng, J. & Williams, R. J. (1993), "Efficient learning and planning within the Dyna framework", *Adaptive Behavior* **1**(4), 437 – 454.

Plotkin, G. (1969), A note on inductive generalization, *in* B. Meltzer & D. Michie, eds, "Machine Intelligence 5", Edinburgh: Edinburgh University Press, pp. 153 – 163.

Pollack, M. E. & Ringuette, M. (1990), Introducing the Tileworld: Experimentally evaluating agent architectures, *in* "AAAI-90: Proceedings of the Eighth National Conference on Artificial Intelligence", AAAI Press / The MIT Press, pp. 183 – 189.

Pomerleau, D. (1991), "Efficient training of artificial neural networks for autonomous navigation", *Neural Computation* **3**(1), 88 – 97.

Pomerleau, D. (1993), *Neural Network Perception for Mobile Robot Guidance*, Boston: Kluwer Academic Publishers.

Quinlan, J. R. (1986), "Induction of decision trees", *Machine Learning* **1**, 81–106.

Quinlan, J. R. (1990), "Learning logical definitions from relations", *Machine Learning* **5**(3), 239 – 266.

Quinlan, J. R. (1991), "Knowledge acquisition from structured data - using determinate literals to assist search", *IEEE Expert* **6**(6), 32 – 37.

Quinlan, J. R. (1992), *C4.5: Programs for Machine Learning*, San Mateo, California: Morgan Kaufmann.

Rabiner, L. (1990), A tutorial on hidden Markov models and selected applications in speech recognition, *in* A. Waibel & K.-F. Lee, eds, "Readings in Speech Recognition", San Mateo, California: Morgan Kaufmann.

Rabiner, L. R. & Juang, B. H. (1986), "An introduction to hidden Markov models", *IEEE Acoustics, Speech, and Signal Processing Magazine* pp. 4 – 16. January, 1986.

Ring, M. (1991), Incremental development of complex behaviors through automatic construction of sensory-motor hierarchies, *in* L. A. Birnbaum & G. C. Collins, eds, "Machine Learning: Proceedings of the Eighth International Workshop", Morgan Kaufmann, pp. 343 – 347.

Rivest, R. & Schapire, R. (1993), "Inference of finite automata using homing sequences", *Information and Computation* **103**(2), 299 – 347.

Rogers, S. O. & Laird, J. E. (1996), Symbolic performance and learning in complex environments, *in* "AAAI-96: Proceedings of the Thirteenth National Conference on Artificial Intelligence", American Association for Artificial Intelligence, AAAI Press / The MIT Press, p. 1405. abstract; longer version available.

Russell, S. & Norvig, P. (1995), *Artificial Intelligence: A Modern Approach*, Prentice Hall.

Sablon, G. (1994), Personal Communication.

Sablon, G. & Bruynooghe, M. (1994), Using the event calculus to integrate planning and learning in an intelligent autonomous agent, *in* C. Bäckström & E. Sandewall, eds, "Current Trends in AI Planning", IOS Press, pp. 254 – 265.

Sammut, C., Hurst, S., Kedzier, D. & Michie, D. (1992), Learning to fly, *in* D. Sleeman & P. Edwards, eds, "Machine Learning: Proceedings of the Ninth International Workshop", Morgan Kaufmann, pp. 385 – 393.

Schoppers, M. J. (1987), Universal plans for reactive robots in unpredictable environments, *in* "AAAI-87: Proceedings of the Sixth National Conference on Artificial Intelligence", AAAI Press / The MIT Press, pp. 1039–1046.

Schoppers, M. J. & Shu, R. (1990), An implementation of indexical-functional reference for embedded execution of symbolic plans, *in* "DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control".

Shapiro, E. (1983), *Algorithmic Program Debugging*, Cambridge, MA: MIT Press.

Shavlik, J. W. (1990), "Acquiring recursive and iterative concepts with explanation-based learning", *Machine Learning* **5**(1), 39 – 70.

Shell, P. & Carbonell, J. (1989), Towards a general framework for composing disjunctive and iterative macro-operators, *in* N. Sridharan, ed., "Proceedings of the Eleventh International Joint Conference on Artificial Intelligence", Morgan Kaufmann, pp. 596 – 602.

Shen, W.-M. (1989), Learning from the Environment Based on Actions and Percepts, PhD thesis, Carnegie Mellon University.

Shen, W.-M. (1990), Complimentary discrimination learning: A duality between generalization and discrimination, *in* "Proceedings of National Conference on Artificial Intelligence", MIT Press, pp. 834 – 839.

Shen, W.-M. (1994), *Autonomous Learning from the Environment*, Computer Science Press, W.H. Freeman and Company.

Shoham, Y. & Goyal, N. (1988), Temporal reasoning, *in* H. Shrobe, ed., "Exploring Artificial Intelligence", San Mateo, California: Morgan Kaufmann.

Shoykhet, A. (1996), Fuzzy t-r trees, Stanford University undergraduate project report.

Silverstein, G. & Pazzani, M. J. (1991), Relational cliches: Constraining constructive induction during relational learning, *in* L. A. Birnbaum & G. C. Collins, eds, "Machine Learning: Proceedings of the Eighth International Workshop", Morgan Kaufmann, pp. 203 – 207.

Stahl, I. (1993), Predicate invention in ILP - an overview, *in* "Machine Learning: EMCL-93. European Conference on Machine Learning Proceedings", pp. 313 – 322.

Tate, A., Hendler, J. & Drummond, M. (1990), A review of AI planning techniques, *in* J. Allen, J. Hendler & A. Tate, eds, "Readings in Planning", San Mateo, California: Morgan Kaufmann.

Teo, P. (1992), Botworld, unpublished manual.

Urbančič, T. & Bratko, I. (1994), Reconstructing human skill with machine learning, *in* "Proceedings of the 11th European Conference on Artificial Intelligence", John Wiley & Sons, pp. 498 – 502.

Valiant, L. G. (1984), "A theory of the learnable", *Communications of the ACM* **27**, 1134–1142.

Wang, X. (1994), Learning planning operators by observation and practice, *in* K. Hammond, ed., "Proceedings of the Second International Conference on AI Planning Systems", AAAI Press, pp. 335–341.

Wang, X. (1995*a*), Personal Communication.

Wang, X. (1995*b*), Learning by observation and practice: An incremental approach for planning operator acquisition, *in* A. Prieditis & S. Russell, eds, "Machine Learning: Proceedings of the Twelfth International Conference", Morgan Kaufmann, pp. 549 – 557.

Wang, X. & Carbonell, J. (1994), "Learning by observation and practice: Towards real applications of planning systems", *AAAI Fall Symposium on Planning and Learning: On to Real Applications*.

Watkins, C. (1989), Learning from Delayed Rewards, PhD thesis, Cambridge University. Psychology Department.

Whitehead, S. D. & Ballard, D. H. (1990), Active perception and reinforcement learning, *in* B. Porter & R. Mooney, eds, "Proceedings of the Seventh International Conference on Machine Learning", Morgan Kaufmann, pp. 179–188.

Yamauchi, B. & Langley, P. (1996), Place recognition in dynamic real-world environments, *in* "Proceedings of ROBOLEARN-96: International Workshop for Learning in Autonomous Robots".