Acquiring Domain Knowledge for Planning by Experimentation

Yolanda Gil

August 24, 1992
CMU-CS-92-175

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

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

This research was supported by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The view and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

Keywords: planning, learning, experimentation, theory refinement.

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Acquiring Domain Knowledge for Planning by Experimentation

YOLANDA GIL

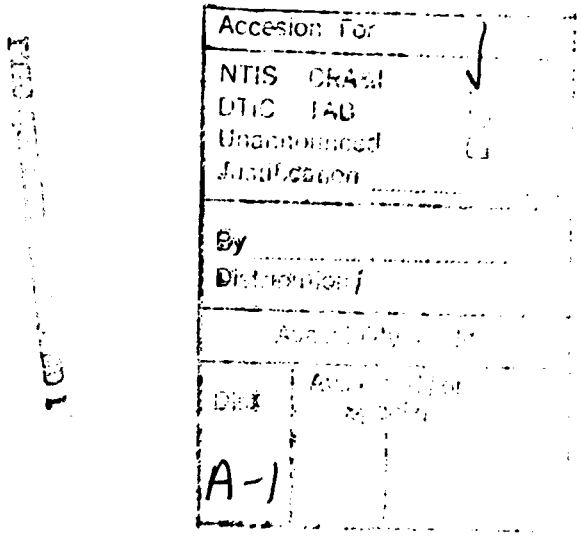
Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

J. G. Carbonell	8/6/92
MAJOR PROFESSOR	**DATE**
R. Reddy	8/21/92
DEAN	**DATE**

APPROVED:

Paul Christiano	25 August 1992
PROVOST	**DATE**



Abstract

In order for autonomous systems to interact with their environment in an intelligent way, they must be given the ability to adapt and learn incrementally and deliberately. It is virtually impossible to devise and hand code all potentially relevant domain knowledge for complex dynamic tasks. This thesis describes a framework to acquire domain knowledge for planning by failure-driven experimentation with the environment. The initial domain knowledge in the system is an approximate model for planning in the environment, defining the system's expectations. The framework exploits the characteristics of planning domains in order to search the space of plausible hypotheses without the need for additional background knowledge to build causal explanations for expectation failures. Plans are executed while the external environment is monitored, and differences between the internal state and external observations are detected by various methods each correlated with a typical cause for the expectation failure. The methods also construct a set of concrete hypotheses to repair the knowledge deficit. After being heuristically filtered, each hypothesis is tested in turn with an experiment. After the experiment is designed, a plan is constructed to achieve the situation required to carry out the experiment. The experiment plan must meet constraints such as minimizing plan length and negative interference with the main goals. The thesis describes a set of domain-independent constraints for experiments and their incorporation in the planning search space. After the execution of the plan and the experiment, observations are collected to conclude if the experiment was successful or not. Upon success, the hypothesis is confirmed and the domain knowledge is adjusted. Upon failure, the experimentation process is iterated on the remaining hypotheses until success or until no more hypotheses are left to be considered. This framework has shown to be an effective way to address incomplete planning knowledge and is demonstrated in a system called EXPO, implemented on the PRODIGY planning architecture. The effectiveness and efficiency of EXPO's methods is empirically demonstrated in several domains, including a large-scale process planning task, where the planner can recover from situations missing up to 50% of domain knowledge through repeated experimentation.

Acknowledgements

I would like to thank many people have made the work on this thesis an education, a challenge, and a great time. The thesis work was a great excuse to enjoy many discussions with the members of my reading committee. Jaime Carbonell coached me through every stage of graduate school. I learned with him about learning, and about learning about learning. He has been a mentor, and an invaluable friend. Herb Simon showed me to care about both the big picture and the small details. Tom Mitchell always pointed towards his robot in the lab, and made me think about how real the work aould be made. Nils Nilsson was a perfect external committee member: he reminded me that there was a world outside of Wean Hall, and one that would be interested in seeing my work completed.

The Computer Science department has provided from my very first day an incredible environment for research. I thank each and everyone of its members for making it what it is, and keeping it that way. Nobody around here knows exactly how the culture in this department came about. But everyone is certain that Allen Newell had to do with it a great deal. Learning from him was a great privilege. He will be greatly missed.

The work always benefited from discussions with many people, especially Kevin Knight, Craig Knoblock, Eduardo Perez, Alicia Perez, Manuela Veloso, Alan Christiansen, Wei-Min-Shen, Steve Minton, and Caroline Hayes. Caroline introduced me to the realm of machining metal parts, which I used as a domain in the thesis. Craig and Steve had always many good pointers. Weekly meetings with them and other members of the PRODIGY group were invaluable. Santiago Rementeria shared our interest in experimentation the most.

Back in Spain, many people supported me in every possible way, including Angel Alvarez, Jose Cuena, Julio Gutierrez, and Eduardo Perez. Here is looking at you, kids.

Thanks also go to the following people for lending me a hand (most of the times two): Anh Nguyen, Vince Cate, Isaac Gil, Antonio Gil, Juan F. Gil, Gary Knight, Jean Harpley, Alicia Perez, Robert Joseph, and many others. Efforts from the speech group made a big difference, especially Alex Hauptmann, Alex Rudnicky, and David Steere.

Contents

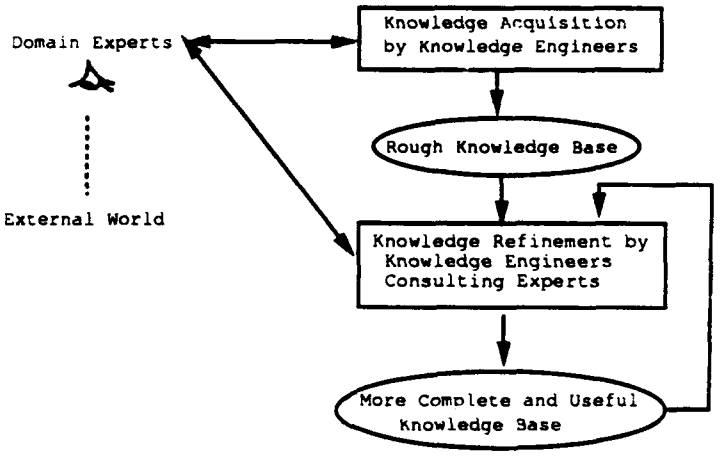
Chapter 1

Introduction

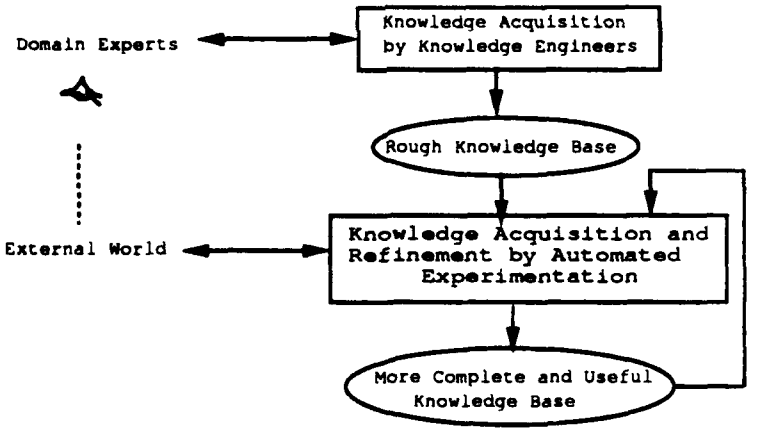
Learning has proven to be a vital ingredient in transforming planners from research tools into real world applications. Of foremost concern has been the area of improving the efficiency of planning. The learning techniques that have been applied range from macrooperator learning [Fikes *et al.*, 1972; Korf, 1985] and acquisition of control knowledge for guiding search [Mitchell *et al.*, 1983; Minton, 1988; Etzioni, 1990], to the synthesis of abstraction hierarchies [Sacerdoti, 1974; Christensen, 1991; Knoblock, 1991]. These techniques fall under the rubric of speed-up learning, and they share the property of acquiring more effective ways of expressing the knowledge that the system already implicitly has. After learning, a planner solves more efficiently the same kinds of problems that it is able to solve before learning. In other words, it is able to solve more problems within a given time bound. This type of learning is also known as *symbol-level learning* [Dietterich, 1986].

But learning is also necessary in other dimensions of planning systems. The representation given to the planner is bound to contain many inaccuracies, which may be corrected automatically through a learning cycle. Human planners in any sizable domain (e.g. factory production planning, routing in transportation planning, configuration in telecommunication networks, and so on) rarely make the assumption that they have omniscient world knowledge. A much more realistic assumption is that given domain knowledge is operationally accurate and complete, but there is a recovery procedure to acquire more knowledge or correct existing knowledge if and when this assumption is violated. Learning has, in this case, a different meaning. The new knowledge will enable the planner to solve problems that it was not capable of solving before learning no matter what the time bound was. As Newell describes this situation [Newell, 1982]:

> "... When we say ... that a program "can't do action A, because it doesn't have knowledge K", we mean that no amount of processing by the processes



(a) Traditional knowledge acquisition and refinement



(b) Automated knowledge refinement and incremental acquisition by experimentation

Figure 1.1: Knowledge acquisition and refinement using experimentation. An initial knowledge base is obtained from the domain experts, but is refined autonomously through direct interactions with the external world.

> now in the program on the data structures now in the program can yield the selection of A."

A qualitative augmentation of the knowledge available to a planner goes beyond the

reformulation of its initial knowledge. This type of learning is known as learning at the *knowledge level* [Dietterich, 1986]. This area has received less attention from the planning and learning communities, but it is of major importance for building autonomous intelligent systems.

Augmenting incomplete models benefits planning in three different ways. First, the coverage is expanded because a planner can solve more problems after acquiring the knowledge needed. Second, the prediction accuracy is raised since learning side effects and unusual conditions allow for planning further ahead. Lastly, the ability to adapt provides increased autonomy to the planner.

Many systems for guiding the acquisition of knowledge can be found in the literature (see [Marcus, 1990] for an overview). Knowledge acquisition tools provide a framework for the direct interaction of knowledge engineers with domain experts. The resulting knowledge base is an approximate model of the domain, whose degree of correctness and completeness varies with the complexity of the task domain. The knowledge engineers engage in test-and-revise procedures to refine the knowledge base asymptotically until a satisfactory model is obtained, as shown in Figure 1.1(a). Our work is concerned with the acquisition of knowledge for planning domains. None of the current knowledge acquisition systems are designed for planning domains nor do they emphasize full automation. Planning systems offer the possibility of direct interaction with the environment. The autonomous refinement and acquisition of knowledge by directed experimentation is invoked once an initial approximation of the knowledge base is available, as shown in Figure 1.1(b). Such is the learning model presented in this thesis: a failure-driven experimentation-based method for incremental acquisition of domain knowledge. In essence, impasses in planning or divergences between internal expectations and external observations trigger the learning procedure. Learning is autonomous and unsupervised, the interaction with the environment being the only source of additional knowledge.

1.1 Learning By Experimentation

Figure 1.2 gives an overview of the experimentation process described in this thesis. Learning is triggered when one of the actions in a plan has unexpected consequences. The first step is to come up with a set of hypotheses that might explain what went wrong. The next step is to choose a hypothesis from that set and devise an experiment to test it. This may involve creating a certain state of affairs, which may itself require planning to set up the experiment. Similarly, after the experiment is concluded (successfully or unsuccessfully), planning may be necessary to return to the state of the world that existed before the experiment. These stages are familiar as a part of the scientific method. But

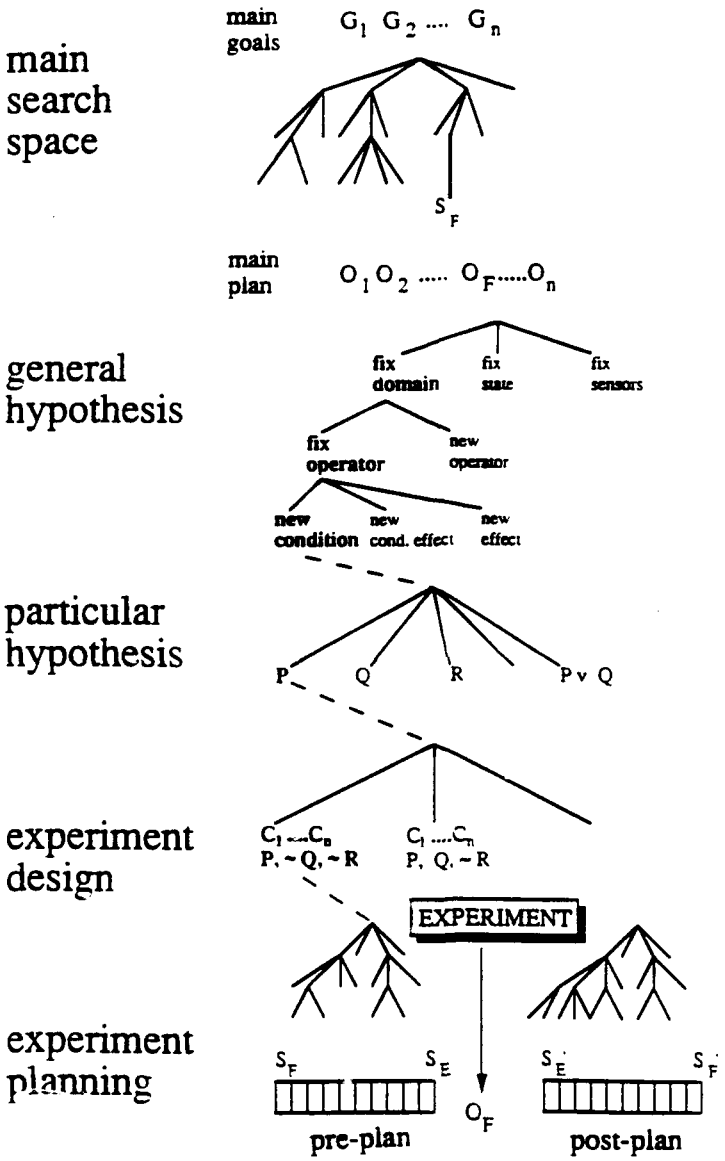


Figure 1.2: Experimentation at a glance. Failures in the execution of a plan trigger learning. A general cause for the failure is hypothesized, then instantiated to a particular hypothesis. The design of experiments includes planning the experimental setup.

humans demonstrate in their everyday life that experimentation is also a powerful tool for acquiring knowledge outside of the laboratory.

EXPO, the system described in this thesis, automates this experimentation process and shows that it is a useful technique for acquiring knowledge for planning systems. Each of EXPO's stages is described in detail in the succeeding chapters. To illustrate some of the issues involved, we turn now to an example of how people use experimentation to autonomously augment their knowledge about the world.

Consider the problem of getting ready for work in the morning. Given adequate domain knowledge, we can easily come up with a plan to achieve this goal. Suppose that one of the subgoals is drying one's hair. One possible plan to achieve this goal is: get hair dryer, plug in hair dryer, turn on hair dryer, and blow hair dry. But sometimes our actions may not yield the expected results when executed in the real world. For example, suppose that one day the hair dryer fails to function when we turn it on. At this point, we have two alternatives. One is to find another person (if one is available) and ask for an explanation. The other is to engage in experimentation to determine the cause of the failure. The advantage of experimentation is that learning is done autonomously, an important ability of human beings that we would like to model in our intelligent systems.

The first step of the experimentation process is to generate hypotheses that explain the failure. One general class of hypotheses is that the person's knowledge about the state of the world is incorrect. For example, three particular hypotheses might be:

- the hair dryer is broken
- the outlet is broken
- the hair dryer is not firmly plugged in

Another general class of hypotheses is that the person's model of the action is incorrect. In this case, we look for conditions under which we are currently executing the action that may cause the failure. For example:

- Today is Saturday, and the hair dryer does not work on Saturdays.
- It is noon, and the hair dryer only works in the morning.
- The light switch of the bathroom is off, and the hair dryer works only when the light switch is on.
- The bathroom window is open, and the hair dryer only works when it is closed.
- The door is open, and the hair dryer only works when it is closed.

We may construct other hypotheses, but let us restrict this discussion to the ones above. The next step is to choose a hypothesis and then design and perform experiments that prove or disprove it. Suppose that we decide to check first that the state of the world is actually what we believe.

The first step is to calibrate the hypotheses and choose which ones to look at first. The first one (broken hair dryer) seems hard to test for someone who is not mechanically inclined. So we decide to try the other hypotheses first. To test if the hair dryer is not firmly plugged in, we do a simple experiment: we plug the hair dryer in firmly and turn it on again. This does not make the hair dryer work, so the hypothesis is rejected. This experiment was quite simple, but our next hypothesis requires a more elaborate one. Suppose that the outlet is broken. One possible experiment to test it is to plug another device in the outlet. To do so, we build a plan with the following steps: unplug hair dryer, get another device (maybe an old hair dryer), plug in device. This plan brings a state of affairs where we can do our experiment: to turn on the device and see if it works. After turning it on, we observe the results: the device is operating. This disproves the hypothesis that the outlet is broken, and we move to consider another hypothesis. But first, we need to go back to the state of affairs before the experiment. So we create a plan to plug the hair dryer back in: turn device off, unplug device, store device away, plug in hair dryer.

Now we are ready to look at another hypothesis. For example, we may consider now conditions under which we are trying to make the hair dryer work, which is our second class of hypotheses. Again, we calibrate them and decide which ones to consider first. Our previous experience with hair dryers helps us decide which hypotheses are more likely to be relevant. We have successfully used our hair dryer at various times and on different days of the week, so the first two hypotheses are ruled out. The third hypothesis is more plausible, because everytime we have used the hair dryer before the lights were on and they are off now. So we try an experiment. We build a plan to turn on the bathroom lights, and then we turn on the hair dryer. This makes it work because the light switch controls the power of the outlet. So we conclude that the hair dryer can only be used in this bathroom when the lights are on.

In summary, with this type of experimentation people acquire autonomously knowledge about the environment that is necessary for solving problems. Notice that in this example in particular, but also in general, we did not rely upon any detailed knowledge about hair dryers, power outlets, or light switches to guide the experimentation process. The automation of this experimentation process based on shallow experiential knowledge is the main concern of this thesis.

1.2 Methodology

The aim of this thesis is to contribute to learning at the knowledge level autonomously from the environment. In this thesis, a complete and correct domain is used to simulate the real world. The planner is given an incomplete version of the domain that, via experimentation, it attempts to flesh out incrementally into a complete model. The new knowledge learned by experimentation is incorporated into the domain and immediately available to the planner. The planner in turn provides a performance element to measure any improvements in the knowledge base. This is a closed-loop integration of planning and learning by experimentation. The thesis provides a theoretical framework for this integration, as well as a practical demonstration in a system called EXPO and its interaction with the PRODIGY planner [Minton *et al.*, 1989a; Carbonell *et al.*, 1991; Carbonell *et al.*, 1990].

The planner is given some initial knowledge base that may contain a number of imperfections, each with its own idiosyncracies. Incorrect facts may lead to contradictions. Lack of knowledge limits the capabilities of the planner. This thesis concentrates on the refinement of knowledge bases that are initially incomplete, i.e., ignorant of facts that are true and needed for the task at hand. The lack of information to solve a task causes a knowledge impasse that triggers learning. We do not address curiosity-driven exploration. Learning is always driven by the need to accomplish some task. One way to solve knowledge impasses is by directed experimentation. This thesis presents methods that set context for systematic experiments (i.e., what type of knowledge is missing, where it is missing,...) to address different faults of the domain knowledge. These methods are domain independent, yet they are shown to be very effective through empirical tests of EXPO.

Once a context is set for the experimentation, we address the issue of the design of specific experiments. Not all experiments are equally desirable. Changing one variable at a time, minimizing interaction with the environment, minimizing resource consumption are among the heuristics typically proposed. This thesis shows that good choices can be made by domain independent rules that can be used to define experimentation strategies.

A domain-independent approach is certainly desirable. But an additional aim of the thesis is to rely exclusively on the knowledge given initially for planning. This means that the learning occurs even when no causal, structural, or common sense knowledge (other than the one embedded in the domain model) is available. This is a major advantage, since we do not need to address in turn the acquisition and refinement of that additional and necessarily complex background knowledge.

Not only is our methodology applicable across domains and independent of additional knowledge, it also yields efficient learning. This is shown by EXPO's empirical results in two different domains, one of them of considerable size and complexity.

1.3 The Application Domains

The methods described in this thesis were tested and evaluated in two domains: a robot planning domain and a complex process planning domain.

The robot planning domain [Minton *et al.*, 1989b] is an extension of the one used by STRIPS [Fikes and Nilsson, 1971] that has been used in other research [Minton, 1988; Etzioni, 1990; Knoblock, 1991]. A robot can push and carry objects to move them between rooms. Rooms are connected through doors, that may be opened, closed, and locked or unlocked with the appropiate keys. The rooms can be in any topological configuration and there can be any number of rooms, doors, keys, and boxes. The domain is described in detail in Appendix A.

The process planning domain contains a large body of knowledge about the operations necessary to machine and finish metal parts [Gil, 1991]. This domain was chosen because it has considerable size in many dimensions (one order of magnitude bigger than most planning domains in the AI literature), which makes the empirical results of the thesis more definitive and scalable. A typical problem in this domain is to produce a rectangular block of 5" x 2" x 1" made of aluminum and with a centered hole of diameter 1/32" running through the length of the part. To perform a machining operation in a part, the part must be securely held by some holding device. Each machine uses different tools, and the appropriate tool for the operation must be installed in the machine. Appendix B can be consulted for more detailed information on this formalization.

1.4 Summary of Contributions

The contributions of this thesis are:

- A domain-independent method to acquire domain knowledge for planning
- Identification of important issues for experimentation in planning
- Computationally effective methods for augmenting incomplete domain knowledge
- Zero-knowledge heuristics for finding relevant hypotheses
- Methodology for planning efficient experiments
- Full implementation integrated in PRODIGY to acquire knowledge effectively in two domains
- Empirical validation of the methods via the PRODIGY implementation and thorough testing

1.5 Organization of the Thesis

The chapters in this thesis are organized as follows.

Chapter 2 presents the related work. This chapter includes a review of work on experimentation, repairing plan failures, and learning from the environment. However, this thesis presents the first work on a planner that learns from the environment using sophisticated experimentation techniques. Chapter 2 also discusses research on less directly related areas such as rule induction and imperfect theory refinement.

Chapter 3 provides the context for the experimentation system described in the succeeding chapters. It begins by describing the type of domain knowledge available to a planner and the possible imperfections of that knowledge. Acquiring domain knowledge is then cast in terms of concept learning, a well understood framework in which an experimenter can be described as a learner that is active in the selection of examples. The chapter turns next to how a planner can monitor the external world to detect plan execution failures, and how it can manipulate the external world through experimentation. This experimentation serves to pinpoint specific imperfections in the knowledge base—only the ones responsible for plan failures. We call this type of experimentation *task-driven* experimentation and it is contrasted with other types of experimentation in the chapter. The chapter also discusses what it means for an experimenter to be efficient. It finishes with a description of the PRODIGY planner, on top of which our experimentation work is built.

Chapter 4 describes the experimentation process as implemented in the EXPO system. This process involves detecting a knowledge impasse, choosing promising hypotheses to overcome it (which EXPO does using domain-independent heuristics), designing experiments, executing them, and incorporating newly discovered facts into the planner's knowledge base. The chapter describes in detail how new preconditions are learned by EXPO.

Chapter 5 takes a broad view of methods for learning by experimentation. It is a comprehensive survey of various types of incompleteness that can exist in a planner's knowledge base. For each type of incompleteness it describes how experimentation techniques can be used to locate and repair faults.

An empirical analysis of EXPO's performance is presented in Chapter 6. Two different types of tests were run. In the first case, EXPO is shown to be effective in that the planner is able to solve many more problems after learning—using the knowledge acquired by EXPO—than it could solve with its initial knowledge. Note that it is not a matter of solving the problems faster, but rather a matter of whether the problems are solvable at all. The second type of test analyzes how efficient EXPO is with respect to the number of experiments that it performs and the amount of effort required to perform them.

Finally, Chapter 7 presents conclusions, the limitations of this work, and suggests directions for future research. Two appendices follow; they describe in detail the application domains both qualitatively and quantitatively.

Chapter 2

Related Work

This chapter presents a discussion on previous work related to this thesis. The first section reviews the topic of experimentation in the AI literature. Work on concept learning, both theoretical and practical suggests that active learners (ones that participate in the learning process by asking their own questions, often posed as experiments) are more powerful than passive learners. The section also examines experimentation in scientific discovery systems. Section 2.2 discusses planning systems that learn by experimentation and planning systems that learn from their interaction with the environment. The final section reviews some relevant work on theory refinement and rule induction.

2.1 Experimentation

This section reviews related work on the topic of experimentation. The work is divided here into two areas: concept learning and scientific discovery. Section 2.2 contains references to some systems that use experimentation for acquiring domain knowledge for planning.

2.1.1 Experimentation in Concept Learning

Active learners (ones that participate in the learning process by asking their own questions) that have the ability to formulate experiments are believed to be much more powerful and efficient than passive learners that do not have that ability. Results in formal learning theory show that finding a consistent hypothesis is NP-hard for many classes of representations of concepts [Pitt and Valiant, 1988; Haussler, 1989]. These results are based on a scheme in which a passive learner collects instances through an

oracle called EXAMPLES. The learner calls the oracle, which randomly chooses an example along with its classification as positive or negative. The use of this oracle may be one of the core reasons for the discouraging results that have been obtained [Haussler, 1988]. In fact, humans seem to be more effective learners than the results show. This may well be because the oracle EXAMPLES involves a very passive attitude on the part of the learner. Research on other types of oracles shows better results [Angluin, 1987]. In particular, membership oracles accept an instance as an input and return its classification (positive or negative). This type of oracle resembles more realistic set ups for learning. Amsterdam [1988] proposes an oracle called EXPERIMENT, that accepts a partial description of an example and returns a complete description (if there exists any). EXPERIMENT is shown to be more powerful than EXAMPLES.

If the learner has the capability to choose examples, how should that choice be influenced? Again, research in formal learning theory has tried to characterize "good" and "bad" examples [Rivest and Sloan, 1988; Ling, 1991]. Learning algorithms are faster with good examples, and learning speed degenerates when the quality of the examples decreases.

Factorization of concepts into independent relations seems to be a powerful technique for generating discrimination experiments efficiently in version spaces [Subramanian and Feigenbaum, 1986]. [Gross, 1988] shows that selecting examples to reduce the difference between a concept description and its curent maximum generalization is more effective than selecting examples at random. A similar experimentation technique is used in [Sammut and Banerji, 1986]. [Ruff and Dietterich, 1989] presents a study on the effectiveness of experiments. The performance of several experimentation strategies was tested on Boolean function learning. The results show that the ability to do any kind of experimentation dramatically increases performance. Simple but clever experimentation strategies were found to be almost as effective as sophisticated and expensive ones. One could argue that the optimal experimentation strategy is one that would generate examples close to the ones that a good teacher would [VanLehn, 1987; Salzberg *et al.*, 1991], and far from the ones that a non-cooperative teacher would generate [Dent and Schlimmer, 1990]. However, it is not possible to generate the optimal sequence of examples (experiments) unless the concept is known beforehand and the appropriate near-misses can be generated [Winston, 1975].

What do these results in experimental and formal concept learning tell us? First, that it is important that the learner be active in the learning process. This is why active, directed experimentation is a very promising approach for learning. Second, that the nature of the examples greatly influences the speed of the learning: good examples make learning faster. In other words, good experiments make learning faster.

2.1.2 Experimentation in Scientific Discovery Systems

Experimentation is a vital component of science. Most scientific discovery programs use the results of experiments to formulate quantitative or qualitative laws, such as [Langley *et al.*, 1987; Falkenheiner and Michalski, 1986; Nordhausen and Langley, 1992]. It is the user who designs the experiments, executes them, and provides the system with the results. Recently there has been an increasing interest on modeling scientific experimentation in recent programs.

COAST

Explanation-based theory revision [Rajamoney, 1988] is a method that uses experimentation to augment and correct theories. It is demonstrated in COAST, a system that revises qualitative theories of physical world processes, like evaporation and osmosis. COAST detects a fault in the theory when (1) an observation cannot be explained, (2) the predictions contradict the observations, and (3) multiple explanations can be built for a given observation. Then, it uses a set of theory revision operators together with constraints (like the type of failure, the situation in which it happened, etc) to produce a set of revised theories. These theories can be tested together by building abstract hypotheses that cover a number of them. The abstract hypotheses are used to build an explanation for the failure. The hypotheses are then tested through experimentation or through previous observations. From all the revised theories that pass the test, one is selected based on simplicity and predictive power.

Let us take a closer look at what is called in COAST *experimentation-based hypothesis refutation*. First, the hypothesis is used to create a prediction that specifies the values of variables that agree with the theory. Then experiments are designed that determine the experimental values of those variables. COAST inplements three strategies for designing experiments. *Elaboration* selects a variable to be measured according to the ease of the measurement. *Discrimination* prefers variables whose predicted values are different for different hypotheses. Finally, the *transformation* strategy produces totally new setups for doing experiments when the possibilities of the current one have been thoroughly exhausted. A more detailed study of discrimination and transformation experiments is presented in [Rajamoney, 1992]. [Falkenheiner and Rajamoney, 1988] presents a method for combining experimentation-based theory revision with analogical reasoning.

In brief, COAST uses experiments to test revisions of theories about physical processes, and relies heavily on the ability of those theories to produce explanations. The design of experiments involves choosing which variables to observe and which values they take under the theory being tested. In contrast, EXPO does not try to learn about how processes evolve in the physical world. Rather the domain knowledge models the conditions and effects of actions over which the planner has control. EXPO does not

have access to a theory that produces explanations for failures as COAST does, since its only available knowledge is the domain operators for planning and they are incapable of producing such explanations. EXPO's hypotheses are produced without looking at the semantics of a failure encountered. While explanations are powerful, we wanted to investigate the potential of a theoryless system, which turns out to have impressive performance.

KEKADA

KEKADA [Kulkarni, 1988] implements a set of experimentation strategies that model scientists at work. It simulates the discovery of the ornithine cycle based on Hans Krebs accounts.

KEKADA's experimentation strategies are implemented as heuristic operators, which are grouped into categories as follows. *Problem choosers* decide which problem to focus on. *Hypothesis generators* create hypotheses about the problem at hand. Then, the *hypothesis or strategy proposers* decide which hypothesis to concetrate on or which strategy to use to work on the problem. The *experiment proposers* design experiments based on the hypotheses. Then, *expectation setters* find out from the hypotheses what the results of the experiments are expected to be. The *experimenters* carry out experiments. Next, the results of the experiments are analyzed by the *hypothesis and confidence modifiers*, which modify the hypotheses and the confidences about them. Finally, if the expectations for the experiments do not agree with the observations the *problem generators* propose to study this phenomenon. When there is more than one alternative in any of the above decisions, *decision maker* heuristics are used to make a choice.

There are several strategies available to the strategy proposers: (1) magnify the phenomenon by varying the values of variables in the experiments, (2) divide and conquer to isolate subprocesses, (3) determine the scope of the phenomenon using an object type hierarchy, (4) determine which factors are necessary for the phenomenon to occur, (5) to relate the phenomenon to another one, (6) to gather more data about the phenomenon systematically, and (7) domain-dependent specialization of general strategies like controlled experimentation.

An experiment in KEKADA is specified by the following: the inputs, the conditions and the place for carrying out, the initial quantities of the inputs, and the observations to be collected after the experiment is carried out. The expectation setters form expectations for an experiment that consist of the expected output substances and the lower and upper bounds on the quantities and rates of those substances.

Thus, KEKADA's specifications of experiments are domain specific. KEKADA is given domain-dependent knowledge about substances, chemical reactions, and other people's experiments on urea synthesis that Krebs was aware of. About half of the heuristics

in KEKADA are domain dependent, although they can be used for other biochemistry applications. Most of KEKADA's domain-independent heuristics are used by EXPO, as discussed in Section 4.6.

STERN

STERN [Cheng, 1990] is a scientific discovery system that models experiments using Galileo's work on free fall. In STERN, hypotheses are expressed as equations. Experiments are used to (dis)confirm hypotheses and to generate new hypotheses.

Experiments are designed at three levels of abstraction. At the most abstract level, an experimental paradigm is chosen such as pendulums or inclined planes. At the next level, an experimental setup is chosen, which is a particular instantiation of the experimental paradigm. At this point, a particular inclined plane with concrete values for physical dimensions such as length, inclination angle, and height would be chosen. At the last, most concrete level, an experimental test is chosen. For example, we may choose to look at how the distance down an inclined plane varies with time.

The parameters involved in the experiment are classified as follows. One is chosen to be the output, another one manipulable, and the rest are considered constant. The constants are always set to the midpoint of their range, and the manipulable variable is given values within its range. The purpose of the experiments is to find out how the output variable is related to the manipulable variable with the other values held constant.

STERN uses two types of knowledge during experiment design. Pragmatic knowledge prefers paradigms with experimental setups that are easier to manufacture. For example, distance is easier to manipulate than time. Background knowledge eliminates experimental setups that are trivial. For example, given the angle of inclination of a plane and its length, the height can be geometrically deduced without need for experiments.

STERN has the ability to design new experimental paradigms by combining existing ones, such as an inclined plane and a projectile. This is necessary when it is not tractable to design experiments in an existing paradigm (for example, if a variable cannot be eliminated from an equation).

Because the experimentation space is quite large, STERN has some heuristics to improve its performance. The practicality of each paradigm, based on the number of setups and the ease of the manufacture of the setups, is used to activate paradigms.

EXPO's experiments are very different in nature from those of a scientific discovery system like STERN. STERN's hypotheses are equations, i.e., mathematical relations between variables. EXPO's hypotheses in our hair dryer example in Section 1.1 are a set of candidate conditions, i.e., predicates (possibly with several variables) that must be true for the action to work. STERN chooses in the equation an output variable, a manipulable

variable, and the rest are kept constant. Then it gives values to the manipulable variable and the constants, and observes the value of the output variable after the execution of the experiment. EXPO, on the other hand, does not need to classify the variables present in the candidate conditions. All the variables in the conditions, and many more, are instantiated by the planner when it is invoked to achieve the state in which to perform the experiment (this is explained in detail in Chapter 4). EXPO has many variables to observe after the experiment's execution, which correspond to all the known effects of the action. Also, STERN repeats experiments with the same setup and different values of the manipulable variable. EXPO, on the other hand, designs experiments so that a hypothesis is disconfirmed or confirmed after each one. Both EXPO and STERN prefer experiments that are easier to perform, and they both share a concern for the efficiency of the experimentation process.

FAHRENHEIT

FAHRENHEIT [Żytkow *et al.*, 1990] extends BACON's ability to discover quantitative laws from numerical data. The system determines not just the regularities of the set of variables, but also the range of values for which the functional relation holds. FAHRENHEIT makes BACON efficient through a multi-level search strategy by changing the order in which variables are considered.

FAHRENHEIT's experimentation ability greatly extends BACON. It automates the experiments and data collection through a hardware system that controls some equipment in a chemistry lab. The experiments are designed according to the current knowledge of the system.

Unlike FAHRENHEIT's, the parameters of EXPO's experiments can be nonumerical. FAHRENHEIT's techniques could be used by EXPO in numerical domains where the operators were applicable for certain values of their parameters (we discuss this in more detail in Section 7.3.1). FAHRENHEIT is given a physical configuration where the experiments are to be carried out. The experiments differ in the values that are given to the controllable parameters. In contrast, EXPO has to design the configuration state in which the experiment can be carried out, and build a plan to achieve such a state. Every experiment is different in nature from the rest, and the selection of designs that satisfy the user's requirements is of crucial importance for EXPO.

2.2 Planning and Learning from the Environment

As we mentioned in the introduction, there is considerable interest in planning systems that acquire control knowledge by introspection [Korf, 1985; Sacerdoti, 1974; Mitchell *et*

al., 1983; Laird *et al.*, 1986; Minton, 1988; Mostow and Bhatnagar, 1987; Veloso, 1992]. All these systems differ from EXPO in two major ways. First, they only learn control knowledge while EXPO concentrates on acquiring factual domain knowledge. EXPO is learning at the knowledge level, as opposed to the symbol level [Newell, 1982; Dietterich, 1986]. Second, they learn by introspection, and not from interaction with an external environment as EXPO does.

LEX

LEX [Mitchell *et al.*, 1983] is a system that has some experimentation capabilities to learn control knowledge by introspection in the domain of symbolic integration. The left-hand side of its heuristics are represented as version spaces.

LEX is composed of four modules. The problem generator proposes a new problem to solve. The problem solver searches for a solution to the proposed problem using the currently available heuristics. Next, the critic examines the solution trace and assigns credit to search steps leading towards or away from a solution. Each step may be classified as a positive or negative instance of one of the heuristics. Then, the fourth module, the generalizer, comes into play. It updates the version space that corresponds to the heuristic of each positive and negative instance. Then, the problem generator looks at the new definitions of the heuristics and proposes new problems to experiment with. LEX then enters a new generate-solve-critic-generalize cycle.

The problem generator is the module responsible for generating experiments. It prefers problems that can be solved with the current operators and heuristics, and problems whose solutions will provide informative instances. One way for a problem to be informative is to produce instances of existing partially learned heuristics. Problems of this kind are generated by choosing a partially learned heuristic, and creating a problem that matches some but not all the members of the version space of that heuristic. LEX does so by using a hierarchy of the types of mathematical functions that it can use in the problems. Another way in which a problem can be informative is that it may lead to create a new heuristic. Problems of this type are problems in which two operators are applicable but there is no current heuristic to recommend which operator to prefer.

LEX uses experimentation to acquire the left-hand side of control rules, while EXPO's intent is operator refinement. Additionally, LEX instantiates functions to create problems through a type of hierarchy. EXPO, on the other hand, has to design goal states with several predicates, and is concerned with the actual planning for achieving such goal states and the interaction of this planning with the main problem at hand.

CHEF

CHEF [Hammond, 1986] is a system that, like EXPO, learns from plan execution

failures. CHEF is a case-based planner for the domain of Szechuan cooking.

CHEF has a memory of plans that are recipes, and it uses them to create new ones. For example, suppose we want a recipe for beef with broccoli. CHEF retrieves a plan from its memory, say beef and green beans, and adapts it to meet the goals of the current problems. In this case, it would add a step to chop the broccoli. After coming up with a plan, CHEF simulates its execution in the real world. If the result of the simulation does not satisfy the goals of the problem, an expectation failure has been found. In this example, the simulator indicates that the broccoli is soggy, and not crisp as wanted. The simulator also returns an explanation of the failure: that the beef leaves water in the pan, and that water makes broccoli soggy. (This did not happen in the original recipe, because green beans are more sturdy). CHEF uses this explanation to repair the plan, adding an extra step to cook the broccoli first and then the beef. The new plan is stored in memory, indexed by the causes of the failure contained in the explanation.

The repair used in a plan may be transferred to a new problem that may have the same failure. For example, if CHEF is asked for a recipe for chicken and snow peas, it remembers the broccoli episode and anticipates a potential problem of plans that cook the chicken and the snow peas at the same time. It then uses the beef and broccoli recipe to create a plan that avoids the same failure.

So CHEF, like EXPO, learns to avoid plan failures. But one important difference is how learning is done. CHEF calls a simulator of the real world with a plan, and gets back a description of the failures of the plan together with a causal explanation for the failures. EXPO uses a simulator of the world as well, but it monitors the simulation step by step and detects local failures instead of being informed of them. EXPO determines the causes of failures by designing and executing experiments. It is not told about the causal chain that provokes a failure.

CHEF repairs *plans* that cause failures, and reuses them to avoid the same failure in future plans. EXPO learns to repair *operators* that cause failures, and uses the corrected operator to build plans that will not incur in the same failure. Thus, the granularity is different. This has to do with the fact that CHEF is a case-based planner, while EXPO is designed to learn domain knowledge for generative planners. CHEF learns to avoid cooking some vegetables with meats that sweat water. EXPO would learn to avoid cooking some vegetables in the presence of water (whichever its origins), thus covering a larger range of possible failure situations.

LIVE

LIVE [Shen, 1989] is a system that learns from its environment. LIVE is designed for exploration and discovery. It can formulate new operators by executing actions whose

conditions and effects are unknown. It can also formulate new terms if the language is insufficient. EXPO does not do any of this.

The most relevant part of LIVE is its method for refining operators by splitting existing ones. When the expected effects of an operator are not obtained upon its execution (i.e., a surprise is obtained), the operator's conditions are specialized to exclude the current type of situation. In addition, a new sibling operator is created with the existing operator's conditions and the effects actually obtained. (This method is similar in spirit to learning by discrimination [Langley, 1987].) EXPO on the other hand opts for learning only the specialized operator when it encounters an execution failure. The sibling operator is in practice accounting for a set of unexpected unwanted effects which does not agree with a task-directed approach like EXPO's.

LIVE uses experimentation to revise learned rules that prove to be too specific during planning. The experiment consists of an instantiation of the rule's sibling rule that involves applying the action to a different object in a situation that has not been seen before (and so is likely to produce a surprise). LIVE has a preference for experiments that can be immediately executed in the current state or in easily reachable states. EXPO designs experiments with varied conditions. It has a more flexible mechanism for experiment preferences, one that takes into account much more than the ease of execution. EXPO's domains are more complex in size than LIVE's domains.

CAP

CAP [Hume and Sammut, 1991] is a system that uses experiments to build a theory that can be used to recognize sequences of actions performed by other agents. When CAP observes such a sequence, it divides it into meaningful subsequences that are generalized using inverse resolution. The generalizations are tested with experiments.

The variables of an experiment are instantiated through inverse resolution, which also produces changes in the state of the external world if needed for the experiment. If the experiment is successful, then the action description is generalized. When an action cannot be used because a condition P is too specific, a new term $\sim P$ is created and a new action is postulated with $\sim P$ as a condition.

CAP, like EXPO, does some pre-planning for experiment setup. However, work on CAP up to date has not addressed the choice of experiments or the selection of pre-experiment plans, these being major issues for the design of EXPO. CAP detects faults in the domain theory when an action cannot be used to produce a proof. EXPO, on the other hand, detects faults in the domain theory when an action's execution fails that was believed to be a legal step of the plan. CAP refines precondition expressions by generalizing overly specific preconditions. EXPO, on the other hand, learns new preconditions and also new effects of operators.

Soar

The Soar architecture acquires control knowledge from human advice [Laird *et al.*, 1989; Laird and Rosenbloom, 1990] in a robotics environment. When no control knowledge is available to select an operator, Soar either makes a random choice or prompts for advice. When existing control knowledge is incorrect, Soar is forced to reconsider each decision and incorporate human advice. This advice consists both of recommendations and disrecommendations of operators. In contrast, EXPO concentrates on the acquisition of domain knowledge and never interacts with a human during learning.

Other Work on Planning and Learning from the Environment

[Kedar *et al.*, 1991] presents a system that refines operators by building causal explanations of their failures. The explanations are built using a set of domain constraints on the state descriptions. If the reason for the failure is a contradiction of the expectation and the domain constraints, then the difference between the expected and observed states is explained. The result of an explanation is a new precondition for the operator. If it is not possible to build an explanation, then a new operator with a variant outcome (the observed effects) is created. This is in the same spirit as LIVE and discrimination learning. If several explanations can be constructed, then there are several candidates for new preconditions. This may cause complications for [Kedar *et al.*, 1991]. EXPO's experimentation techniques could then be a good way to discriminate amongst the candidates.

Other systems have experimentation capabilities to learn from real robotic environments. [Christiansen, 1992] describes empirical learning of manipulations. Almost no prior knowledge is assumed. With almost no initial knowledge, the system designs experiments by giving values to the task parameters, performing the experiment, and clustering the parameter space according to the resulting state. The system demonstrates two experimentation techniques: random training, and strategic self-training. Random training involves a random choice of values for the experiment parameters. Strategic self-training explores the parameter space randomly until the execution of the action does not unfold as predicted. Then, a similar action is chosen by giving the parameter a new value chosen randomly from a constant interval around its current value. Extensive empirical tests in various manipulation tasks show that strategic self-training yields better theories than random training. Another such system is presented in [Gross, 1991]. Its experimentation design is more sophisticated, in that it is able to vary several parameters at a time. The parameter space is divided into regions. Two types of experiments, generalization and specialization, reduce uncertainty surrounding a region or within a region respectively. Each type of experiments is designed using a set of heuristics that decide the value of the parameter. The system dynamically defines new attributes, a very desirable capability

when learning from the environment. Both of these systems assume the parameters of the experiments to be numeric, discrete, and ordered. The experiments do not require any planning steps for setup, the action can be immediately executed. EXPO's required experimentation capabilities do not assume such restrictions in the parameter values, and produce more elaborate setups. However, these systems are able to deal with noise in the observations, while EXPO is not.

Another project on robots that learn is the subsumption architecture [Brooks, 1986; Maes and Brooks, 1990; Maes, 1991]. Actions are modeled as behaviors, whose conditions are conjunctions of binary perceptual features. The robot receives binary feedback which it uses to learn when to activate behaviors. Each behavior monitors the values of the percepts and detects their correlation with the feedback received. If there is a strong correlation with a percept, it is added as a new precondition of the behavior. Arbitration between behaviors is also achieved by tuning a network to the current goals. There is no directed experimentation in this framework, and learning takes the form of adaptive control. Other systems that learn to control their actions with this type of trial-and-error learning from experience are reinforcement learning systems [Sutton, 1990; Kaelbling, 1990; Watkins, 1989; Mahadevan and Connell, 1992]. These systems use subsymbolic models of the world. In contrast, EXPO has explicit descriptions of the conditions and the expected effects of actions.

Many planners use plan repair techniques to avoid plan failures [Sussman, 1975; Sacerdoti, 1977; Wilensky, 1983; Wilkins, 1988]. Their planning algorithms use plan modification strategies to solve interactions between plan steps during planning. But they assume that the domain knowledge is complete and correct. EXPO, on the other hand, does not make this assumption. It is given a plan that is believed to work based on the planner's expectations. EXPO can be surprised if it finds that the plan's execution fails, because of wrong expectations. EXPO concentrates on repairing the domain knowledge (not the plan) through experimentation. Armed with this new knowledge, the planner will not have the same wrong expectations in the future.

2.3 Theory Refinement and Knowledge Acquisition

Theory Refinement

In explanantion-based learning (EBL) [Mitchell *et al.*, 1986; DeJong and Mooney, 1986], a theory composed of rules is used to build an explanation that justifies why an example is an instance of the concept described by the theory. When the rules contain errors, no explanation may be constructed for some examples of the concept and (yet worse) an explanation may be built for instances that are not examples of the concept.

The refinement of theories for EBL has been a major focus of research, addressing different types of errors: incompleteness [Danyluk, 1991; Sleeman *et al.*, 1990; VanLehn, 1987; Genest *et al.*, 1990; Mahadevan, 1989; Kodratoff and Tecuci, 1991], incorrectness [Ourston and Mooney, 1990; Bylander and Weintraub, 1988], intractability [Tadepalli, 1989; Ellman, 1989; Chien, 1990; Flann, 1990], or combinations of these types of errors [Pazzani, 1988; Hall, 1988]. A theory is incomplete when only partial explanations can be built due to lack of information in the theory. The above mentioned systems refine the theory by building partial explanations and completing them using various techniques including inductive methods [Pazzani, 1988; Danyluk, 1991], analogical reasoning [Falkenheiner, 1989; Genest *et al.*, 1990], apprentice-type techniques [Kodratoff and Tecuci, 1991; VanLehn, 1987], and experimentation (see the COAST system in Section 2.1.2.)

Although EXPO is also designed to refine incomplete knowledge it acquires both conditions and effects of actions, which is quite a different type of rule than EBL rules. The failures obtained from executing actions are very different from explanation failures. There is no reason to believe that the same learning paradigms cannot be applied to refine incomplete domain knowledge, although this is an open issue.

Knowledge Acquisition

Many tools have been designed to aid in the engineering of knowledge bases (see [Marcus, 1990; Boose, 1992] for good overviews). The acquisition of knowledge is done through interaction with a human expert. EXPO, on the other hand, is given an initial knowledge base that is produced by the expert, and is able to acquire knowledge autonomously in domains that allow direct interaction with the system being modeled in the knowledge base.

2.4 Other Related Work

There is work in the field of fault diagnosis on violated expectations [Davis *et al.*, 1982; Genesereth, 1984]. Any disagreements between the expected behavior of a device and its actual behavior indicate malfunctions that must be repaired. In this literature, the term "failure" is used in a different sense than in the planning literature: faults are misbehaviors, and failures are the causes of faults. Many failures of a device may be possible causes of a fault, much in the way EXPO must consider many possible domain adjustments for a given execution failure. However, fault diagnosis systems find the causes of a fault by building a causal explanation of the fault, using a detailed theory of the functionality of the device (often a qualitative model) [Davis, 1984; Genesereth, 1984; Patil *et al.*, 1981; Pazzani, 1990]. Such models are clearly powerful, but extremely

difficult to craft. One positive and unique aspect of EXPO is that it is able to find the cause of a failure without relying on such models.

Chapter 3

The Role of Experimentation in Planning

As we saw in the previous chapter, experimentation techniques have been used for learning in various contexts. This thesis applies experimentation to learning from the environment in order for a planner to acquire the new knowledge necessary to accomplish each new task at hand. The purpose of this chapter is to explain how our work on experimentation fits into the context of planning.

We begin by describing our planning paradigm, and the type of domain knowledge that it uses. Then, Section 3.2 describes operators as concepts. Because concept learning is a well understood framework with many years of research behind it, it provides a useful perspective on the automatic refinement of operators. One important point is that a planner is given an initial body of knowledge, and these concepts are not initially empty. However, the initial definitions may contain various types of imperfections that need to be understood and addressed in an individual basis. Section 3.3 presents four types of imperfections that may occur in the knowledge base. Interaction with the environment to acquire new knowledge presents many issues still under research. Section 3.4 presents our assumptions and states the limitations of our approach in this respect. Then, Section 3.5 describes precisely our definition of experimentation. The experimentation process must be directed and efficient and this section explains why this is important within a planning context. Finally, Section 3.6 presents PRODIGY [Minton *et al.*, 1989a; Minton *et al.*, 1989b; Carbonell *et al.*, 1991], the particular system used for the implementation.

3.1 Domain Knowledge for Planning

Through many years of research in this area, different paradigms for planning have emerged, including the problem space framework [Newell and Simon, 1972], case-based planning systems [Kolodner, 1980; Hammond, 1986; Veloso, 1992], and plan refinement [Schoppers, 1989]. This work concentrates on the problem space framework. The planner is given a set of rules (called *operators*), each of which defines the legal transitions between states. Plans are found by searching through the space of possible states. Many planners have used this model, including STRIPS [Fikes and Nilsson, 1971], NOAH [Sacerdoti, 1977], and SIPE [Wilkins, 1988]. In essence, the planner is given a set of operators that model the possible actions. Each operator contains the conditions under which the action can be executed, and the effects of the action. The planner is also given an initial state, which is a model of the state of the external environment. Operators specify the legal transitions from one state to another. The search for a plan consists of trying different sequences of operators to reach a state that satisfies a given goal statement. The operators together with the legal states constitute the domain knowledge of the planner.

Consider our robot planning domain. An operator for opening a door is:

```
(OPEN
 (params (<door>))
 (preconds
     (and
       (is-door <door>)
       (unlocked <door>)
       (next-to robot <door>)
       (dr-closed <door>)
         ))
 (effects (
       (del (dr-closed <door>))
       (add (dr-open <door>))
         )))
```

The variable **<door>** is a parameter that can be instantiated to open particular doors. The preconditions that have to be satisfied in order to open a door are that the robot is next to a door, and that the door is closed and unlocked. The effects of the operator are expressed in two lists. The delete list (`del`) specifies the facts that are no longer true after the operator is applied. The add list (`add`) is composed of the facts that the application of the operator makes true. In our example, after opening a door, the door is no longer closed and it is open. To open door `Door12` we use OPEN with the variable **<door>** instantiated to `Door12`. `Door12` is a *binding* for the parameter **<door>**. When all the preconditions of an operator are satisfied in a state, then the operator is said to be

applicable. An operator is applied by changing the state according to its list of effects. If the current state S_A contains the following facts:

```
(is-a BoxA BOX)
(is-door Door12)
(in-room ROBOT Room1)
(in-room BoxA  Room2)
(arm-empty)
(connects Door12 Room1 Room2)
(dr-closed Door12)
(unlocked Door12)
(next-to ROBOT Door12)
```

then we can apply the operator [OPEN Door12] and obtain the following state S_B:

```
(is-a BoxA BOX)
(is-door Door12)
(in-room ROBOT Room1)
(in-room BoxA  Room2)
(arm-empty)
(connects Door12 Room1 Room2)
(dr-open Door12)
(unlocked Door12)
(next-to ROBOT Door12)
```

Notice that the operator is applicable in any state in which the robot is next to a door that is closed and unlocked. The preconditions of an operator represent the class of states in which the operator is applicable. In contrast, the effects do not express the class of states that result from the application of the operator. What they represent is the transformation itself, i.e., the additions and deletions that must be done on the state where the operator is applied. This asymmetry in the representation of the operators must be taken into account when learning domain knowledge. We explain why in the next section.

3.2 Refinement of Operators as Concept Learning

As we pointed out in the previous section, the preconditions of an operator represent the class of states in which the operator is applicable. In fact, the preconditions form a concept that expresses the (hepefully minimal) generalization of all those states. Similarly, the effects are a generalization of the transition between states that the operator represents. This means that learning the correct expression of an operator is, in fact, a matter of concept learning from examples [Michalski *et al.*, 1983]. This section shows

where these examples come from and how they can be used to learn the definition of an operator.

Building a knowledge base is a process that requires iteration to correct errors that keep lurking after each new version of the system. When users define operators for a planning system, it is not uncommon that they would forget to write a precondition, or a side-effect of the action. Suppose that a planner is given the following incomplete operator:

```
(OPEN'
 (params (<door>))
 (preconds
   (and
     (is-door <door>)
                                ;the condition (unlocked <door>) is missing
     (next-to robot <door>)
     (dr-closed <door>)
       ))
 (effects (
   (del (dr-closed <door>))
   (add (dr-open <door>))
     )))
```

Notice the missing condition (**unlocked <door>**). Now suppose that the planner is given the goal (**dr-open Door12**) in state S_A (shown in the previous section). The operator OPEN' can be applied to achieve the goal. And in fact, if the robot tries to open the real door represented by **Door12**, the door will open. This is because the door happens to be unlocked, so even if the planner is unaware of the missing condition, the execution of the action is successful. A state in which the execution of the action is successful can be considered as a positive example of the concept expressed in the conditions of the operator.

Consider now that the planner is given the goal (**dr-open Door23**) and the following initial state S_C:

```
(is-door Door23)
(next-to ROBOT Door23)
(unlocked Door23)
(closed Door23)
```

The operator OPEN' can be applied to achieve the goal. If the robot tried to execute this action it would also be successful, again because the unknown condition that the door must be unlocked happens to be true in S_C. In fact, this state is another positive example of the concept expressed in the conditions. We can generalize from states S_A

and S_C by replacing the constants **Door12** and **Door23** by the variable **<door>**, and say that a door can be opened when the following facts are true in a state:

```
(is-door <door>)
(next-to ROBOT <door>)
(unlocked <door>)
(closed <door>)
```

Now suppose that the goal is **(dr-open Door34)**, and the state S_D is:

```
(is-door Door34)
(next-to ROBOT Door34)
(locked Door34)
(closed Door34)
```

This time, the planner will also believe that it can use OPEN' to achieve the goal since all the conditions are true in S_D. However, if it tries to execute the action and open the door, **Door34** will remain closed. This is because this time the door does not happen to be unlocked. S_D can be considered a negative example of the concept that the preconditions of the operator represent.

In summary, when the planner is given the ability to execute actions in the external world and observe their effects, it can detect faults in the operators that model these actions. Each successful execution of the action corresponds to a positive example of the concept that the precondition expression should represent. Similarly, each failure is a negative example of that concept. So in fact, the problem of learning the precondition expression of an operator can be cast in terms of concept learning as follows:

Given:
a set of positive examples
(i.e., a set of states in which the action was successfully executed)
a set of negative examples
(i.e., a set of states in which the execution of the action failed)

Find:
a description that covers all the positive examples and that
does not cover any of the negative examples
(i.e., the generalization of the states in which the action
can be successfully executed)

The effects of an operator also represent a concept. This concept corresponds to the transformation that the operator causes in the state in which it is applied. For example, when OPEN' is applied in S_A, the following transformations occur:

```
(add (dr-open Door12))
(del (dr-closed Door12))
```

When OPEN' is applied in S_C , the transformation is:

```
(add (dr-open Door23))
(del (dr-closed Door23))
```

A generalization of these two examples of the transformation is:

```
(add (dr-open <door>))
(del (dr-closed <door>))
```

which correspond, in fact, to the effects of OPEN'. If some effect is missing, the problem will not be noticed locally (execution will be successful), but may be noticed later when the observed world state diverges from the predicted one. Notice that we always encounter positive examples of the transformation, since the known effects always occur when the conditions are true. So in fact the problem of acquiring the effects of an operator is also a concept learning problem:

Given:

a set of positive examples
(i.e., a set of states in which the action was successfully executed
and the resulting state)

Find:

a description that covers all the positive examples
(i.e., a minimal generalization of the transition between the states)

There are some references in the literature that consider the left-hand side of rules as concepts to be learned [Mitchell, 1978; Mitchell *et al.*, 1983; Langley, 1987; Langley *et al.*, In press]. However, none of this previous work has pointed out the fact that the effects of operators represent a concept and consequently view their acquisition as a concept learning problem.

Since we provide the planner with an initial domain, there is an initial description for the concepts of the precondition expression and effects. This initial description may be faulty in several ways that are described next.

3.3 Imperfections in Domain Knowledge

As we discussed in the previous section, the domain model that the planner is initially given is not necessarily perfect. Several types of imperfections can appear simultaneously

in a domain model. There have been several attempts to classify imperfections [Mitchell *et al.*, 1986, Rajamoney and DeJong, 1987; Huffman *et al.*, 1992]. This section presents a more exhaustive classification tailored to planning systems. For each imperfection, we discuss the types of planning failures that it causes. The section concludes with a more detailed description of the imperfections addressed by this thesis.

3.3.1 Incomplete Models

Incomplete models are those in which some aspect is missing. Known operators may be missing preconditions and/or effects. Entire operators may be absent from the model.

Let us examine first the case of incomplete preconditions. Consider the operator OPEN' from the previous section. Again, OPEN' is incomplete: it is missing the condition `(unlocked <door>)`. As we saw in the previous section, when the planner executes OPEN', the action has no effects when the door happens to be locked. If that is the case, the planner makes the wrong prediction (that the door will be open). So if the preconditions of an operator are incomplete, the planner's predictions will fail because the effects of the operator will not be obtained.

Now let us look at a case when the effects of an operator are incomplete. Consider for example the following operator:

```
(PUTDOWN'
 (params (<ob>))
 (preconds
   (holding <ob>))
 (effects
   ((add (arm-empty))
                                    ;the effect (del (holding <ob>)) is missing
    (add (next-to robot <ob>)))))
```

Notice that the operator is incomplete: it is missing the effect that should delete `(holding <ob>)`. When a planner executes `PUTDOWN'`, it will obtain the desired effects. However, it will continue to believe that the robot is holding the object. So in the case of incomplete effects, the planner's predictions will fail when the wrong fact is used in the future.

Incomplete effects may also force the planner to do unnecessary work. Consider the following operator:

```
(PUTDOWN''
 (params (<ob>))
 (preconds
   (holding <ob>))
```

```
(effects
  ((add (arm-empty))
   (del (holding <ob>))
                          ;the effect (add (next-to robot <ob>)) is missing
))
```

Now suppose that the planner is given the goal (and (arm-empty) (next-to robot BoxA)) when the robot is holding BoxA. The planner builds a two step plan that uses PUTDOWN" first to achieve (arm-empty) and then GOTO-OBJ to achieve (next-to robot BoxA). Notice that this last step is unnecessary, but the planner believes it is needed because it ignores the fact that PUTDOWN" also achieves (next-to robot BoxA). Thus, unknown effects may cause the planner to build unnecessary subplans.

A domain model is also incomplete when entire operators are missing. For example, suppose that no operator is available for opening doors. In this case, the planner has strong limitations as to the problems that it can solve.

Another case of incompleteness occurs when a state is missing facts about the world. For example, consider a state containing a description of a door Door45 that connects Room4 and Room5. The state does not contain information about the door being either locked or unlocked. In this case, some operator's preconditions will not be matched in the state. So when facts are missing from the state, the applicability of operators is restricted to the known facts.

3.3.2 Incorrect Models

Incorrect models have some aspect that does not correspond to reality, or contain overly specific knowledge. This happens when an operator has erroneous conditions or effects, or some conditions or effects that are overly specific.

Let us consider the first case of erroneous conditions.

```
(OPEN''
 (params (<door>)
 (preconds
   (and (is-door <door>)
        (next-to robot <door>)
        (unlocked <door>)
        (dr-closed <door>)
        (holding <door>)))              ; this condition is erroneous
 (effects
   ((del (dr-closed <door>))
    (add (dr-open <door>)))))
```

Notice that this operator has an incorrect condition: it requires that the robot is holding the door. When a planner tries to use OPEN" in a plan it will always fail, since there is no way for the robot to be holding the door. So when a condition is erroneous, it may not be possible to use the operator to construct a plan.

Let us look at another case of erroneous conditions. Consider the following operator:

```
(OPEN'''
 (params (<door>)
 (preconds
   (and (is-door <door>)
        (next-to robot <door>)
        (unlocked <door>)
        (dr-closed <door>)
        (next-to <box> <door>)))              ; this condition is erroneous
 (effects
   ((del (dr-closed <door>))
    (add (dr-open <door>)))))
```

In this case, the erroneous condition can be achieved by the planner, so this operator can be used to construct a plan. However, the part of the plan that places the box next to the door is, as we know, totally unnecessary for opening the door. So an erroneous condition may force the planner to create plans that are longer than needed in order to achieve unnecessary subgoals.

Now let us look at the case of overly specific conditions. Consider for example the following operator:

```
(OPEN''''
 (params (<door>)
 (preconds
   (and (is-door <door>)
   (next-to robot <door>)
   (unlocked <door>)
   (dr-closed <door>)
   (color-of <door> RED)))              ; this condition is overly specific
 (effects
   ((del (dr-closed <door>))
    (add (dr-open <door>)))))
```

The predicate `(color-of <door> RED)` is unnecessary, making the precondition expression too specific, since the operator can only be used when the door to be opened is red. Non-red doors can never be opened. So when a condition of an operator is overly specific, the planner's capabilities are restricted with the more limited range of applicability of the operator.

The facts that the state contains can be incorrect as well. For example, the planner may contain the fact (locked Door12) when the door is, in fact, unlocked. In this case, some operator's preconditions will be matched in the state when the action is not applicable, and vice versa.

3.3.3 Inadequate Models

Inadequate models are those whose language lacks the appropriate primitives to express the aspects of the external world that are needed for problem solving. Consider OPEN'. If the predicate (unlocked <door>) was not only missing from the preconditions but also did not exist in this domain, the planner would not be able to reason about locks in the doors, thus failing to open any locked door.

3.3.4 Intractable Models

Intractable models are those in which it is prohibitively expensive (time-consuming) to derive a plan. In this case, control knowledge is needed to direct the search. As we mentioned in Chapter 2 much research has been done to address intractable domain models by learning control knowledge to expand the boundary of problems solvable with given time restrictions.

3.3.5 Types of Incompleteness

This thesis is concerned with refining incomplete theories only. Learning when the given domain is incorrect, inadequate, or intractable will be discussed briefly in the future work section. Notice that inadequate and intractable models can be considered incomplete, since they are in fact missing some aspect of the external world. They are listed separately, however, because they are best addressed with different mechanisms.

A domain theory may be incomplete in several ways:

- Operators may be partially specified—the planner may know only some of their preconditions and some of their consequences.
- Entire operators may be missing—the planner may not know all its capabilities.
- Object types or instances may not appear in the description of the state—knowledge about the objects that must be manipulated may be missing. The operators may not contain enough information about which object types they may be applied to in order to achieve the desired effects.

- Attributes of objects in the world may be unknown—Attributes of objects can be combined to form new attributes. For example, mass and volume define the attribute weight via a formula. The range of values that already known attributes can take may be further specified.

- Factual properties may be missing from the state—the concrete value of an attribute of some object is unknown (e.g., size, color, weight, category...)

Section 3.3.1 contains examples of the first and last cases. As we saw in that section, each case causes a different type of planning failure. This is why each case needs to be addressed differently. Chapter 5 describes methods for detecting different types of failures and how to adjust the domain knowledge in each one of the above cases. There are several ways to detect and refine incomplete knowledge. One is to rely on a human to build the knowledge iteratively by testing it on sample problems and correcting errors by hand. Another is to have the system learn autonomously by interacting with the environment, as the next section describes.

3.4 Learning from the Environment

A planner is a problem-solving engine typically used in applications that involve physical systems. Some examples are:

- Path planning [Brady, 1982], which involves finding a route for a robot controller.

- Process planning [Chang and Wysk, 1985], where the planner is given a specification of a product and finds a sequence of operations to manufacture it.

- Using plans for understanding natural language [Wilensky, 1981], where information about an agent's goals and plans proves to be very useful for interpreting stories.

The resulting plans represent sequences of actions that, once executed, transform the current state of the physical system (also called environment) into a desired state. Thus, the domain knowledge of a planner models the external system in order to reason about its behavior and act accordingly. The operators constitute the planner's knowledge of how to affect its environment. The domain model is a good representation of the external processes if it allows the planner to extract all conclusions that are relevant or necessary for its task. In other words, a good model encompasses what is expected from the external system.

Any disagreement between these expectations and the results of the external processes indicates an imperfection in the model (of some of the types indicated in Section

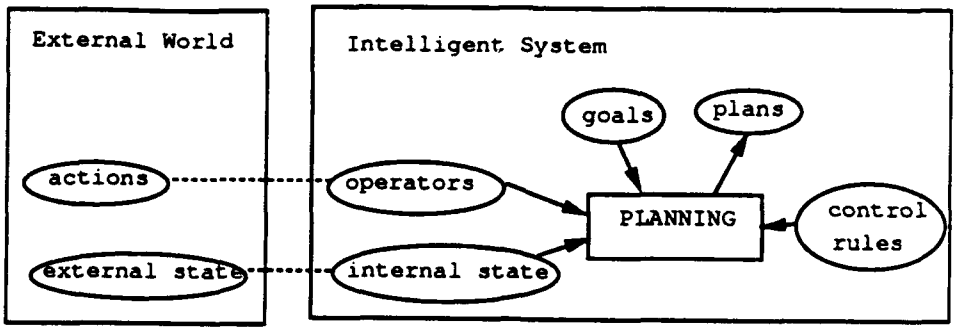


Figure 3.1: An intelligent agent interacts with the world. Operators correspond to actions. The external state is represented by an internal state.

3.3). Any autonomous system must be able to observe its environment and to adjust its internal model when expectation failures occur. Many times it is not clear which fault in the model caused the wrong prediction. It may be necessary to perform a series of directed manipulations of the external system in order to collect more observations related to the failure. These directed manipulations are what we call *experiments*, and their purpose is to gather enough data for the system to update its imperfect model. In summary, observing and manipulating the environment is necessary for this type of learning to occur. These interactions with the environment raise many issues currently under research. This section describes the particular limitations of our system that are directly related to its interaction with the environment.

3.4.1 Interaction with an External Environment

Figure 3.1 shows an intelligent system that has the ability to interact with some external system, also referred to as external world or environment.

Operators are internal models of external actions. Operators are *applied* by updating the *internal state* according to their effects. The action that corresponds to the operator is *executed* always in the *external world.*

> **Definition.** The execution of an action **succeeds** if all the known effects of the corresponding operator happen in the external world when all the preconditions are satisfied.
>
> **Definition.** The execution of an action **fails** if some effects of the corresponding operator did not happen as expected when all the preconditions are satisfied.

When a goal is given to the planner, it designs a plan to achieve that goal. Then the plan is executed step by step. Each step is an operator whose corresponding action must

be executed in the environment. Whenever the system decides to execute an action in the external world it is always the case that the internal state indicates that the corresponding operator is applicable. At this point the system first checks if the preconditions of the operator are indeed satisfied in the external world. If the model is correct then the check will be positive, and the action is executed in the external world. Then the system checks if it has been in fact executed correctly by checking the effects of the operator in the external world. If the model is correct then the execution will be successful; whatever the goal of the system is, it is achieved after the execution of the sequence of actions proposed by the planner.

Notice that in this scheme, the system is not necessarily observing all possibly observable facts about the external state. Its attention is focused only on the facts that are relevant to the application of the action, which are precisely the predicates included in the preconditions and the effects of the corresponding operator.

The system always has some expectations about the world, and they are represented by the internal state. The observations that the system can make correspond to the real state of the external world. In order to know if the model is accurate, the system compares its expectations with its observations. When there is a difference between the system's expectations and its observations, then some fault in the model has been detected and there is opportunity for learning how to correct it.

One possible cause for a difference between expectations and observations is the presence of other agents that can interact with the same environment. If there are other agents, then the cause of the difference might not be a fault in the model. The internal state of the agent is not updated with the effects of actions that are executed by other agents inadvertently. If no cause for the difference is found, the system should consider that some action was executed without its knowledge, and update its internal state accordingly. Another possible source for a difference are nondeterministic environments, in which the outcome of an action under the same circumstances can be different. Noisy sensors can signal unexpected observations that do not correspond to the real external state. This work does not consider any of these possibilities.

The actions that the agent performs are considered to be *independent*. This means that the results of an action can be observed immediately after it is executed and their results do not depend on the actions executed previously. This last assumption simplifies the problem enormously. Fortunately, it holds in most planning domains.

3.4.2 Simulator

For our implementation, we built a simulator of the external environment. The simulator uses a complete and correct set of operators to model the available actions[1], as well as a state to represent the external state. In addition to the domain operators, the simulator is also given operators to simulate failure conditions. So if the preconditions of an operator O are $(p1 \wedge p2 \wedge p3)$, a failure operator can be constructed with the conditions $(\sim p1 \vee \sim p2 \vee \sim p3)$ and the effects to be obtained when one or more conditions are not true. For example, a failure operator would represent the action of opening the door when the door is locked. When an observation is requested from the simulator, it is obtained from the state. When an operator must be executed, the simulator applies to its state the simulator's operators whose conditions match.

In our simulations, the failure operators do not have any effects. In some domains, executing these operators may have spurious effects. For example, consider a drilling operator in the process planning domain. Suppose that the presence of cutting fluid is a necessary condition for drilling, since it absorbs the heat produced by the operation. If that condition is missing from the drilling operator, the failure operator used by the simulator should have the effects that this operation has in the real world, i.e., that the drill bit is damaged by the excess of heat as well as the part.

Our simulator did not represent noise in observations, nor spurious effects that the execution of an erroneous operator may have. This is not a very sophisticated scheme to model the complexity of the real world, but it provides the types of external interactions necessary for experimentation.

3.5 Experimentation

As we saw in the previous section, the interaction with the external world is a powerful tool for acquiring new domain knowledge. The directed manipulation of the environment through experiments makes the learner proactive and reactive in the learning process. This section describes what experiments mean in this thesis, why they facilitate enormously the learning task, and what is involved in the formulation of experiments.

3.5.1 Task-driven Experimentation

In recent years, the topic of experimentation has received significant attention in Artificial Intelligence. The range of concepts embraced by the word "experimentation" is so broad

[1]Notice that neither EXPO nor the planner have access to this complete domain, which is used solely for the simulation.

that it is not possible to give an operational definition that includes them all. Scientists, philosophers, and psychologists have used this term in such diverse contexts that any attempt to reconcile the various perspectives is doomed to failure. Even in the field of Artificial Intelligence there are different ways of understanding the term. Figure 3.2 presents a classification under which the different interpretations of experimentation may be grouped.

The broadest definition of experimentation includes *thought experiments* (also called Gedanken experiments). These include any mental supposition followed by its mental test. For example, we all do this kind of experimentation when trying to solve some problem that requires making suppositions and figuring out what would happen if they were made true. When the test is actually performed in some way, then the experimentation is *active* and usually involves an action in the external world.

Purposeful experimentation can be intentional or curiosity-driven. Many of the actions taken by children at play are of the latter kind, where actions are applied just to see what happens, just to determine their effects. Pure curiosity can lead to the exploration of the consequences of the set of actions available. In this case, surprises can trigger experiments that have some intention by themselves. Another purpose of this kind of experimentation can be to analyze the consequences of certain actions that have shown to be interesting for the system. This means that it will be able to gather knowledge from the experiment that the system may otherwise be missing. Passive observations of the actions performed by another entity could be included in this group.

Task-driven experiments imply deliberately provoking some change in external conditions when an experiment is performed as a means to gather knowledge that is necessary to achieve a previously set goal. The consequences of such deliberate actions are observed and the system corrects its knowledge to adjust it so as to match more closely its environment. The experiments are *directed* to find the knowledge that the system needs to solve the task. **Task-driven experimentation** describes best the work in this thesis, and is highlighted in Figure 3.2.

Confirmation experiments are performed to test the degree of validity of a certain hypothesis. In this case, there is some preconceived knowledge of what the exact consequences might be. If the system can have a range of values that describe the credibility of its knowledge, experimentation can be useful to give the system a more accurate idea of the validity of each belief. Other systems can accept or reject a hypothesis on the basis of a single experiment.

A particular case of confirmation experiments is the *scientific method* (sometimes also called experimental method) in which experiments are designed to test some theory. As Kuhn [Kuhn, 1977] described them, they can either refute or confirm a theory, but never assure its complete validity. We do not relate any of our current research to this definition

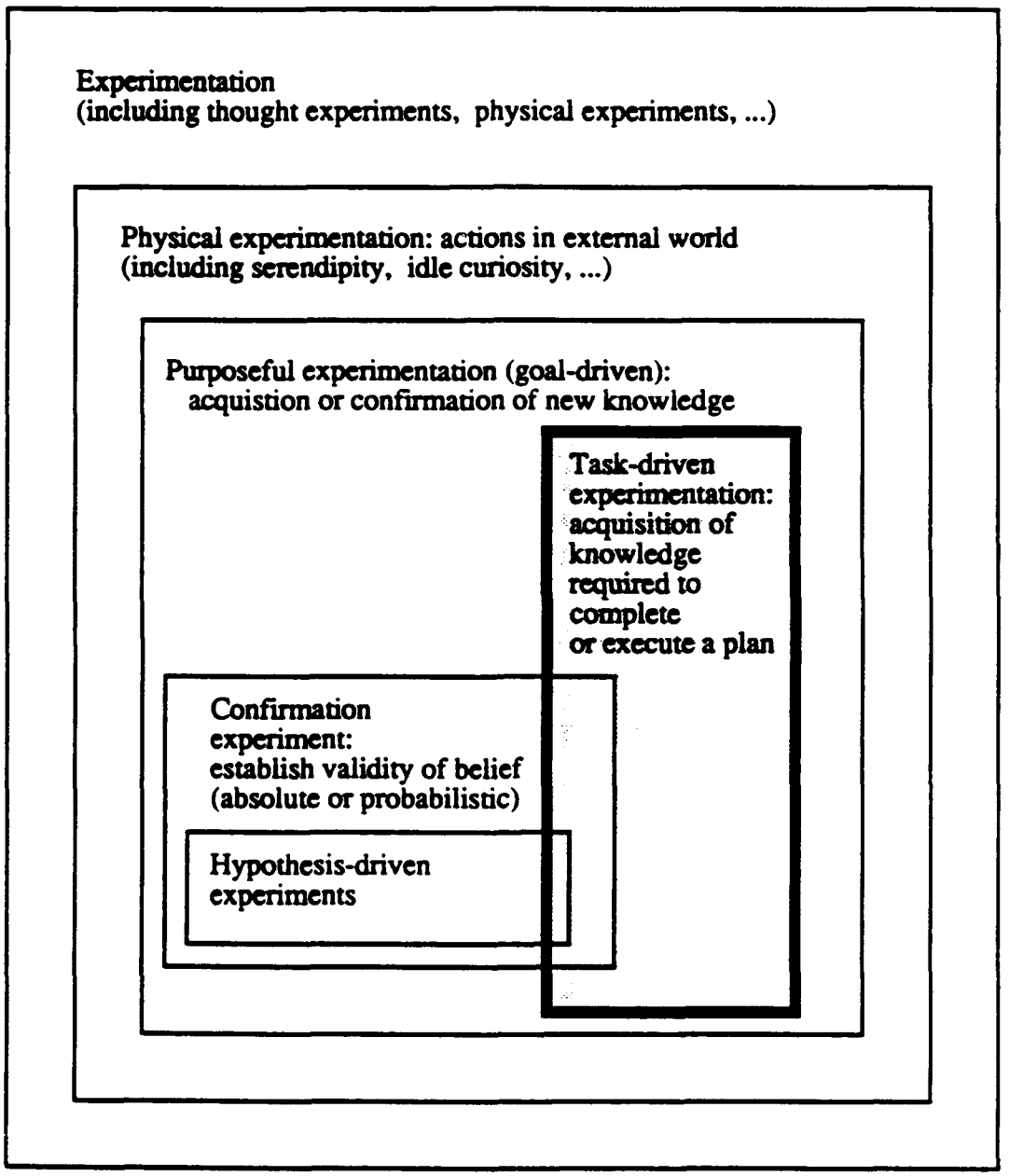


Figure 3.2: What is Experimentation? Our operational definition is task-driven experimentation, where deliberate changes in external conditions are preformed as a means to gather knowledge that is necessary to achieve a previously set goal.

of experimentation. On the contrary, we will explore ways in which experimentation will allow our system to acquire new knowledge, but never with a preconceived theory to be confirmed or refuted by the outcome of the experiments. The word experimentation will be dissociated from the usual interpretation in the context of the scientific method. This does not mean a total separation, however. Many of the early chemistry experiments, for example, lacked theoretical basis.

Our work does not represent an effort to give a solution to the global problem of automating the process of making experiments as a whole. Rather, we focus our attention on a few points of the fairly large space of experimentation. Here, we always refer to experimentation in an active planning context: there is a goal, a state, and a (partially) formulated plan. Experiments are task-driven, always directed at overcoming a current impasse in the planning processes due to a lack of domain knowledge. This means that the description of the world that is learned is one that is useful for solving the problems that the intelligent system must solve. We never learn in this framework any properties of the world irrelevant to the problem-solving task, i.e., we are not modeling idle curiosity. This kind of task-driven experimentation gives the system a context in which to learn and more focused information for the experiments.

3.5.2 Efficient Experimentation

When expectations and observations differ, the system engages in an expensive process of finding what knowledge it is missing that would account for the difference. Experimentation can be described as having the following steps:

1. **Hypothesis formation**: Find possible hypotheses that explain the phenomenon. It is not necessary to enumerate all possibilities, since the system should try first the most plausible ones. Identifying the most plausible hypotheses facilitates the process enormously, but it is also a complicated matter.

2. **Requirements for an Experiment**: Decide what is required to test a given hypothesis. Testing a hypothesis might require several experiments.

3. **Experiment**: Experiments are done in three phases:

   (a) **Design**: In order to obtain the data that the system needs, an experiment must be designed with the appropriate functionality. Experiments are designed following the requirements specified in Step 2, and instantiating any variables that are not constrained by the requirements. If many experiments are possible, one must be chosen. The design phase includes planning to achieve the state where the experiment is to be performed.

(b) **Execution**: Once designed, the experiment can be carried out on the external environment.

(c) **Observation**: After the experiment has been performed, the system obtains feedback from the external world.

4. **Analysis**: When the results of experiments are analyzed the system might have found the information that it sought. If not then it might design and perform more experiments, or go back to the hypothesis formation stage to revise its hypotheses.

5. **Confirmation**: Confirmation experiments may be designed and carried out to corroborate the hypotheses emerging from the results of the experiments just performed.

6. **Acquisition**: Based on the observations, the system might or might not change its current knowledge. Possible changes include correcting what is inaccurate, adding missing information, and confirming existing knowledge.

7. **Recovery**: The state of the world before the experiment was performed might have to be restored. Performing an experiment might have affected the initial set of goals either violating goals (negative interactions) or achieving goals (positive interactions).

The cycle of steps 1 through 4 is repeated until the experiments yield the information sought or the system decides to give up and work on another task.

The requirements $E_{requirements}$ for experiments that result from Step 2 are specified as follows:

- $E_{operator}$: the operator about which the system tries to collect more information.
- $E_{current-state}$: State the system is currently in.
- $E_{exper-state}$: A state in which the experiment is to be performed. It is any state that matches all the preconditions of the operator, plus an additional set of conditions necessary for the experiment (usually related to the hypothesis being tested).
- $E_{observe}$: Observations to be collected before and after the action that corresponds to $E_{operator}$ is executed.

The methods for learning by experimentation in Chapter 5 detect expectation failures, find hypotheses to correct them, and produce $E_{requirements}$. The rest of the experimentation stages are addressed in Chapter 4.

Many hypotheses can be plausible for any given phenomenon. For each hypothesis, we can envision many possible experiments. Each experiment requires, among other things, setting the environment in the appropriate state to perform it. This involves the use of the planner to achieve that state. Many plans may be possible, each involving different resources. Experiment design and execution can be costly. Thus, the use of experimentation requires a framework where the most promising hypotheses and experiments are considered first.

3.6 PRODIGY

The methodology described in this thesis is implemented in an experimentation system called EXPO. EXPO uses PRODIGY [Minton *et al.*, 1989a; Minton *et al.*, 1989b; Carbonell *et al.*, 1991] as the underlying planning system. PRODIGY is a general-purpose problem solver that serves as a testbed for planning and machine learning research. The central problem solver was purposefully designed with a "glass-box" approach: all the steps taken, all the decisions made, and all the information consulted by the engine are available in a problem's trace. This is a very useful feature for any learning system, since there is an information context in which learning can take place. In addition, PRODIGY is a well-developed and thoroughly tested tool.

This section first presents the particular description language that PRODIGY uses to represent domain knowledge. Then it describes briefly other learning methods implemented on PRODIGY to discuss their relationship with EXPO.

3.6.1 PRODIGY's Domain Knowledge

In PRODIGY, the domain knowledge is given by a set of operators and inference rules. The operators are models of the available actions, specifying under which conditions (preconditions) an action has which effects (postconditions). Inference rules are used to deduce additional information from the state. A problem is given by an internal state, representing the current state of the world, and a goal statement. PRODIGY searches for a solution using backward chaining means-ends analysis.

The preconditions of an operator are represented by an expression in a special type of first-order logic called PDL (for PRODIGY's Description Language). PDL allows negation, conjunction, disjunction, and universal and existential quantification. The effects can be primary or conditional (when their application depends on the state in which the operator is applied). Figure 3.3 presents a BNF description for PDL.

```
LOW-LEVEL SYNTAX:
  constant := ATOM
  variable := <ATOM>
  predicate := ATOM
  term := variable | constant | exp
  var-list := (variable variable ...)

SYNTAX FOR FORMULAS:
  exp := atomic-exp | negated-exp | existential-exp | universal-exp |
         conjunctive-exp | disjunctive-exp
  atomic-exp := (predicate term term .....)
  negated-exp := (~ existential-exp) | (~ atomic-exp)
  disjunctive-exp := (OR exp exp exp ....)
  conjunctive-exp := (AND exp exp exp ....)
  existential-exp := (EXISTS var-list generator exp)
  universal-exp := (FORALL var-list SUCH-THAT generator exp)
  generator := atomic-exp

SYNTAX FOR OPERATORS:
  operator-name := ATOM
  simple-effect := (ADD atomic-exp) | (DEL atomic-exp)
  conditional-effect := (IF exp [simple-effect]*)
  effect := simple-effect | conditional-effect
  operator := (operator-name (PRECONDS exp) (EFFECTS ( effect effect ...)))
```

Figure 3.3: PRODIGY's Description Language and Operator's Syntax

Inference rules are used in PRODIGY to deduce additional facts about the current state. While the application of an operator produces a new state, the application of an inference rule augments the facts that are known about the current state. The predicates added by an inference rule are called open world, and are only computed on demand by backward-chaining on the rule. Inference rules, unlike operators, do *not* correspond to any external actions.

PDL allows functions to be part of the preconditions of an operator. Consider, for example, the following operator:

```
(PICKUP-OBJ
    (preconditions
        (and (armempty)
             (next-to ROBOT <obj>)
             (is-object <obj>)
             (weight-of <obj> <weight>)
             (less-than <weight> 10)))
    (effects (
        (del (arm-empty))
```

```
(del (next-to <obj> <*other-obj-1>))
(del (next-to <*other-obj-2> <obj>))
(add (holding <obj>)))))
```

less-than is a function whose two arguments range over the real numbers. It is written as a Lisp function, and it returns true if its first argument is smaller than the second one. The possibility of including functions in the preconditions makes PDL very powerful, since any computable function can be used as a precondition. But this same property makes learning more difficult, as we describe in Section 4.1.

3.6.2 Learning in PRODIGY

Figure 3.4 depicts the different learning modules that have been developed for PRODIGY.

Learning is used to speed up problem solving through the automatic acquisition of episodes useful for analogical reasoning [Veloso, 1992], producing abstraction hierarchies [Knoblock, 1991], and learning control rules [Minton, 1988; Etzioni, 1990; Pérez and Etzioni, 1992]. All these methods are designed to capture control knowledge to guide the search process. The domain knowledge is never changed.

None of these learning methods address the issue of how the domain knowledge is acquired. In PRODIGY learning at the knowledge level is done both from the user through an apprentice-type system [Joseph, 1992] and from the environment through autonomous learning via experimentation (as described in this thesis). The APPRENTICE system provides a user-friendly interface for defining the operators and the problems in a domain.

EXPO is a module that automatically refines a knowledge base by direct interaction with the environment. Given some initial domain knowledge (defined through APPRENTICE or by any other way), EXPO monitors plan execution to detect faults in the operators. Experimentation is used to correct these faults. Learning produces new and improved definitions of the operators. Notice that, unlike APPRENTICE, EXPO does not require interaction with a user, being the only module in PRODIGY that learns new domain knowledge autonomously.

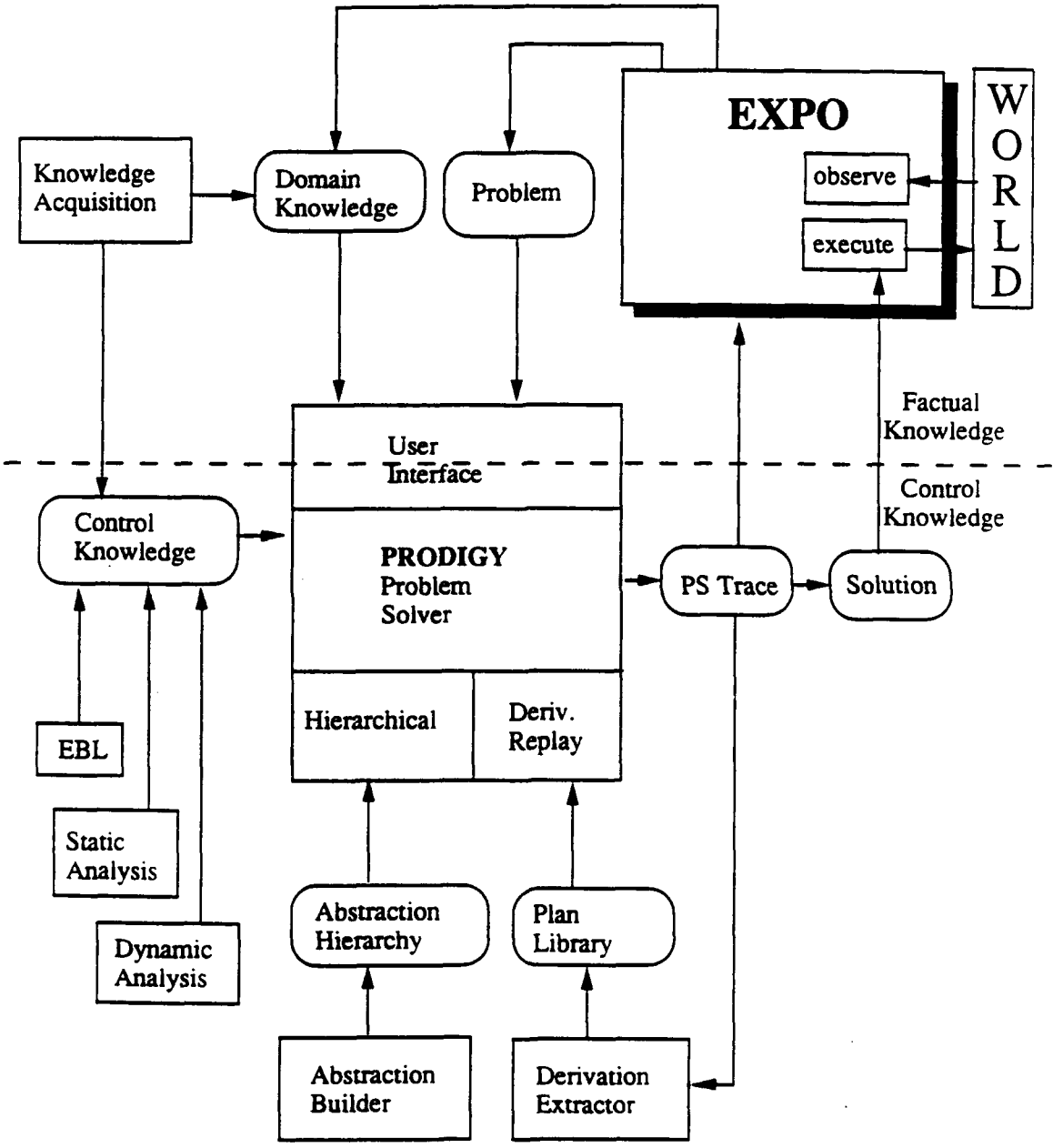


Figure 3.4: A Schematic Representation of PRODIGY. EXPO is the only system that acquires new domain knowledge autonomously.

Chapter 4

The Experimentation Process: Step by Step

This chapter describes how to detect faults in a planner's domain knowledge, and how to design experiments to pinpoint the faults and correct the domain. The experimentation process is described in detail for one particular case: acquiring new preconditions of operators. The chapter presents both general descriptions of the techniques used and their particular implementation in EXPO.

The chapter begins describing a method for detecting operators that are missing some preconditions. Then it shows how to construct hypotheses as a set of predicates representing possible new preconditions of the operator. Section 4.3 describes a set of heuristics that compare the hypotheses and choose the ones most likely to yield the condition missing from the operator. Section 4.4 describes how to design experiments to test each chosen hypothesis. Experiment design is cast as a search for a set of conditions necessary to (dis)confirm the hypothesis, and a plan to bring them about. Many different criteria considered for this design space are described in this section as *policies*. A combination of policies forms a *strategy*, which guides the search to design experiments that meet the desired criteria. This section describes two very different strategies used by EXPO. The chapter continues describing how the experimentation process is carried out until the missing precondition is found, and how problem solving is continued after learning from the experiments. The chapter ends with a discussion on how the techniques described compare with experimentation techniques of other systems.

4.1 Detecting Missing Preconditions

Suppose that a planner is given the incomplete operator from the process planning domain shown in Figure 4.1. This operator models the process of grinding a metallic surface. A grinder holds a part with some holding device, and, using a grinding wheel as a tool, it changes the size of the part along a selected dimension. This representation may seem correct, but in fact the system will find additional facts that are required through its experience. For example, the operator is missing the precondition that the grinder must have cutting fluid. Grinding is an abrasive operation that generates heat as a result of the friction between the tool and the part. If no cutting fluid if present to absorb the heat, then the grinding process will not produce the desired size (the grinder and the part will overheat instead.)

```
(GRIND-INCOMPLETE
 (preconditions
   (and
        (is-a <machine> GRINDER)
        (is-a <tool> GRINDING-WHEEL)
        (is-a <part> PART)
        (holding-tool <machine> <tool>)
        (side-up-for-machining <dim> <side>)
        (holding <machine> <holding-device> <part> <side>)))
 (effects (
        (add (surface-finish <part> <side> SMOOTH))
        (add (size-of <part> <dim> <value>)))))
```

Figure 4.1: An incomplete model of grinding

Suppose that the system is trying to grind a part to make its length smaller. Before grinding the part, the system checks that the preconditions are true in the external world, as shown in Figure 4.2(a). Since the observations confirm the expectations, the system goes ahead and applies the action to try to grind the part. After applying it, the postcondition of GRIND is checked in the external state. The size of the part has changed to be of size k, but the surface finish is not as it was expected, as shown in Figure 4.2(b). This may be because the known effect that specifies the new surface finish is wrong, or because the operator is missing a necessary precondition. We consider the later hypothesis first. that some unknown precondition is not true in the state and thus the grinding action is not working as the given operator specifies.

How could we find out what the missing precondition is? We can try to find out what conditions were true in an earlier successful application of the operator that are not true now. Figure 4.2(c) shows a previous successful situation when the grinder had fluid and

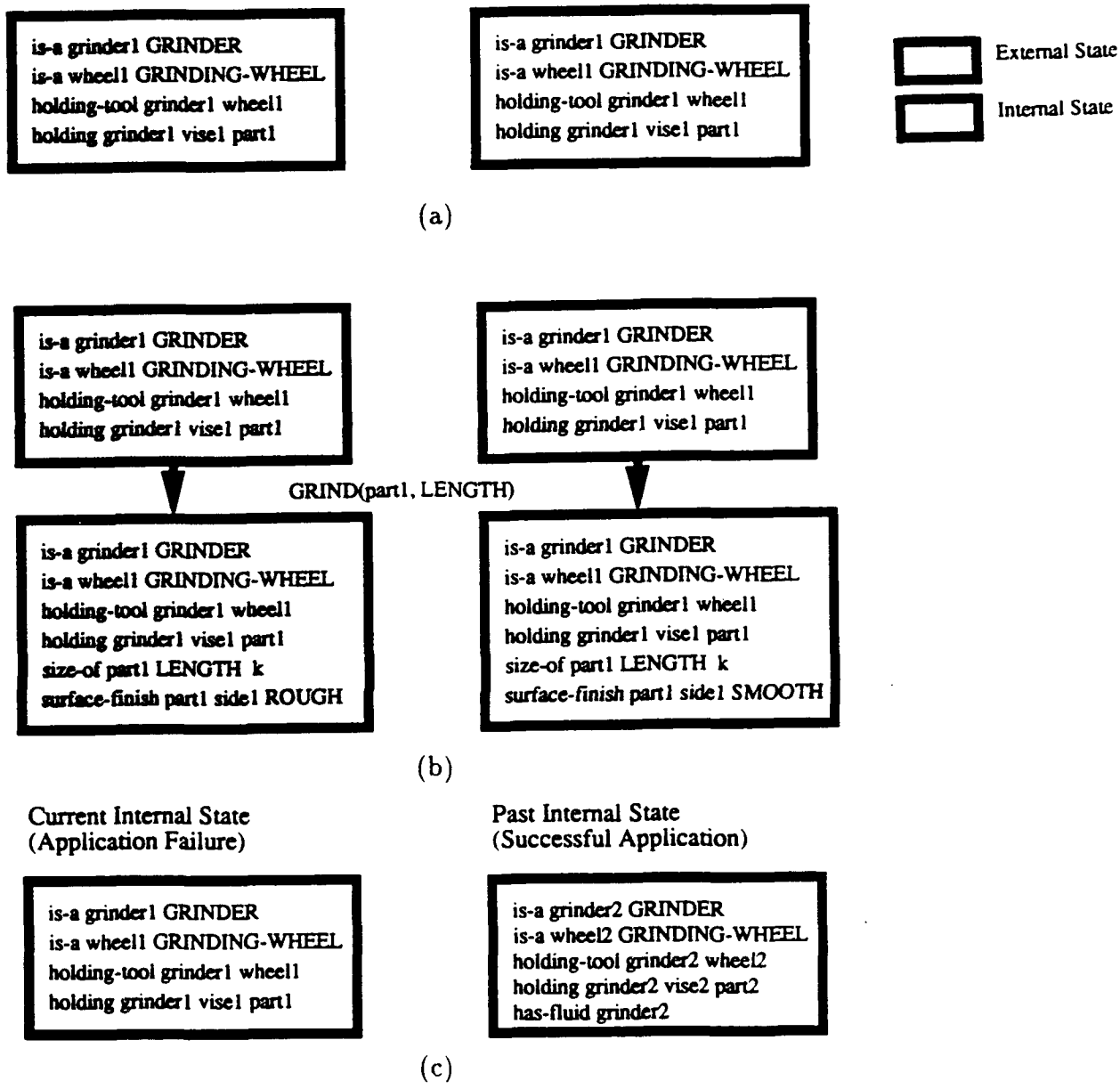


Figure 4.2: Finding new preconditions of grinding

the operation worked. The system now puts fluid in the grinder, and tries again to apply the operator. Now the action is successfully applied, and the operator is corrected.

But in the general case, there can be several differences between the state in which the operator is applied successfully and the state in which a failure happens. Then, experimentation is needed to determine which one of the differences is relevant for this

particular failure. The method for learning new preconditions is summarized in Table 4.1. Notice that $\Delta(S_{old}, S_{current})$ contains the following two sets of predicates: (1) predicates in S_{old} that are not in $S_{current}$, and (2) the negation of predicates in $S_{current}$ that are not in S_{old}. So this method accounts for learning of positive as well as negative preconditions, depending on which subset of the differences contains the relevant condition.

If after manipulating the world the effects of the operator O are not true, then hypothesize that a precondition of the operator is missing.

1. Select candidate preconditions. The candidate set $\Delta(S_{old}, S_{current})$ is formed by calculating all the differences between the most similar earlier state in the previous problem solving history in which O was applied successfully (S_{old}) and the current state $S_{current}$ (an unsuccessful application of O).
2. Identify missing precondition. Formulate experiments observing if the operator is successfully applied when one of the differences P is true in the state. Use any information available to formulate the most promising experiments first. In absence of knowledge, apply a divide and conquer strategy to isolate the precondition from $\Delta(S_{old}, S_{current})$.
3. Add P as a new precondition of operator O.

Table 4.1: Method for learning new preconditions. When the effects of an operator do not occur in the external world, a previous successful application of the operator is used to find a missing condition of the operator.

This set of hypotheses does not necessarily contain the relevant condition as it may not be represented as a single atomic observable expression. Other possible hypotheses to be considered as candidate conditions are:

- Disjunctive expressions of predicates
- Inferred predicates deduced in a state by theorem proving
- Quantified expressions of some predicates
- Predicates that are never observed because they are not needed for planning (i.e., the weight of a box)
- A functional relation of several predicate arguments

So if the cause of the failure is not found after experimenting with $\Delta(S_{old}, S_{current})$, then these additional hypotheses must be considered. EXPO does not expand the hypotheses further, and it confines the experiments to $\Delta(S_{old}, S_{current})$. When it runs out of hypotheses, it gives up learning and continues plan execution.

4.2 Constructing the Set of Hypotheses

As we described in the previous section, if there are several differences between the success state and the failure state, experimentation is needed to find the relevant condition for the failure.

Here is a typical set obtained by EXPO. In this case, GRIND(grinder1, wheel1, vise1, part7, TOP) is successful but GRIND(grinder1, wheel1, vise1, part3, TOP) fails:

```
(size-of <part> WIDTH 3)
(size-of <part> LENGTH 7)
(size-of <part> HEIGHT 2.5)
(material-of <part> BRASS)
(has-fluid <machine>)
(surface-finish part26 <side> SAWCUT)
(holding drill1 vise2 part26 <side>)
(material-of part26 STEEL)
(is-a drill1 DRILL)
(is-a drill-bit1 DRILL-BIT)
(material-of part37 COPPER)
(has-hole part37 <side>)
```

The problem can now be specified as follows:

Given: an operator OP that has an incomplete set of preconditions
a set of predicates *Candidates* that contains a precondition that OP is missing

Find: which predicate in *Candidates* is the missing precondition of OP

If all the predicates in *Candidates* are equally likely as possible new conditions, a divide and conquer strategy through the set *Candidates* is the most appropriate experimentation strategy. The algorithm is described in Table 4.2. Notice that if the cardinality of *Candidates* is n, this algorithm requires $log(n)$ experiments. Furthermore, each experiment has a large set of requirements. Besides $Preconditions(OP)$, the first experiment requires $n/2$ predicates to be satisfied, the second requires $n/4$, and so on until there is only one predicate left (a total of $2n - 1$ predicates). The algorithm always requires $log(n)$ experiments and a total of $2n - 1$ predicates to achieve. The planner has to build

a plan to set the environment in a state that satisfies that many predicates. Apart from the planning effort involved, the execution of those plans raises non-trivial issues. Plan execution may use up valuable resources (including time), produce non-desirable changes in the environment that are hard to undo, and interfere with the main goals of the system's task. For all these reasons, it is important to minimize the number of experiments and their requirements.

Divide_and_Conquer_Experimentation(*OP*, *Candidate*)

1. $New\_Candidates \leftarrow \{\ \}$
2. Divide *Candidates* into two subsets of equal cardinality: $Candidates_A$ and $Candidates_B$.
3. Prepare experiment: achieve a state where $Preconditions(OP) \wedge Candidates_A$ are satisfied.
4. Experiment: execute *OP*.
5. If execution is successful, then $New\_Candidates \leftarrow Candidates_A$ else $New\_Candidates \leftarrow Candidates_B$
6. If $Cardinality(New\_Candidates) = 1$, then return *New_Candidates* else Divide_and_Conquer(*OP*, *New_Candidates*).

Table 4.2: Algorithm for divide and conquer experimentation. The algorithm divides the set of candidates into two subsets of the same size, and uses an experiment to find out which subset contains the missing precondition, then the process is iterated on the subset until its size is one. The algorithm always requires $log(n)$ experiments and a total of $2n - 1$ predicates to achieve.

Another consideration is that the set of hypotheses constructed contains many candidates that may not be worth exploring unless everything else fails. In the hair dryer example of Section 1.1 some of the initial candidate hypotheses were the time of the day and the day of the week. In the set of hypotheses above for the GRIND operator, bogus hypotheses include "GRIND fails if there is a part made of steel" and "GRIND fails if there is a part that has a hole". Additionally, if the operator is missing *more* than one condition, the algorithm will fail. The divide and conquer algorithm is very simple to implement, but is definitely far from satisfactory. If any information is available to determine a smaller subset of *Candidates* as more relevant, the experimentation effort may be greatly reduced. In particular, if we could devise a way of ranking the predicates in *Candidates* from most relevant to least, then each candidate could be tested individually.

Such an informed algorithm is shown in Table 4.3. The number of experiments required is inversely proportional to the competence of the ranking procedure. And, most importantly, only one predicate needs to be satisfied in each experiment (apart from *Preconditions*(OP)). On average, $n/2$ experiments are needed In the worst case n experiments are needed each involving also 1 top level goal.

Informed_Experimentation(OP, *Ranked_Candidates*)

1. *Current_Candidate* $\leftarrow$ *Pop*(*Ranked_Candidates*)
2. Prepare experiment: achieve a state that satisfies *Preconditions*(OP) $\wedge$ *Current_Candidate*.
3. Experiment: execute OP.
4. If execution is successful then return *Current_Candidate* else return Informed_Experimentation(OP, *Ranked_Candidates*).

Table 4.3: Algorithm for informed experimentation. The candidates most likely to be relevant are ranked higher. In average, $n/2$ experiments are needed (n in the worst case) and each involves 1 top level goal.

Many systems discussed in Chapter 2 use causal theories or other types of background knowledge to build explanations that lead to the causes of the failure. EXPO relies exclusively on the knowledge given initially for planning. This means that the learning occurs even when no causal, structural, or common sense knowledge (other than the one embedded in the domain model) is available. This is a major advantage, since we do not need to address in turn the acquisition and refinement of that additional and necessarily complex background knowledge.

In summary, any information that may be used to rank the hypotheses greatly reduces the experimentation effort. EXPO's approach is to use heuristics that extract any such information strictly from the domain knowledge given to the planner. The heuristics for choosing hypotheses presented in the next section are a step in this direction.

4.3 Choosing Hypotheses: Finding Relevant Conditions for Failure

This section presents different ways to exploit knowledge about the planning task to evaluate which predicates in a set of differences are more likely to have caused the failure.

The section begins by describing three heuristics to choose hypotheses. Then, their implementation in EXPO follows. Section 4.6 presents a discussion of these heuristics. Their evaluation is presented in Chapter 6 together with other empirical results for EXPO.

4.3.1 Locality of Actions

The first heuristic is the *locality of actions*. The preconditions and effects of actions are concentrated locally, usually affecting the objects under direct influence of the action. In our example we are grinding part7. The fact that this part is made of BRASS may be relevant to the failure obtained. However, it is probably not important that part37 is made of COPPER. This means that we can select the predicates in the set related to objects that the operator GRIND refers to directly.

This locality heuristic is implemented considering only the predicates in the state that contain any of the objects included in the bindings of the parameters of the operator. In our example, if we extract the predicates that include any of {grinder1, wheel1, vise1, part7, TOP} we obtain the following subset:

```
(size-of <part> WIDTH 3)
(size-of <part> LENGTH 7)
(size-of <part> HEIGHT 2.5)
(material-of <part> BRASS)
(has-fluid <machine>)
(surface-finish part26 <side> SAWCUT)
(has-hole part37 <side>)
(holding drill1 vise2 part37 <side>)
```

Notice that with this heuristic we eliminated from the list many predicates that were in fact irrelevant for grinding. For example, many facts about parts not being ground have disappeared.

This heuristic is not helpful when the set of variables that appears in an operator is incomplete. If the operator for grinding lacks any predicates that have to do with the tool being used, the system would never learn that the tool is important for the action. A possible way around this problem is to give some structured knowledge to the state. For example, to have information in the state about where everything is, and what things are close to each other. In this work, we avoid this kind of approach because it requires adding to the system knowledge that is not strictly required for planning.

Another problem is that this heuristic does not always propose relevant differences. Consider the subset of differences just obtained. Because grinding is being done to part7, all the facts about part7 could be relevant. But since the TOP is the side being

ground, any facts that have to do with TOP are also considered relevant. This includes for example the fact that part37 has a hole on the TOP, which is not relevant to the application of the operator.

4.3.2 Generalization of Experience

Another helpful heuristic is *generalization of past experience.* Generalizing successful situations tells us what predicates appear in all success states. This summary of past experience helps us to locate relevant causes of failures.

This heuristic is implemented by generalizing successful situations through the bindings of the operator. This gives us the set of predicates that have appeared in all of them. After removing from that set the predicates that correspond to the preconditions of the operator, we obtain the following set:

```
(material-of <part> BRASS)
(surface-finish <part> <side> SAWCUT)
(has-fluid <machine>)
```

Notice that this set is much smaller than the one in the previous section, where we only considered a single success situation. When the system encounters more successful situations, then the set of differences becomes smaller.

If the system has no previous experience with the application of the operator this generalization strategy is not helpful. This strategy also fails when not much generalization can be extracted from successful applications.

A generalization of all the possible situations where grinding is successfully applied is exactly the correct precondition expression sought. The preconditions of an operator express the sufficient conditions for applying the operator, and represent the class of states in which the operator is applicable. Thus, learning the precondition expression of an operator is a problem of concept learning. The initial precondition expression of an operator is the initial description of the concept. Each successful execution of an action is a positive example of th ? concept, and each failure a negative example. Experimentation is an additional source of examples, and it provides the learner with the ability to design instances and direct the learning.

However, this concept learning is simpler due to common simplifying idealized assumptions of planning tasks. There are no misclassified examples. The effects of actions can be observed immediately after execution. The observations are collected through noise-free sensors. Under these assumptions, our classification of execution success and failure never produces noisy data. As far as the language used for expressing the concepts, the large majority of the precondition expressions in operators are conjunctions

of predicates (or negations of predicates). This is because actions are easier to express if their effects under different conditions are described in separate operators. Disjunctions can be (and are) expressed explicitly in different operators. In this sense, limiting learning to conjunctive expressions is still useful.

4.3.3 The Structure of Domain Knowledge

Operators for a single task are often closely related to one another. Some operators are inverses, i.e., they undo each other's effects. Some operators have similar effects, but are applied under different conditions. Both of these relations appear in the machining domain. There are operators for holding a part with a certain holding device, and there are operators to release the part from the device. There are operators for holding a tool in a machine, and operators for releasing tools from machines. The operators for drilling are all similar to one another. So are the operators for polishing surfaces. These relations of similarity and reversibility constitute the heuristic of *structural regularity* of the domain.

Structural similarity will help us identify what hypotheses are more plausible by looking at similar operators to the operator being considered. This is a very general idea, and it can be used for learning new preconditions, as described next.

One way to implement this heuristic is to organize the operators in a hierarchy, so that similar operators can be easily located. The hierarchy can be built through comparing the preconditions and effects of operators. In our machining domain, part of the hierarchy that includes the grinding operation is shown in Figure 4.3.

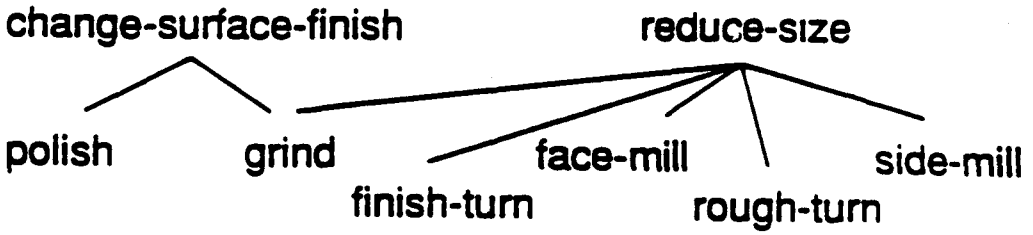


Figure 4.3: Part of the operator hierarchy in the process planning domain.

Consider the set of differences obtained in the previous section as possible candidates for a new precondition of grinding. Many other operators change the size of a part. Many

of them require the use of cutting fluid, which is in fact the relevant condition for this particular failure.[1] Only some of them have conditions about the material of the part. And none of them has any conditions about the surface finish of a side of the part. The heuristic suggests that the differences should be considered in the following order:

1. `(has-fluid <machine>)`
2. `(material-of <part> BRASS)`
3. `(surface-finish <part> <side> SAWCUT)`

This heuristic is not very helpful if there are no similar operators or if there are similar operators but they are also incomplete.

4.3.4 Implementation

This section describes in detail the algorithms that implement in EXPO the heuristics just described.

Each execution of the operator is either a success or a failure. As Section 3.2 described, the precondition expression of an operator can be seen as a concept that represents the states in which the operator can be executed successfully. A state in which a successful execution occurs corresponds to a positive instance of the concept, and a state in which a failure is obtained is a negative instance. Each constant in the instances must be parameterized according to the bindings of the operator. For example, if the variable `<part>` is bound to part1 when we execute GRIND, and the state contains `(material-of part1 BRASS)` we would like the concept to contain a more generalized version of this fact, i.e., `(material-of <part> BRASS)`. EXPO keeps information about action executions in *situations*, which are composed of:

- **Operator**: the operator whose action was executed.
- **Result**: the result of the execution, i.e., success or failure.
- **Bindings**: the list of bindings for the operator variables.
- **State**: the list of predicates believed to be true immediately prior to the operator being executed.

[1] Cutting fluids cool both the cutting edges of the tool and the part, aid in chip clearance, and improve the surface finish. Notice that a great deal of background information would be needed to explain that the presence of cutting fluid is important for grinding.

To generalize from experience, EXPO applies the algorithm presented in Table 4.4. Given two situations, their generalization is a new situation generated as follows. First, the corresponding bindings are generalized. Then the literals in the state are changed substituting the constants and variables according to the new bindings. The state of the generalization includes only the predicates that appear in the generalized states of both situations.

Generalization(S_1, S_2)

- $S_{new}.Operator \leftarrow S_1.Operator$
- Generate $S_{new}.Bindings$ from $S_1.Bindings$ and $S_2.Bindings$
  - the generalization of a variable and a constant is the variable itself.
  - the generalization of two different constants is a variable.
  - the generalization of two equal constants is the constant itself.
- Generate A by substituting the constants and variables of $S_1.Bindings$ that appear in $S_1.State$ by the bindings in $S_{new}.Bindings$
- Generate B by substituting the constants and variables of $S_2.Bindings$ that appear in $S_2.State$ by the bindings in $S_{new}.Bindings$
- $S_{new}.State \leftarrow A \cap B$
- Return S_{new}

Table 4.4: Algorithm for generalizing two successful situations.

Notice that this generalization algorithm is biased to produce conjunctive descriptions of the concept. This bias is appropriate for this application. The large majority of the precondition expressions in operators are conjunctions of predicates (or negations of predicates). This is because actions are easier to express if their effects under different conditions are described in separate operators. In this sense, even if the system aims to learn only conjunctive expressions of predicates it would be a great win. In fact, even though PRODIGY allows for a very expressive language in the preconditions, the generalization only contains the predicates in the preconditions that are part of the conjunct. For example, if the precondition expression of an operator is `(and (A B C D (or E F)))` E and F are never included in the generalization.

EXPO maintains the current description of each operator's preconditions as a version space [Mitchell, 1978]. A version space is defined within a lattice of concept expressions

that are ordered from more general to more specific. Concept instances are the most specific expressions. Successively more general descriptions are found at higher positions in the lattice. A version space is defined by two boundary sets: a set of maximally specific descriptions (S) and a set of minimally specific descriptions (G). A version space is maintained for each operator. The examples correspond to situations in which the system tried to apply the operator. Recall that successful situations are positive instances, and failure situations are counterexamples or negative instances. Given two situations S_1 and S_2, S_1 is *more general than* S_2 if both of the following hold:

- Each literal in the state of S_1 has a corresponding literal in the state of S_2. The correspondence is done through the bindings of both situations.
- The bindings of S_1 are more general. A variable is more general than a constant. If two constants are equal, the generalization is the constant itself.

The G set, the most general description, is initialized to the initial preconditions of the operator and its value is kept to the current preconditions. The S set is updated as new success situations are obtained using the generalization algorithm just described. When a new failure situation is obtained, the S set is updated by removing from it any conjuncts that also appear in the failure state. Because G is always the current preconditions of the operator, G always covers failure situations and must be specialized. Instead of following the usual procedure for updating G (which is highly inefficient when there are many possible new conditions), EXPO waits until the missing condition is found through experimentation, and then adds it to the current conjunct in the G set.

The version spaces implement the heuristic for selecting hypotheses based on its generalization of experience. From the set of current candidate hypotheses, only the ones that appear in S (the ones that are common to all successful situations) and do not appear in G (since G contains the preconditions, they appear in the failure state) are selected.

The set of hypotheses selected by the generalization heuristic is then filtered by the locality heuristic. This heuristic selects only the hypotheses that contain constants and variables that appear in the bindings of the failure situation. This new subset of the hypotheses is then ranked by the heuristic of structural similarity as we explain now.

All the domain operators are organized by EXPO in a hierarchy using a simple clustering algorithm described in Table 4.5. The top node contains all the operators in the hierarchy. For every node, the operators that are not in any of its children yet are examined to build a child node. The expression or expressions[2] that appear in a larger number

[2]preconditions, postconditions, or both. In our experience with EXPO's domains, this does not make a difference in the effectiveness of the structural similarity heuristic.

of operators define the child node, and the operators that contain them are transferred to it. The algorithm works its way down in the tree until a node is reached that contains only one operator or all of its operators expressions are included in the node. When a new condition or effect for a operator is learned, the hierarchy is updated by recomputing the children of the node that contains the node operator.

Build_Operator_Hierarchy (*Operators*)

1. For each $OP \in Operators$ do
   $Expr(OP) \leftarrow$ expressions in the preconditions and effects of OP.
2. $Open \leftarrow \{\}$.
3. $Node.Ops \leftarrow Operators$
4. $Open \leftarrow Node$
5. Repeat
   - $Node \leftarrow Pop(Open)$
   - $Node.Subtypes \leftarrow$ Produce_Subtypes($Node$)
   - $Node.Ops \leftarrow Node.Subtypes$
   - $Push(Open, Node.Subtypes)$

   Until $Null(Open)$

Produce_Subtypes(Node)

1. Repeat
   (a) Find the set of expressions E that are most common for operators in $Node.Ops$.
   (b) Make a subtype node with all the operators in $Node.Ops$ that have all the expressions in E and remove them from $Node.Ops$.

   Until $Ops\_in\_Subtypes = Node.Ops$.

Table 4.5: Algorithm for building an operator hierarchy.

EXPO considers first the hypotheses that are selected by the three heuristics. Then, it considers the ones that the structural regularity rejected, then the ones rejected by the

locality heuristic. Last, EXPO considers the rest of the hypotheses in the initial set.

Determining the missing precondition is done through iterative experimentation with the ranked list of candidate predicates. In EXPO, this process converges if the missing condition is an observable and non-inferred predicate that is within a conjunctive expression. If this is the case, the missing condition is included in the group of candidate hypotheses, and EXPO eventually encounters it and learns it through experimentation.

Although the algorithms presented in this section can be made more sophisticated, we must keep in mind that they are used to build heuristics. In their simplicity, the results in Chapter 6 show that they are effective for implementing these heuristics.

4.4 The Experimentation Search Space

The previous section described how to compare hypotheses heuristically to evaluate which ones are more promising. Once a particular hypothesis is chosen, an experiment must be designed to test it. In our particular example, the heuristics suggest that the most promising hypothesis is that the precondition that the operator GRIND is missing is `(has-fluid <machine>)`.

In order to perform an experiment, the world must be brought to a state where the conditions of the experiment are satisfied. In our example, we must reach a state where the current known preconditions of GRIND and the hypothesized new condition are satisfied. In other words, our goal is to reach a state where the following are true:

```
(exists (<machine> <tool> <part> <dim> <side> <holding-device>)
   (and
        (is-a <machine> GRINDER)
        (is-a <tool> GRINDING-WHEEL)
        (is-a <part> PART)
        (holding-tool <machine> <tool>)
        (side-up-for-machining <dim> <side>)
        (holding <machine> <holding-device> <part> <side>)
        (has-fluid <machine>)))
```

The planner must first come up with a plan to achieve this state from its current state, which is the state in which the failure occured that triggered experimentation. We call this search process *pre-experiment planning.*

Once the pre-experiment plan is executed, the experiment can be carried out. In our example, we GRIND and check if this time the effects specified for GRIND are obtained. If not, other hypotheses must be tested with other experiments. But if grinding works now, then the missing condition must be `(has-fluid <machine>)`. The new condition is

added to the operator GRIND. Then, the original plan that failed must be continued in order to achieve the original goal. If the pre-experiment plan has undone any of the facts necessary for the original plan, then a *post-experiment plan* is needed to restore those facts and continue with the main plan. Whether a post-experiment plan is used to enable the continuation of the original plan or replanning is done to achieve the original goals is not the issue here. The issue is that there is some effort needed to restore facts that were undone during pre-experiment planning and we call that post-experiment-planning.

Clearly, some pre-experiment plans are better than others. Minimal interference with the main plan is important. In our example, it would be better to use another holding device for the experiments since vise1 is already holding part1. So maybe using grinder2, wheel2, and vise2 is better. But perhaps it is more important to make the pre-experiment plan as short as possible, so we can recover from the failure and go on with our main plan. If this is the case, maybe using grinder1, wheel1, and vise1 is better since they are already set up and ready for grinding operation. So, one experiment may be better than another one, depending on what policy is preferred.

EXPO designs experiments following a set of policies chosen by the user from a pool. Each policy defines a preference to be used for decision making and can be thought of as a piece of control knowledge to be used during experimentation planning. Policies are grouped together to define strategies. We describe now EXPO's policies and strategies in detail.

4.4.1 Experiment Policies

The experiment policies described in this section are grouped under four topics: search depth and plan length, goal interactions, operator properties, and binding interactions. They are summarized in Figure 4.4. Notice that all the policies described in this section are domain independent.

Search Depth and Plan Length

Limiting the search depth helps control the search time. Limiting the plan length helps control the execution time.

Each level of a search involves the application of an operator or an inference rule. An inference rule represents a deduction from the current state, whereas an operator represents an externally executable action. The final plan is composed only of actions. This is why the depth of the search does not correspond to the length of the plan, although they are related.

- **Search depth and plan length**
  - Avoid deep nodes
  - Prefer shallow nodes
  - Avoid long plans
  - Prefer short plans
  - Avoid plans with too many state changes
  - Prefer plans with fewer state changes
- **Goal interactions**
  - Support main goal concord
  - Avoid main goal protection violation
  - Avoid main prerequisite violation
- **Operator properties**
  - Avoid irreversible operators
  - Prefer reversible operators
  - Prefer operators that minimize state changes
  - Prefer more reliable operators
  - Avoid unreliable operators
- **Binding interactions**
  - Avoid objects of very high protection
  - Prefer objects of lower degree of protection
  - Prefer least number of protected objects

Figure 4.4: EXPO's experimentation policies.

EXPO's available policies that concern experimentation search depth and plan length are:

- **Avoid deep nodes**: Never expand nodes below a certain depth. This maximum depth for the experimentation search must be given a value.
- **Prefer shallow nodes**: Prefer expanding shallower nodes.
- **Avoid long plans**: Never choose plans that are longer than a given length.
- **Prefer short plans**: Prefer plans that are shorter.

- **Avoid plans with too many state changes**: Never choose plans that cause changes in the external world over a given number. The amount of changes that a plan produces in the sum of the effects of the operators that compose it.

- **Prefer plans with fewer state changes**: Prefer plans that cause a smaller amount of changes in the external world.

Goal Interactions

The goal interaction policies refer to the interactions between the goals in the experimentation space and goals in the main search space. They are different from the types of interactions within a search space, as in [Sussman, 1975; Sacerdoti, 1977], where for example goal G_1 may be preferred to another goal G_2 if achieving G_1 first causes G_2 to undo G_1. Here, a search path is preferred over another one it minimizes negative interference (or maximizes positive interference) with the top level goals. Notice that the preference is over which search paths to pursue, not over which goals.

EXPO's policies for interactions with the main goals are:

- **Support main goal concord**: If a search path achieves a goal that remains to be achieved by the main plan, prefer it over other paths.

- **Avoid main goal protection violation**: If a search path clobbers a goal previously achieved by the main plan that is still needed to achieve the main goals, then prefer other search paths over this one.

- **Avoid main prerequisite violation**: If a search path undoes a fact that the remaining main plan requires to be true, then prefer other search paths to this one.

Operator Properties

Local decisions about which operator to prefer in order to achieve a goal may be based on properties of the candidate operators. Some properties may be domain dependent, such as the execution time of the operator or other resources involved (see Section 4.4.2 for more details on domain-dependent policies). These are EXPO's policies based on domain-independent properties of operators:

- **Avoid irreversible operators**: Never use irreversible operators. Determining that an operator is irreversible requires proving that there is no plan that can undo its effects. This is at least undecidable, since planning is undecidable [Chapman,

1987]. Also, the irreversibility of operators is not a binary feature: the same operator may be irreversible in some states and reversible in others. Because of these and other issues that make the automatic determination of irreversibility very complex, EXPO relies on a user-defined classification of operator's reversibility.

- **Prefer easily reversible operators**: If the effects of operator O_1 are easier to undo than the effects of operator O_2, prefer O_1 over O_2. Determining the degree of reversibility of an operator is not a simple matter, so EXPO relies on an ordered list of operators defined by the user.

- **Prefer operators that minimize state changes**: If an operator O_1 has less effects than operator O_2, prefer O_1 over O_2. This policy is a more local version of the policy to prefer plans with fewer state changes.

- **Prefer more reliable operators**: If an operator O_1 has a higher rate of success to number of times that it has been used than operator O_2, then prefer O_1 over O_2. This policy avoids obtaining execution failures during the experiments.

- **Avoid unreliable operators**: If an operator's rate of failure to number of times that it has been used is over a user-defined threshold, do not use it.

Binding Interactions

During planning, the variables of each operator are given values by binding them to objects in the current state. Some bindings may be preferred to others. For example, we may prefer to use in the experiments a different machine than the one that is being used in the main plan, since the machine used in the main plan is probably all set up for the operation. Other objects may not bring up such preferences. For example, if a brush is being used in the main plan to clean the metal burrs in the part we may not mind using it if needed during the experiment planning. In summary, there may be different binding preferences for different types of objects.

One interesting case in the process planning domain is the type part. Suppose that the main goal is to drill a hole of a certain width and depth in part1. Now suppose that the drilling operation fails because of a missing precondition, and experiments with the drilling operator are needed. If the experiments are done drilling part1, we may not interfere with the main goal, but we would violate an implied goal: "Do not drill other holes in the part other than the ones specified in the goal". In fact, when we specify a goal to the planner in this domain (and many others) many such explicit goals are also desired but too complex to specify. A planner works by default on building a plan to achieve each of its given goals, so by default it would not interfere with the implied goals. But since the experimentation process requires producing plans for other goals, such implicit

goals may be violated by default. Notice that since the implicit goals are not declared in the goal set of the main problem, they are not protected by the goal interaction policies. We have addressed this problem through binding preferences as follows.

When a domain is defined, each type of object is assigned to one of the following classes:

- *Very high protection*: The instances of these types that are being used in the main plan are never to be used for the experiments.
- *High protection*: During experiment planning, other instances are preferred to instances of these types that are being used for the main plan.
- *Low protection*: During experimentation planning, other instances are preferred to instances of these types that are being used for the main plan, but never prefer instances of high or very high protection.
- *Very low protection*: The instances of these types can be used any time during experiment planning.

In the robot planning domain there are only four types of objects, classified as follows:

- High protection: boxes
- Low protection: doors, keys
- Very low protection: rooms

The process planning domain is more complex, and has 33 types of objects, classified as follows:

- Very high protection: parts
- High protection: holding devices
- Low protection: machines, machine tools, objects consumed during an operation.
- Very low protection: objects not consumed during an operation.

If necessary, the number of degrees of protection may be augmented, but the mechanism would be the same.

Once the protection classes have been defined, they are used to determine the policies that EXPO can use for choosing bindings. They are the following:

- **Avoid objects of very high protection**: Never use objects that are used in the main plan and whose type is classified as very high protection.
- **Prefer objects of lower degree of protection**: If two objects used in the main plan are being considered for binding the same variable, prefer the object with a lower degree of protection.
- **Prefer least number of protected objects**: If several objects used in the main plan are being considered for binding different variables, prefer the set of objects that minimizes the total degree of protection.

In some domains, it may be desirable to have a policy to prefer bindings that were used previously in successful executions of the operator. For example, in the process planning domain it is preferable to use a tool that has worked previously with any materials, than to use a tool that has not worked in the past for certain types of materials. This policy is not implemented in EXPO. Since it is a policy that applies to the main planning process as well, as we explain next.

4.4.2 Universal Policies

All the policies that the user may define for the main planning task are also applicable to experiment planning. These policies correspond to the control knowledge (be it domain independent or not) given to the planner to be used for decision making in the domain. They can be considered *universal policies*, since they apply in both the main and the experiment search spaces. For example, we would consider an experiment that uses cheaper materials than another one to be better. But the same principle applies to any two plans. The quality of the experiment plans is determined in many dimensions by these policies that are to be addressed by other more specific work in plan quality.

Experiment policies and universal policies may be in conflict. When this is the case, EXPO gives priority to universal policies.

4.4.3 Experimentation Strategies

The experiment policies described in the previous section express different concerns that an experimenter may consider to design and choose experiments. Some of these policies may be conflicting, but the experimenter must have some overall, global strategy that determines which policies serve the strategy best.

In EXPO, many different strategies may be designed. In this section, we describe two strategies that illustrate the capabilities of EXPO in this respect. The two strategies lie in opposite sides of the spectrum:

- **The learner-at-heart strategy**. The main concern in this strategy is to acquire new knowledge, and as such novel sutuations are preferred over ones already experienced, and short experiment plans are preferred over longer ones that may delay learning.
- **The problem-solver-at-heart strategy**. The main concern of this strategy is to acquire new knowledge in order to solve the problem at hand. Consequently, interactions with the main plan are avoided when possible, and repeating proven solutions is preferred over trying new ones.

The learner-at-heart strategy is implemented using the following policies:

- Avoid deep nodes
- Prefer shallow nodes
- Avoid long plans
- Prefer short plans
- Prefer unreliable operators

The problem-solver-at-heart strategy is implemented using the following policies:

- Support main goal concord
- Avoid main goal protection violation
- Avoid main prerequisite violation
- Avoid irreversible operators
- Prefer reversible operators
- Prefer more reliable operators
- Avoid unreliable operators
- Prefer plans with fewer state changes
- Avoid plans with too many state changes
- Prefer operators that minimize state changes
- Avoid objects of very high protection
- Prefer objects of lower degree of protection
- Prefer least number of protected objects

4.4.4 Implementation

Each policy is implemented in EXPO as a control rule for PRODIGY. We summarize now briefly their syntax and semantics; more details can be found in [Minton *et al.*, 1989b].

PRODIGY considers four choice points during the search process: which node to expand, which goal to achieve, which operator to use to achieve a goal, and which bindings to use to instantiate the variables of an operator. For each type of decision, PRODIGY makes a choice using a set of heuristic rules that recommend one candidate over another one, to select a candidate and not consider any others, and to reject a candidate to be never considered again for this decision point. The left-hand side of each control rule expresses the criteria upon which the recommendation is based. These criteria are described in terms of the planner's meta state (the current goal, the current state, etc) and expressed as a special type of predicate called a *meta predicate*.

Appendix C contains all the policies that are defined in EXPO as control rules for PRODIGY. This way of implementing the policies is very flexible. Any new policies can be easily added as new control rules. Any new strategies can be easily defined by choosing a set of control rules. At the same time, the current implementation of policies as control rules can greatly be improved. The control rules in PRODIGY 2.0 have limited capabilities. For example, good policies that cannot be expressed are policies that would suspend a search path until a later point. Also, there is no framework in PRODIGY at present to shift attention to different goals (in our case hypotheses) changing the definition of the problem, although some efforts within the project were in this direction [Kuokka, 1990]. EXPO can benefit greatly of current ongoing research on control mechanisms for PRODIGY.

4.5 Experiment Execution, Learning, and Recovery

After calibrating and prioritizing the set of hypotheses with its heuristics, EXPO tests one hypothesis after another until it finds the one which is the missing condition of the operator. For each hypothesis, EXPO designs a pre-experiment plan as the previous section described. Then, the plan is executed to reach a state where the experiment can be carried out. If any other failures are obtained during the execution, EXPO stores them and comes back to learn from them after the cause of the current failure under study is determined.

If the missing precondition is found, EXPO adds it immediately to the operator's precondition expression. The new operator is used in any future planning. If none of the hypotheses is found to be the missing condition, EXPO notifies the user that it believes the operator's preconditions to be incomplete but that it cannot find the missing

condition. Section 5.2 described the types of missing conditions that cannot be learned by EXPO.

Since the missing precondition was the cause of the failure obtained in the main plan, its acquisition allows the planner to overcome that failure. Now the execution of the main plan may be continued. However, the experiments execution brought about many changes in the external state since the time when the main plan was designed and it may now be invalid. EXPO replans to achieve the top level goals from the current state of the world. Then, EXPO continues with the execution of this new plan and continues to watch for failures that signal faults in the domain knowledge that it can correct by experimentation.

4.6 Discussion

TEIREISIAS [Davis, 1976] is a knowledge acquisition system with a similar technique to EXPO's structural similarity heuristic. TEIREISIAS used a simple clustering algorithm to discover similarity between rules. When the user entered a new rule that was clustered together with other rules, TEIREISIAS checked that the new rule had the same predicates in the left-hand side. If any was missing, TEIREISIAS would warn the user that it believed that predicate should be mentioned n the rule. EXPO uses this structural knowledge to refine rules not when they are defined, but when they are found to be faulty. Also, EXPO uses the heuristic to discriminate among a set of hypotheses, which TEIREISIAS never produced.

As described in Chapter 2, the COAST system [Rajamoney, 1988] has several criteria for choosing experiments: preferring experiments whose observations can be collected easier, preferring experiments that are guaranteed to disprove some hypothesis, and changing the current state to enable experiments with different observations. In EXPO's implementation all the cost of collecting any observation is considered the same, but if this were not the case COAST's first strategy would be helpful. EXPO does have the other two strategies, since every experiment proves or disproves a hypothesis and every experiment causes changes to the external world.

KEKADA [Kulkarni, 1988] (described in detail in Section 2.1.2) contains many heuristics for guiding experimentation in scientific discovery. Although EXPO's experimentation is geared to do more mundane learning, it is worth comparing both systems. Most of the heuristics lead KEKADA to behavior that is similar to that of EXPO. Some of the heuristicts are hard coded in EXPO (PC0, PC1, PC4, PC5, HG3, HG8, HSC1, HSC2, ES4, PG1, and DM8), others are expressed as strategies (PC3, PC7, EP6, HM4, HM5, DM1, DM2, DM3, DM5, DM6, and DM7). EXPO could be expanded with some of KEKADA's heuristics. PC2, PC6, and PC8 implement a task-handling mechanism that

EXPO does not have. HG1, EP1, and ES3 provide KEKADA with class generalization, which EXPO does not currently have. EP7, ES1, and ES2 have an exploratory flavor, and as such are not appropriate for EXPO's task-driven learning. KEKADA has a mechanism for switching from one hypothesis to another based on confidence factors (CF3, CF4, CF5, and DM4). EXPO sticks to one hypothesis until it is proven or ruled out, but intelligent switch of attention would make EXPO more flexible.

Chapter 5

Methods for Learning by Experimentation

The previous chapter showed how a difference between the system's expectations and the collected observations indicates a fault in the domain model. Differences are opportunities for learning, so our system must be able to identify them, hypothesize which part of the domain is incomplete, determine the particular fact that is missing in the domain model, and correct it accordingly. All these steps are different depending on the types of failures, and the previous chapter described how to determine that a failure is caused by an operator's incomplete preconditions. We present in this chapter a collection of methods for learning under different types of failures. This collection is not exhaustive, but it is indicative of how experimentation can be used to learn new knowledge from the environment.

The chapter begins with a taxonomy of the types of the facts that may be missing when the domain knowledge is incomplete, which is used as a guideline for the presentation of the methods in the rest of the chapter.

5.1 Refining Incomplete Domain Knowledge

Section 3.3.5 described different types of incompleteness in a planner's domain knowledge. Figure 5.1 summarizes them and describes every type in more detail. All these facts may be acquired by experimentation. The methods in this chapter describe how it can be done for some of the cases, which are highlighted in the figure.

In the first case, an existing operator can be missing either a condition or an effect. The condition may be a predicate or a negated predicate. Also, it can be a simple

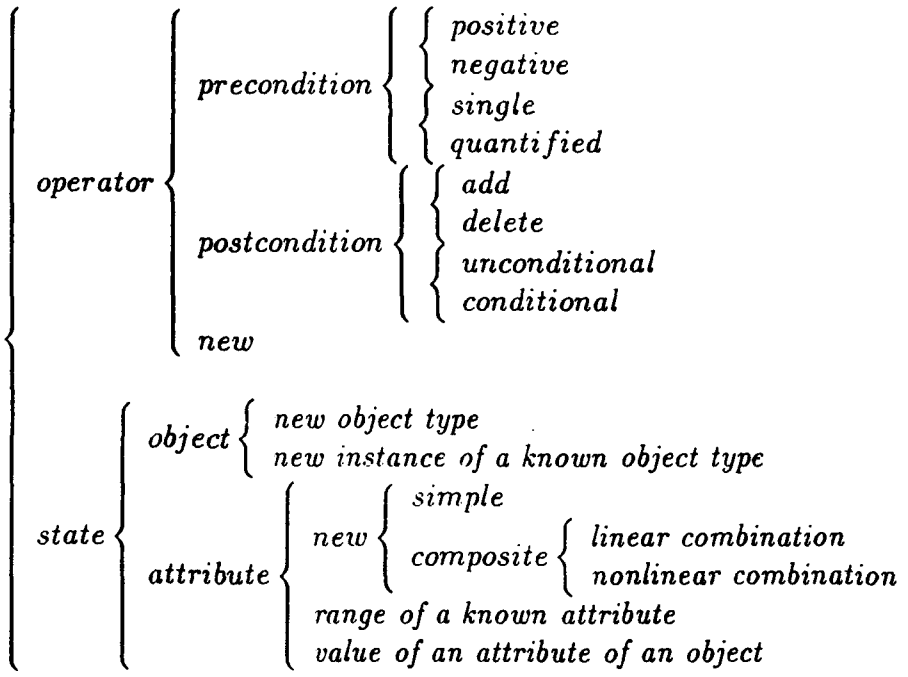


Figure 5.1: Domain knowledge that can be theoretically acquired by experimentation. EXPO concentrates on operator refinement.

predicate or one with a quantification over some of its variables. A missing effect can be either in the add list or in the delete list. In either case, it may be unconditional or context dependent (i.e., when it occurs only under certain state conditions). Section 5.2 describes a method that can be applied to acquire new conditions and effects of operators.

Entire operators may also be missing. If this is the case, several methods can be applied that form an initial definition for the operator based on existing ones. This is done by direct analogy, or by decomposition in a subsequence of operators, or by splitting existing operators under different conditions. There is also the possiblity of probing the environment by trying out the available actions under new conditions. These methods are described in detail in Section 5.3.

The operators can be incomplete, but the state may also be missing many types of knowledge. Certain types of objects may be unknown. New instances of objects types may be encountered by the system. Attributes of objects may be missing. New attributes can be discovered, either in isolation or as combinations of other attributes. The range of a known attribute may also be determined through interactions with the environment. Finally, the value of an object's attribute may be found through experimentation. This

last case is addressed in Section 5.4.

The methods presented in this chapter and their implementation in EXPO are summarized in Figure 5.12.

5.2 More on Operator Refinement

Section 4.1 described a method for learning missing preconditions of operators. But an operator can also have an incomplete set of effects. Consider again the GRIND operator shown in Figure 4.1. The operator is missing an effect: that grinding uses up cutting fluid, so the machine does not have cutting fluid any longer. It is also missing information about the surface finish of the part after grinding. As it turns out, depending on the coarseness of the grit of the wheel the finish is either rough or smooth. We show in this section how these facts can be learned.

The method for acquiring missing preconditions and effects will be referred to as the *Operator Refinement Method* (ORM).

5.2.1 Learning New Postconditions

Our model is still missing the fact that a grinding operation uses up the cutting fluid. We show now how this new effect can be learned.

Suppose that our goal now is to grind part1 so that it is smaller in height and width. This involves two successive applications of the operator GRIND, one for each dimension, as shown in Figure 5.2. For the first grinding operation, our system would check that all the preconditions of GRIND are true in the external world. Since this is the case, it continues planning by applying the operator. Then it checks that the postconditions of GRIND are true in the external state. Notice that because the system doesn't know that the grinder uses up the fluid the internal state reflects this fact by containing `(has-fluid grinder3)` after GRIND is applied. In the real world, the fluid has disappeared, but the system is not yet aware of that fact.

Before the system tries to grind for the second time, it checks if the preconditions are true in the external world. It is at this point that it finds out that the grinder has no fluid. The only action that was performed since the fluid was last checked has been grinding. The system then concludes that one of the effects of grinding is consuming the fluid in the machine, and so it modifies the delete list of the operator grinding.

But in the general case, several operators could have been applied since the fluid in the grinder was last checked. In that case, experiments are needed in order to determine

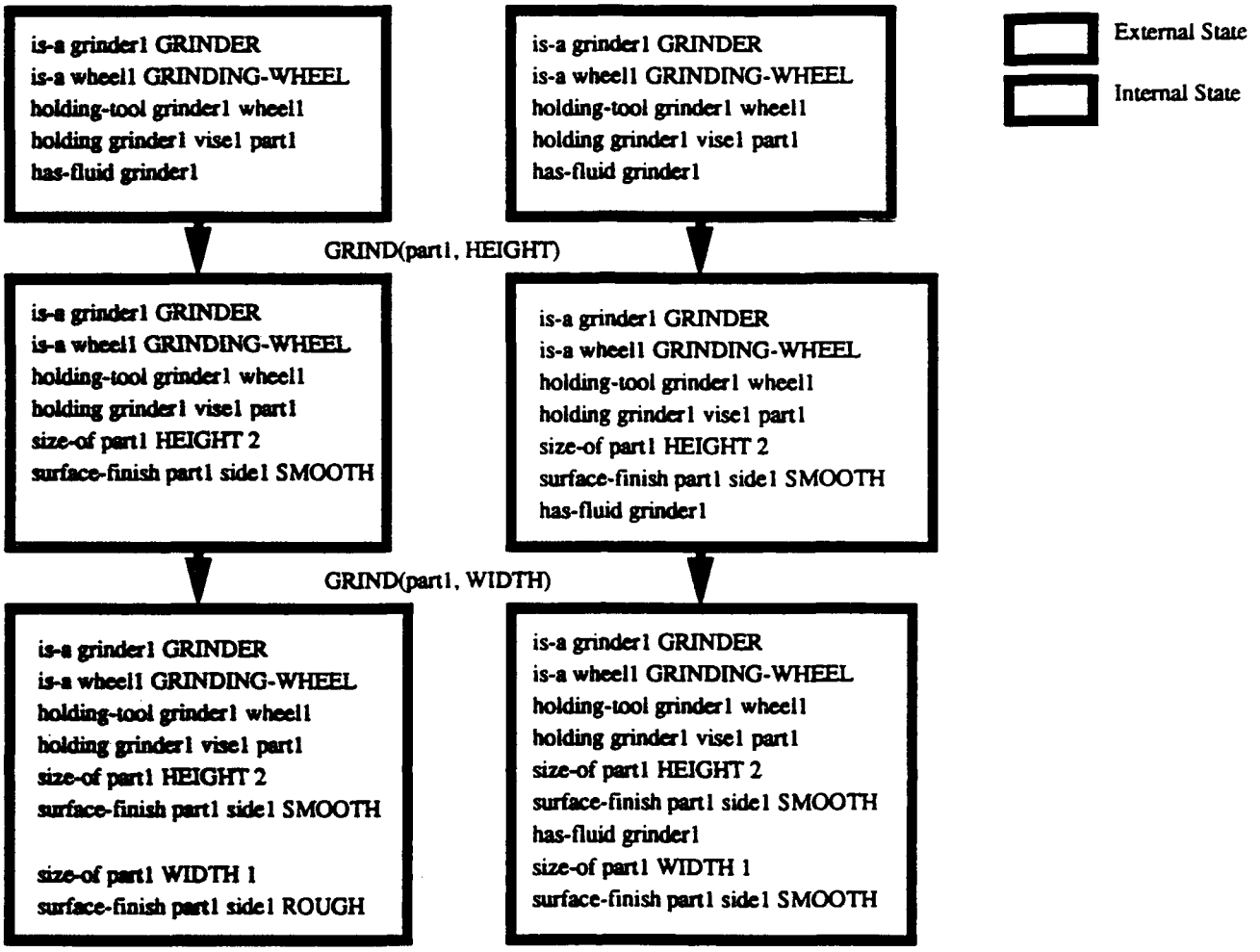


Figure 5.2: Finding new postconditions of grinding

which of those operators is missing a postcondition that specifies the deletion of the predicate has-fluid from the state. The method is summarized in Table 5.1.

This example shows how to learn from failure but the same method can be used for learning from unexpected successes. Notice that delete effects can also be learned from this method, when the condition P is a negated predicate.

The heuristics in Section 4.3 were described for choosing hypotheses in the case of missing preconditions, but they can also be used for learning new effects. Their use and implementation is different. The first heuristic applied is locality. EXPO looks at the bindings of the candidate operators and selects operators that affected the objects in E. Notice that if the effect has any wildcard variables, E's objects do not appear in the bindings of the candidate operators, and so this heuristic is not very helpful. Next,

If a precondition P is true in the model, but it is not true in the external world, then one of the operators applied after P was established in the model has a previously unknown postcondition affecting P.

1. Select candidate operators. The candidate set consists of all operators $\{O_1, O_2, \ldots O_n\}$ applied since the P was last checked.
2. Identify incomplete operator. Formulate experiments over the candidate set. In each experiment, after an operator is applied P is checked in the external world. If as a result of an experiment with operator O_i, P is unexpectedly changed in the world, then O_i is incompletely specified.
3. Add P as a new postcondition of operator O_i.

Table 5.1: Method for learning new postconditions

the structural similarity heuristic is applied. EXPO looks in the operator hierarchy for operators that have the effect, and looks in neighboring nodes for the operator in the list of candidates, and ranks them according to their distance. When EXPO finds the incomplete operator O, then it can use the generalization heuristic. It cannot be used before because EXPO has focused attention and only observes known effects after the execution of an operator. So E was never observed in previous executions of O. EXPO starts monitoring E and generalizes according to the observations collected. Through the generalization, the objects in E may be kept constant, generalized to operator variables, or generalized to wildcard variables.

The method described in this section is limited to observe only the known conditions and effects of each operator. It is possible to learn new effects more quickly if a larger set of predicates is observed after the execution of an action. This would detect changes in the state immediately after the execution. However, limited observation capabilities is a more realistic setting in domains where large collections of data may be observed, and it is the one chosen for EXPO.

5.2.2 Learning Conditional Effects

Learning conditional effects is a mixture of learning new preconditions and new postconditions. But it requires that the system keeps additional information about the actions.

Suppose that the agent's goal is to grind two parts. Grinding part3 changes the surface condition just as the system expects and is shown in Figure 5.3(a). Now it is trying to grind part4. After executing the action, the effects of the operator are checked.

At this point, the system finds out that the postcondition that specifies that grinding makes the surface condition of the part be smooth does not always occur, as shown in Figure 5.3(b). The system would then detect the presence of a conditional effect. Now it will compare the state in which the effect happened and the state in which it did not happen. The only difference in this case is the grit of the wheel, so it will add that as the condition of the conditional effect. Another conditional effect can be learned to account for the situation in which the surface finish produced by grinding is not smooth. Again, if there are several differences between the states then experimentation would be needed to determine the relevant one.

The method is summarized in Table 5.2. This example illustrates that the system will sometimes encounter situations with a great potential for learning. In this case, it can also learn about the conditional effect in case of using a wheel with coarse grit, which is to produce a rough finish. Because the conditional expression associated with an effect is a concept to be learned, it presents similar problems to precondition learning with respect to the set of hypotheses.

If an effect of an operator takes place in situation S_A but not in situation S_B, then it is a conditional effect of the operator.

1. Select candidate conditions for the effect. The candidate set $\Delta(S_A, S_B)$ is formed by calculating all the differences between S_A (the state in which the effect occurs) and S_B (the state in which the effect does not occur).
2. Identify missing condition. Formulate experiments observing if the effect of the operator occurs when one of the differences P is true in the state. Use any information available to formulate the most promising experiments first. In absence of knowledge, apply a binary search to isolate the precondition from $\Delta(S_A, S_B)$.
3. Add P as a condition for the conditional effect of the operator O.

Table 5.2: Method for learning new conditional effects

Let us return for a moment to our example. Let us go back to the point when the system encountered the situation in Figure 5.3(a). When the situation in Figure 5.3(b) is found, and the postcondition does not occur, then the system must have a way to retract its knowledge restricting the effect with a condition that it learns applying the method for learning conditional effects. This example illustrates how the methods presented here are not completely independent. A framework must be devised to allow the system to combine them and apply whichever one seems more appropriate at each time as we will discuss in Chapter 7.

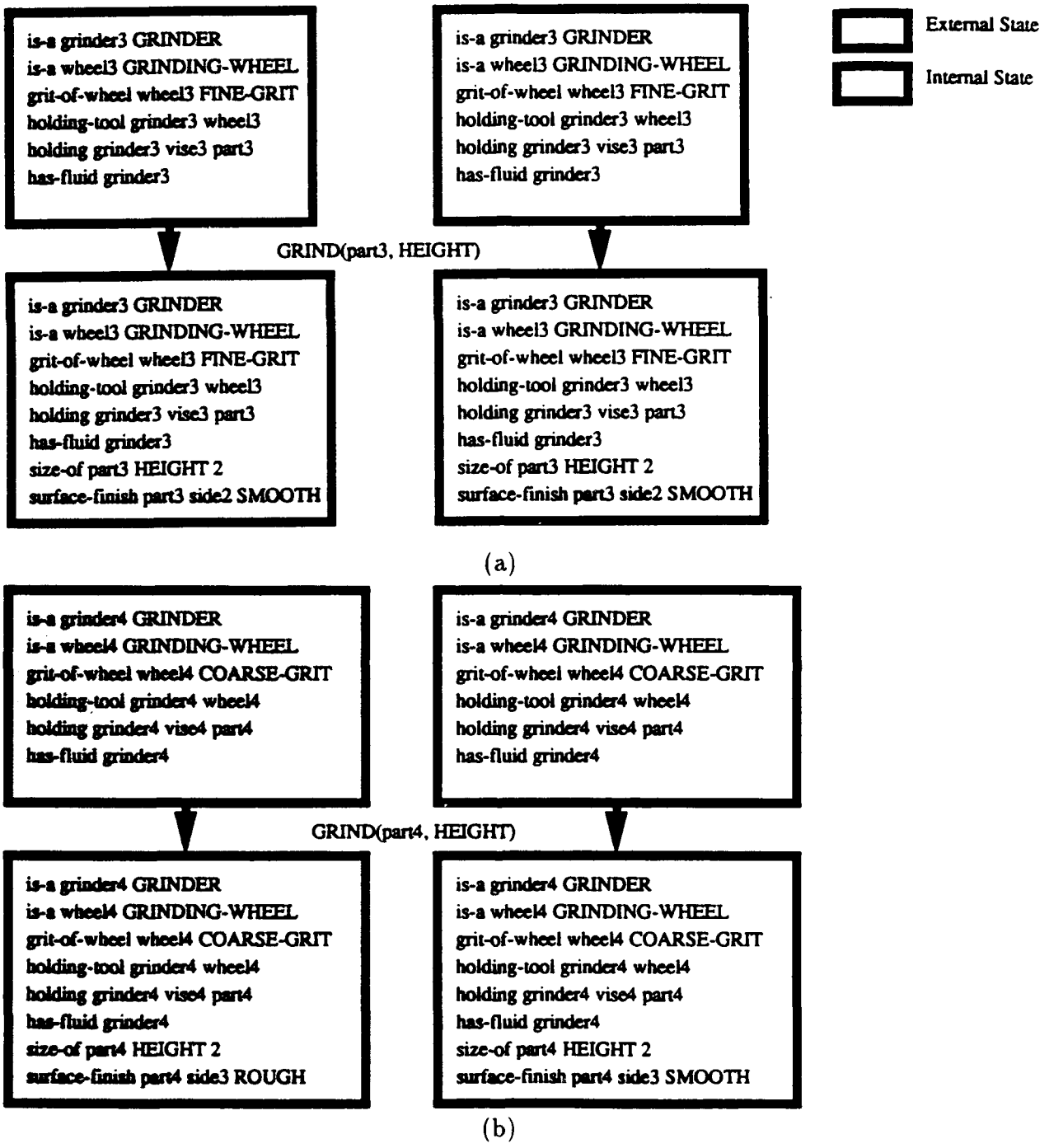


Figure 5.3: Finding conditional effects of grinding

5.3 Learning New Operators

There are many ways to learn new operators. Our methods are goal-directed: they are triggered when the planner finds itself in a situation where it cannot solve a problem.

The system assumes that the available knowledge is incomplete and tries the various methods to formulate new operators. Learning is always incremental, preferring overly incomplete specifications (that are progressively refined by the ORM) to more detailed specifications that may be incorrect. None of the methods is guaranteed to work, only the external execution of the new operators can show if the newly acquired operator has a meaning in the domain. In this section, we describe through examples different ways of learning new operators followed by a more formal description of each method.

5.3.1 Direct Analogy

New operators can be learned by direct analogy with existing ones. As an example, suppose that the system has the knowledge about drilling holes shown in Figure 5.4(a). A hole can be made if a drill has a high-helix drill bit of the size of the desired hole and some cutting fluid, and if it is holding a part that has a spot hole in the appropriate location. Suppose now that the system is given the goal of producing a part with a hole in it, and there are no high-helix drill bits available. The preconditions of the operator for drilling cannot be achieved, and PRODIGY is not able to solve the problem. But instead of returning a failure, our system uses the following reasoning to derive a new operator for drilling with other types of drill bits that might be available. The system finds that both high-helix and twist drill bits are of the same object type: DRILL-BIT, and thus it creates the new operator shown in plain font in Figure 5.4(b). The new operator only gets from the original one the types of the objects that it is applied to, and the effect that it is created for. Experiments are performed by executing the action under different conditions until a successful application is found. We describe in the next paragraph how the experiments can be designed efficiently. If the new operator cannot be applied successfully, then the process is repeated with other types of drill bits. If this does not yield any success either, then other object types are tried. In this case, a new operator for drilling holes with a milling machine is acquired when a different type of machine is considered. These experiments end when a successful application of a newly formulated operator is found that proves its existence. Once this happens, the ORM helps to locate additional conditions and effects that are specific to the new operator. They are shown with a star (*) in Figure 5.4(b). The method is summarized in Table 5.3. Notice that the power of this method comes from the possibility of relating P to P' through the object type hierarchy.

Choosing the right experiments is an important issue for making learning efficient. The conditions for the experiments are guided by the preconditions and effects of the original operator. If there are several operators for drilling that are available, then experiments that involve the preconditions and postconditions common to all drilling operations are preferred. The more available operators that already contain information about

```
(DRILL-WITH-HIGH-HELIX-DRILL
 (preconditions
   (and (is-a <machine> DRILL)
        (is-a <drill-bit> HIGH-HELIX-DRILL-BIT)
        (same <drill-bit-diameter> <hole-diameter>)
        (diameter-of-drill-bit <drill-bit> <drill-bit-diameter>)
        (has-fluid <machine> <fluid> <part>)
        (has-spot <part> <hole> <side> <loc-x> <loc-y>)
        (holding-tool <machine> <drill-bit>)
        (holding <machine> <holding-device> <part> <side>)))
   (effects (
        (del (is-clean <part>))
        (add (has-burrs <part>))
        (del (has-spot <part> <hole> <side> <loc-x> <loc-y>))
        (add (has-hole <part> <hole> <side> <hole-depth>
                       <hole-diameter> <loc-x> <loc-y>)))))
```

(a) An operator for drilling a hole using a high-helix drill bit

```
(DRILL-WITH-TWIST-DRILL
 (preconditions
   (and
        (is-a <machine> DRILL)
        (is-a <drill-bit> TWIST-DRILL-BIT)
     *  (same <drill-bit-diameter> <hole-diameter>)
     *  (diameter-of-drill-bit <drill-bit> <drill-bit-diameter>)
     *  (has-spot <part> <hole> <side> <loc-x> <loc-y>)
     *  (holding-tool <machine> <drill-bit>)
     *  (holding <machine> <holding-device> <part> <side>)))
   (effects (
     *  (del (is-clean <part>))
     *  (add (has-burrs <part>))
     *  (del (has-spot <part> <hole> <side> <loc-x> <loc-y>))
        (add (has-hole <part> <hole> <side> <hole-depth>
                       <hole-diameter> <loc-x> <loc-y>)))))
```

(b) New operator for drilling with a twist drill bit. The stars indicate new facts acquired by the Operator Refinement Method for the new operator.

Figure 5.4: Learning a new operator for drilling by analogy with an existing one.

drilling, the more efficient the experiments designed to refine the new operator. Notice that these are heuristics and they do not make any guarantees about the convergence of the process.

If a given problem cannot be solved by a set of operators because a precondition P that specifies the type of an object of an operator O cannot be achieved, formulate a new operator by direct analogy with O through P.

1. Find a related predicate. Look through the type hierarchy of the objects in the domain and find P' such that it refers to objects of the same type of the unachievable precondition P.
2. Formulate a new operator. Construct a new operator O' with the effects of O that the original problem subgoaled on and all the object types of O except P.
3. Experiment with the new operator. Execute the action. If the desired effects are not obtained, apply experimentation to isolate which of the other preconditions of O need to be added to O'. If O' is applied successfully in some state, then continue with step 4. Otherwise, go back to step 1, either looking for a different P' or considering a different P.
4. Refine the new operator. Apply the ORM to find all the preconditions and additional effects of the new operator.

Table 5.3: Method for learning a new operator by direct analogy with an existing one.

5.3.2 Micro-operator Formation

New operators can also be acquired by learning useful partial specifications of an existing one. One possible way to do this is when the system encounters situations in which only some of the effects of the action are desired. If this is the case then experimentation is used to find if only some of the preconditions are required for the partial effects needed.

Suppose the system has the operator for cutting specified in Figure 5.5(a). The operator expresses that if a circular saw has a type of attachment called friction saw and some cutting fluid and if it is holding a part, then the size of the part can be reduced and the resulting surface is smooth. Now suppose that the system is given a problem whose goal is to make the size of a part smaller, and that no fluids are available in the initial state. The goal cannot be achieved with the available knowledge, and yet there is a way to solve the problem. The system formulates a new cutting operator that has only the effects that it needs from the original one, and only the preconditions that specify the type of the objects required for the operator. The action is then executed. If the desired effect is not obtained, then the system finds which additional conditions are required. This is done by experimenting with the action applying it under different situations. The experiments are guided by the preconditions of the known operator for cutting.

This process ends when a successful application of the new operator is found (thereby proving its existence). This happens when the desired effect is obtained in a state where not all the preconditions of the original operator are true. Finally, the ORM is called to further refine the operator. The result is a cutting operator without the preconditions and effects that have to do with obtaining a reasonable surface condition quality (having fluid on the machine), as shown in Figure 5.5(b). This method for learning a *partial* operator is summarized in Table 5.4.

```
(CUT-WITH-CIRCULAR-FRICTION-SAW
 (params (<machine> <part> <attachment> <holding-device> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> CIRCULAR-SAW)
   (is-a <attachment> FRICTION-SAW)
   (has-fluid <machine> <fluid> <part>)
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <attachment>)
   (holding <machine> <holding-device> <part> <side>)))
 (effects (
   (del (has-fluid <machine> <fluid> <part>))
   (add (surface-finish-side <part> <side> SMOOTH))
   (add (size-of <part> <dim> <value>)))))
```

(a) An operator for cutting

```
(CUT-TO-SIZE
 (params (<machine> <part> <attachment> <holding-device> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> CIRCULAR-SAW)
   (is-a <attachment> FRICTION-SAW)
*  (size-of <part> <dim> <value-old>)
*  (smaller <value> <value-old>)
*  (side-up-for-machining <dim> <side>)
*  (holding-tool <machine> <attachment>)
*  (holding <machine> <holding-device> <part> <side>)))
 (effects (
   (add (size-of <part> <dim> <value>)))))
```

(b) New operator for cutting to reduce the size. The stars indicate new facts acquired by the Operator Refinement Method for the New Operator.

Figure 5.5: Micro-operator formation when only some effects are needed.

A second possibility is *sequencing*, i.e. to detect a sequence of subactions that are cur-

When a given problem cannot be solved by the current set of operators because a precondition P of an operator O cannot be achieved, formulate a new operator O'.

1. Formulate a new operator. Construct a new operator O' with the desired effect and the type of the objects in O.
2. Experiment with the new operator. Execute the action. If the desired effects are not obtained, apply experimentation to isolate which of the other preconditions of O (not including P) need to be added to O'. End the process when O' is successful in a state where the preconditions of O are not true.
3. Refine the new operator. Use the ORM to find additional preconditions and effects of O'.

Table 5.4: Method for learning a new operator by micro-operator formation

rently represented by an operator. As an example, consider the operator in Figure 5.6(a) used to set up a machine for performing a machining operation. The operator has several preconditions that check the availability of a machine, a holding device, a tool and a part. The set up consists of holding the tool in the tool holder, having a holding device on the machine, and holding the part with the holding device. Since a different setup is used for each machining operation, representing this set of actions as a single operator is an efficient way of expressing the configuration for the next operation. Now, suppose that we want to perform some manual operation on a part. We ask the system to find a plan to hold it. With the available knowledge, holding a part is not possible because there are no tools that can be installed in the machine. But instead of returning a failure our system tries to find if the operator can be divided into a sequence of actions, one of them involving only holding the part. The operator to do the setup gives several independent operators, shown in Figure 5.6(b). Sequencing is done by following the same basic steps shown in Figure 5.4, but in this case additional operators are formulated with the effects not originally needed.

The two methods just presented for acquiring new operators by sequencing or by partially specifying a given one are engaged in a process that we call *micro-operator formation*[1]. Notice that the original operator is not discarded since it can still be useful to solve some problems efficiently.

[1]These methods can be thought of as opposite to the formation of macro-operators. However, learning micro-operators is not necessarily the reverse process because it does not imply the decomposition of an operator into a set of operators.

```
(SETUP
    (preconditions
      (and
        (is-a <machine> MACHINE)
        (is-of-type <tool> MACHINE-TOOL)
        (is-of-type <holding-device> HOLDING-DEVICE)
        (is-available-tool-holder <machine>)
        (is-available-tool <tool>)
        (is-available-table <machine>)
        (is-available-holding-device <holding-device>)
        (has-device <machine> <holding-device>)
        (is-empty-holding-device <holding-device> <machine>)
        (is-clean <part>)
        (~ (has-burrs <part>))))
    (effects (
        (add (holding-tool <machine> <tool>))
        (add (has-device <machine> <holding-device>))
        (add (holding <machine> <holding-device> <part> <side>)))))
```

(a) Operator to set up a machine for an operation

```
(SETUP-HOLDING-DEVICE
    (preconditions
        (is-a <machine> MACHINE)
        (is-of-type <holding-device> HOLDING-DEVICE)
        (is-available-table <machine>)
        (is-available-holding-device <holding-device>)))
    (effects (
        (add (has-device <machine> <holding-device>))))

(SETUP-HOLD
    (preconditions
        (is-a <machine> MACHINE)
        (is-of-type <holding-device> HOLDING-DEVICE)
        (has-device <machine> <holding-device>)
        (is-empty-holding-device <holding-device> <machine>)
        (is-clean <part>)
        (~ (has-burrs <part>))
    (effects (
        (add (holding <machine> <holding-device> <part> <side>)))))

(SETUP-TOOL
    (preconditions
      (and
        (is-a <machine> MACHINE)
        (is-of-type <tool> MACHINE-TOOL)
        (is-available-tool-holder <machine>)
        (is-available-tool <tool>)))
    (effects (
        (add (holding-tool <machine> <tool>)))))
```

(b) New operators for different aspects of a setup

Figure 5.6: Micro-operator formation by dividing an operator into sequential actions.

5.3.3 Learning New Operators by Splitting Existing Ones

One way is to refine an existing operator by distinguishing different aspects of the action that it represents.

Shen [Shen, 1989] describes a method for learning operators by splitting an existing one. This method takes advantage of failures like the one described in Figure 5.2. When the effects of the operator do not occur as expected we described how to refine the operator adding the condition necessary to obtain the desired effects. But we can also learn an additional operator with the effects that were observed instead of the expected effects. For example in the situation of Figure 5.2, the system learns an operator for grinding without fluid shown in Figure 5.7. The description of this method is shown in Table 5.5. Since either method can be selected under the same circumstances, the decision to be made is if both actions are interesting to the system.

```
(GRIND-WITH-FLUID
 (preconds
   (and
        (is-a <machine> GRINDER)
        (is-a <wheel> GRINDING-WHEEL)
     *  (has-fluid <machine>)
        (holding-tool <machine> <wheel>)
        (side-up-for-machining <dim> <side>)
        (holding <machine> <holding-device> <part> <side>)))
   (effects (
     *  (add (surface-finish <part> <side> SMOOTH))
        (add (size-of <part> <dim> <value>)))))

(GRIND-WITHOUT-FLUID
 (preconds
   (and
        (is-a <machine> GRINDER)
        (is-a <wheel> GRINDING-WHEEL)
        (holding-tool <machine> <wheel>)
        (side-up-for-machining <dim> <side>)
        (holding <machine> <holding-device> <part> <side>)))
   (effects (
        (add (size-of <part> <dim> <value>)))))
```

Figure 5.7: Splitting the operator GRIND when effects are different.

If after manipulating the world only a subset E of the effects of the operator happen, then a precondition of the operator is missing.

1. Select candidate preconditions. The candidate set $\Delta(S_{old}, S_{current})$ is formed by calculating all the differences between the most similar earlier state in the previous problem solving history in which O was applied successfully S_{old} and the current state $S_{current}$ (an unsuccessful application of O).
2. Identify missing precondition. Formulate experiments observing if the operator is successfully applied when one of the differences P is true in the state. Use any information available to formulate the most promising experiments first. In absence of knowledge, apply a binary search to isolate the precondition from $\Delta(S_{old}, S_{current})$.
3. Substitute O by the two new operators O_1 and O_2. O_1 is formed with O and the additional precondition P. O_2 is formed by the preconditions of O and the set of effects E.

Table 5.5: Method for splitting an operator (Shen, 1989).

5.3.4 Explicit Expressions

Another method for splitting operators follows the same steps described for learning conditional effects. Given the situation described in Figure 5.3 we could obtain two operators for grinding instead of learning new conditional effects. One would be built with the original version with the additional condition that the grit of the wheel be fine, and the additional effect that the surface finish is smooth. A second operator would be built with the original one plus the precondition that the wheel is not of fine grit. The result is shown in Figure 5.8. The method is summarized in Table 5.6.

Yet another possibility along this line is to split disjunctive concepts among different operators. Suppose that using the method for refining preconditions presented in Figure 4.1 we learn the disjunctive precondition expression shown in Figure 5.9(a). To grind a part, we need to hold it first, and to do so we need to have some kind of holding device in the grinder. This operator represents the action of putting a holding device in the grinder. The disjunction expresses that a grinder can use two different holding devices: a magnetic chuck and a vise. But instead, we could express the same concept as two different operators: one for putting a vise in a grinder, and another one for putting a magnetic chuck. The two operators are expressed in Figure 5.9(b).

Let us have a closer look at the last two methods for learning new operators by split-

```
(GRIND-WITH-COARSE-GRIT
 (preconds
   (and
        (is-a <machine> GRINDER)
        (is-a <wheel> GRINDING-WHEEL)
        (has-fluid <machine>)
        (holding-tool <machine> <wheel>)
        (side-up-for-machining <dim> <side>)
        (holding <machine> <holding-device> <part> <side>)
     *  (grit-of-wheel <wheel> FINE-GRIT)))
   (effects (
     *  (add (surface-finish <part> <side> SMOOTH))
        (del (has-fluid <machine>))
        (add (size-of <part> <dim> <value>)))))


(GRIND-WITH-NON-COARSE-GRIT
 (preconds
   (and
        (is-a <machine> GRINDER)
        (is-a <wheel> GRINDING-WHEEL)
        (has-fluid <machine>)
        (holding-tool <machine> <wheel>)
        (side-up-for-machining <dim> <side>)
        (holding <machine> <holding-device> <part> <side>)
     *  (grit-of-wheel <wheel> COARSE-GRIT)))
   (effects (
     *  (add (surface-finish <part> <side> ROUGH))
        (del (has-fluid <machine>))
        (add (size-of <part> <dim> <value>)))))
```

Figure 5.8: Splitting the operator GRIND according to its conditional effect

ting an existing one. Instead of learning a new conditional effect for an operator, we split it into two different operators using the effect and its conditions. Instead of learning a disjunctive precondition expression, we split the preconditions into two different operators. In both cases, what the system is doing is expressing some features of the action in the form of several operators thereby representing more explicitly what other methods already seen can learn. The new operators represent information in a different but logically equivalent manner. However, it is important to provide the system with this ability because it makes the description of actions easier to understand. As we mentioned in Section 3.4, an action can be represented by many operators, each operator reflecting a certain aspect of the action. It is our experience that when the domain knowledge for a planner is written, the user expresses actions not in a single complex operator, but in

If an effect E of an operator takes place in situation A but not in situation B, then it is a conditional effect of the operator.

1. Select candidate conditions for the effect. The candidate set $\Delta(S_A, S_B)$ is formed by calculating all the differences between S_A (the state in which the effect occurs) and S_B (the state in which the effect does not occur).
2. Identify missing condition. Formulate experiments observing if the effect of the operator occurs when one of the differences P is true in the state. Use any information available to formulate the most promising experiments first. In absence of knowledge, apply a binary search to isolate the precondition from $\Delta(S_A, S_B)$.
3. Substitute O by the two new operators O_1 and O_2. O_1 is formed with O adding the additional precondition P and the effect E. O_2 is formed by the preconditions of O and the effects of O excluding E.

Table 5.6: Method for splitting operators according to conditional effects

several simpler and more detailed operators that are easier for humans to understand.

5.3.5 Learning New Operators by Probing the Environment

Another way to create operators is to start with an empty description of the action and try it out in the external world and observe the changes that are produced. In this case, the system would learn a new action from null knowledge about it. This is very common in systems that explore the environment, and so they often try actions to learn about their capabilities [Shen, 1989]. We call this method *probing*, and is shown in Table 5.7. The most important part of the method is what to perceive in order to notice the effects of the action and its conditions, yet not requiring that the system collects all possible observations. A set of predicates P is chosen, to direct the system's attention. First the predicates in P are observed, then the action is executed, and finally the predicates in P are observed again. Whatever changes are observed in any P' in P are included as effects of the new operator. If no changes are observed, a new set of predicates is tried and the process is iterated. If the action still doesn't seem to change the environment, then the system tries to change the state by applying other known actions and iterate the process again. For example, suppose that we are exploring an action that pushes the drill spindle over the drill table. The drill spindle raises again after the action of pushing is stopped. If there is no part on the table, the environment remains unchanged. Executing the action in a new state when there is a part on the table will yield observations of changes in the

```
(PUT-HOLDING-DEVICE-IN-GRINDER
   (preconds
     (and
          (is-a <machine> GRINDER)
       *  (or (is-a <holding-device> MAGNETIC-CHUCK)
       *      (is-a <holding-device> VISE))
          (is-available-table <machine>)
          (is-available-holding-device <holding-device>)))
   (effects ( (add (has-device <machine> <holding-device>)))))
```

(a) Disjunction

```
(PUT-MAGNETIC-CHUCK-IN-GRINDER
   (preconds
     (and
          (is-a <machine> GRINDER)
       *  (is-a <holding-device> MAGNETIC-CHUCK)
          (is-available-table <machine>)
          (is-available-holding-device <holding-device>)))
   (effects ( (add (has-device <machine> <holding-device>)))))

(PUT-VISE-IN-GRINDER
   (preconds
     (and
          (is-a <machine> GRINDER)
       *  (is-a <holding-device> VISE)
          (is-available-table <machine>)
          (is-available-holding-device <holding-device>)))
   (effects ( (add (has-device <machine> <holding-device>)))))
```

(b) Explicit disjunction

Figure 5.9: Splitting an operator by a disjunction

external state.

5.4 Learning New Facts about the State

Even when a system has perfect knowledge about the operators of its task domain it might be impossible to solve some problems without the ability to interact with the environment. The internal state might not contain all the data about the world needed to plan. Some missing data can be acquired by direct observation, like the color of an object within the visual field. Other observations require planning. For example, in order to observe the color of an object in a distant room we first have to plan how to get

When there is an available action with no corresponding operator, probe the action and try to find a model of the action.

1. Choose what to observe. Choose a set of predicates P to observe. Collect observations.
2. Execute the action. Then, observe all predicates in P again. Make the effects of the operator be the subset of predicates P' in P that changed. If no changes are observed, either go back to step 1 or change the world by performing known actions and then go to step 1.
3. Refine the new operator. Apply the operator refinement method to find additional preconditions and effects of the new operator.

Table 5.7: Method for probing available actions to learn new operators

there. But perception and planning might not be enough to collect information about a situation, and experimentation may be the only way to acquire some facts about the state of the world.

Consider for example the observation of the lock status of a door. This is not directly observable by looking at the door. Yet we can design an experiment to collect this observation as follows. Since the predicate `(unlocked <door>)` is one of the preconditions of the operator OPEN, we can design an experiment to try to open the door. If the door is unlocked, then all the conditions of OPEN are true and the door will open. If the door is locked, then OPEN will fail. The experiment used a special version of OPEN that is missing the unknown predicate in the preconditions

Other observations need a more complicated experimentation process. For example, consider a domain where an agent can carry objects of weight smaller than its own. A simplified description of the knowledge necessary is presented in Figure 5.10. Suppose that the agent does not know its own weight. Since this observation is absolutely necessary to solve any problems involving carrying objects, the system engages in the process of acquiring this particular piece of data through experimentation.

To do so, it experiments with the action of carrying different objects and see if it can carry them or not, as shown in Figure 5.11. The weight of the objects is a controllable parameter that is chosen as part of the design of the experiments and depends on the availability of the objects. A special version of the operator is used in the experiments, constructed by dropping the preconditions which correspond to the unknown and its relationship with the controllable variable. In our example they correspond to the weight of the robot and the predicate `smaller-than`. When the action succeeds, then the

```
(CARRY-OBJECT
  (preconditions
    (and
        (arm-empty <robot>)
        (next-to <robot> <obj>)
        (weight-of <obj> <obj-weight>)
        (weight-of <robot> <robot-weight>)
        (smaller-than <obj-weight> <robot-weight>)))
 (effects (
        (del (arm-empty <robot>))
        (del (next-to <robot> <obj>))
        (del (next-to <*other-obj> <obj>))
        (add (holding <obj>)))))
```

Figure 5.10: Operator for carrying objects of smaller weight than the agent.

preconditions of CARRY-OBJECT are true, including the relationship in question. Each experiment collects new data about this relationship, constraining more the possible values of the unknown variable. Determining the value of a parameter doing binary search over its possible values is a well known experimentation method, and the process eventually converges to a value of the maximum weight that the agent can carry, which is equal to its own. Notice that this is different from situations where we need to know the value of an attribute that is deducible from observations whose acquisition requires planning. Here, we are describing a more complicated process in which the system needs to engage with experimentation strategies.

5.5 Notes on Other Types of Imperfect Knowledge

This thesis addresses the problem of acquiring knowledge in incomplete domains. As we mentioned in Section 3.3, other types of imperfections require additional mechanisms. We point out why in this section.

5.5.1 Refining Incorrect Knowledge

Incorrect postconditions can be corrected in a very straight forward way. Since the system always observes the effects of an operator immediately after applying an action, it can detect the effects that are incorrect because they will not be true in the external world. Effects that appear only sometimes should be considered conditional effects.

```
OBSERVATIONS COLLECTED:

(weight-of <obj> <weight>)   CARRY-OBJECT succeeded?   range of <robot-weight>

(weight-of object1 2)                y                    [1,?)
(weight-of object2 100)              n                    [1,100)
(weight-of object3 50)               y                    [50,100)
(weight-of object4 75)               n                    [50,75)
(weight-of object5 62)               y                    [62,75)
(weight-of object6 69)               n                    [62,69)
(weight-of object7 65)               n                    [62,65)
(weight-of object8 62)               n                    [62,62)

RESULT:  (weight-of ROBOT 62)
```

Figure 5.11: Gathering data from the state by directed experimentation. Repeated execution of the operator with objects of different weight uncovers the weight of the robot.

Detecting and removing incorrect preconditions from the operators requires mechanisms additional to the ones described above. The preconditions describe the class of states where the action can be applied. If the preconditions are incorrect, they are overspecific. This implies that the operator will be only applied to a subset of the class of states where the action can be executed. The system would need additional mechanisms that allow it to consider an incorrect operator applicable even if some of its preconditions are not matched.

The presence of incorrect knowledge might be detected by introspection if it yields inconsistencies. Experimentation could be used to determine the source of the inconsistencies and the necessary corrections.

5.5.2 Learning with an Inadequate Domain Model

The attributes known to the system might not be enough to describe the state of the external world. New attributes can be discovered from the environment when the system detects that the given attributes are not sufficient to discriminate between situations that produce different results. Shen [Shen, 1989] presents a method to discover new attributes. Another problem arises when the predicates used to represent the attributes are missing certain parameters that are important. We will suppose in our work that the system is given the necessary attributes.

Attributes observable in the world can be combined to deduce new attributes that are not directly observable. For example, the material of an object and its size determine its weight. Combinations of attributes are functional constructs. Learning these constructs requires providing the system with some basic constructs that it can combine to find the right expressions for calculating the values of the derived attribute. For example, consider our model for grinding. The operators are still incomplete because they do not contain any information about the fact that they can only be applied when the dimensions of the part become smaller (and not bigger). If a situation arises when grinding is applied with that purpose, we could detect using experimentation strategies that there is an relationship between the predicates

```
(size-of <part> <dim> <value>)
(size-of <part> <dim> <value-old>)
```

that is relevant for grinding and that should appear in the preconditions.

In fact the correct precondition expression to be learned in this case would contain:

```
(size-of <part> <dim> <value>)
(size-of <part> <dim> <value-old>)
(smaller <value> <value-old>)
```

Learning these expressions is an issue that discovery systems address and is beyond the scope of this work.

5.5.3 Learning in Intractable Domain Models

Intractability arises when control knowledge is missing. Control knowledge avoids planning inefficiencies. But in some cases, planning failures may be caused by unknown interactions among operators because the system is missing the control knowledge that represents those interactions.

A method for learning control rules by experimentation is described in [Carbonell and Gil, 1990]. The method consists of detecting goal interactions when the system observes that an action undoes a previously achieved subgoal. A lot of research has been done on learning control rules by other methods [Minton *et al.*, 1989a; Laird *et al.*, 1986; Mostow and Bhatnagar, 1987], but learning control knowledge from experience may prove to a very powerful approach.

5.6 Summary

Figure 5.12 presents a summary of all the methods described in this chapter. Notice that the method determines the type of knowledge acquired. Each method is triggered by a certain type of failure.

All the methods in Figure 5.12 have been implemented in EXPO to demonstrate the feasibility of learning by experimentation. They are triggered when EXPO detects a lack of domain knowledge, but the subsequent experimentation process is simulated manually. The full experimentation process (as we described in Chapter 4) is implemented only for learning new preconditions and new effects. Empirical tests on this implementation are described in detail in the next chapter.

WHAT IS LEARNED	*WHEN IT IS LEARNED*
new preconditions	when an action fails but it succeeded before, some unknown precondition was true before and is not true now.
new effects	when an observation contradicts information in the internal state, some action was executed that had unknown effects
new conditional effects	when an expected effect only occurs sometimes after an action is executed
new operators	
analogy	formulate operator by analogy with a known one
splitting	when an action fails but it succeeded before learn one operator for each outcome
conditional effects	when an effect only occurs sometimes, learn an operator for each case (when effect occurs and when it does not)
disjunction	make disjunction explicit having several operators
microoperators	when only some effects are wanted, build partial operator
attribute values	when needed to plan: observe, infer, and plan if needed. Design observations if several are needed.

Figure 5.12: EXPO's methods for refining incomplete domain knowledge. EXPO can acquire new preconditions, effects, operators, and attribute values.

Chapter 6

Empirical Results

Given any learning method, it is important to demonstrate its effectiveness, i.e., that it can indeed be used to acquire new knowledge. In many cases, the efficiency of learning (the time spent acquiring new knowledge) is also a main concern. This chapter presents empirical measurements that demonstrate the effectiveness and efficiency of the methods for learning by experimentation described in this thesis. The first section contains results that show the effectiveness of EXPO as it learns to refine the domain operators. And more importantly, we show that the new versions of the operators are useful for the problem solver. The second section demonstrates EXPO's efficiency. Our learning methods are very directed, and the experiments actually performed are geared towards testing the most promising hypotheses. This translates into an efficient use of time and other resources of concern.

EXPO implements the techniques for learning by experimentation presented in Chapters 5 and 4. The baseline planner is the PRODIGY system described in Section 3.6. EXPO was not tested interacting with a physical environment, but with a software system that simulates one. The details of this simulation are described in Section 3.4.2.

The results presented in this chapter correspond to two different domains. One domain is the large and complex model of process planning described in Appendix B. The other one is a simpler robot planning domain, presented in full length in Appendix A.

6.1 Effectiveness

The results presented in this section confirm that learning by experimentation is a useful technique to acquire new domain knowledge. By useful we mean that whatever is learned is needed in order to solve a task (i.e., a given set of problems). Notice how this differs

from other work on learning from the environment [Shen, 1989], where the focus is more on exploring and on learning what is unknown about the external world be it useful or not.

We want to control the degree of incompleteness of a domain in the tests. We have available a complete domain D which has all the operators with all their corresponding conditions and effects. Only c conditions and e effects are learnable by EXPO. With this complete domain, we can artificially produce domains D' that have certain percentage of incompleteness (i.e., 20% of the preconditions are missing) by removing preconditions or effects from D randomly. We will use D'_{prec20} to denote a domain that is incomplete and is missing 20% of the c learnable conditions. D'_{post20} is a domain missing 20% of the e postconditions. Notice that EXPO never has access to D, only to the incomplete domain D'.

EXPO learns new conditions and effects of incomplete operators. What is a good measure of the ampunt of new knowledge acquired by EXPO? As we described in Section 3.3, an incomplete domain may cause plan execution failures. Consider the case when an operator O is missing a condition p. Now suppose that we want to execute O in state S. If p happens to be true in S then the execution will be successful, since p is a necessary condition of O. But if p is not true in S, then the execution of O will fail. This means that missing preconditions can cause execution failures. Notice that after EXPO learns that p is a condition of O, the problem may be solved (if subgoaling on the unsatisfied new precondition P yields a subplan to achieve P and the rest of the plan does not yield any execution failures.) If knowledge is sufficiently complete then a plan is always successfully executed. If knowledge is incomplete then a plan is not necessarily successfully executed. Thus, an increment in the number of successful executions of plans after learning is indicative of the amount of new preconditions acquired.

Now consider a case where an operator O is missing the postcondition `(add (P))`. If we apply O in state S where P is not true, P will continue not to be true after O is applied. Sometime later, we may need P to be true (e.g., if it is a condition of a subsequent operator). The system believes P to be false, and after checking the external world it finds out that P is true. Incorrect predictions of literals trigger learning to acquire new effects (in this case `(add (P))` for O). After learning, P is always predicted to be true after applying O. Thus, a reduction in the number of incorrectly predicted literals is indicative of the amount of new effects acquired.

We generated n problems randomly. All of the n problems were solvable within the time bound that PRODIGY was given. From the set of n solvable problems, we randomly chose m of them to be the training set. The rest constituted the test set. Notice that both sets are independent (they do not have any common instances). Initially, PRODIGY is given the incomplete domain and EXPO starts running the training problems. For each problem, EXPO obtains a plan from PRODIGY and tries to execute it in the simulated

environment. EXPO examines any expectation failures and applies the methods for learning by experimentation described in this thesis. The more failures encountered during training, the more opportunities for learning. At certain points during learning, we run the test set. Learning is turned off at test time, so when a failure is found the internal state is corrected to reflect the observations but no learning occurs.

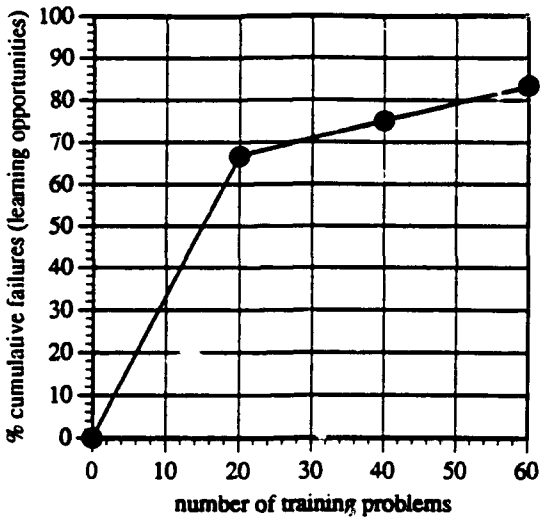
In the robot planning domain, there were 60 training problems and 12 test problems, taken from previous work in PRODIGY [Minton, 1988]. We ran tests with 20% and 50% missing preconditions. D'_{prec20} is missing 12 preconditions, and D'_{prec50} is missing 28. Figures 6.1(a) and 6.2(a) show the number of failures that EXPO detects during training with D'_{prec20} and D'_{prec50} respectively. Figures 6.1(b) and 6.2(b) show how many solutions for problems in the test set were successfully executed with D'_{prec20} and D'_{prec50} respectively. The number of plans that PRODIGY is able to execute correctly increases with learning. This is because the problems in the training set cause expectation failures, which EXPO uses to gain new knowledge after undergoing experimentation.

For D'_{prec20} EXPO has not examined enough failures to acquire all domain knowledge, but it has acquired the knowledge necessary to execute successfully the solutions to all the problems in the test set. For D'_{prec50}, only 4 solutions to the test problems are executed successfully. This is because the training set does not contain problems that cause failures that yield the knowledge necessary to overcome the execution failures in the test set. After training with the test set, one more new condition is learned which turns out to be the common cause of the execution failures in the test set and thus the solutions to all the test problems can be successfully executed.

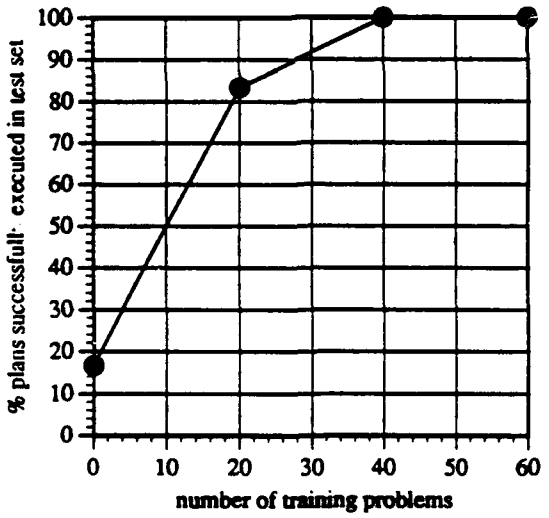
In the process planning domain, there were two sets of training and test problems. Each training set had 100 problems, and each test set had 20 problems. The problems were generated randomly, as we explain in Appendix B. The tests were run in domains with 10% and 30% incompleteness. Figures 6.3 and 6.4 present results for D'_{prec10} and D'_{prec30} respectively when EXPO acquires new preconditions. The curves show results very similar to the results obtained for the robot planning domain

As an example of what is learned, EXPO refines the operator GRIND shown in Figure 4.1 adding the facts shown with a star (*) in Figure 6.5.

We also run tests with domains where 20% and 50% of the postconditions of operators were missing. Figures 6.6 and 6.7 show the results for D'_{post20} and D'_{post50} respectively in the robot planning domain. As more failures are encountered, EXPO acquires new effects of operators. Thus, the number of incorrect predictions when running the test set is reduced continuously.

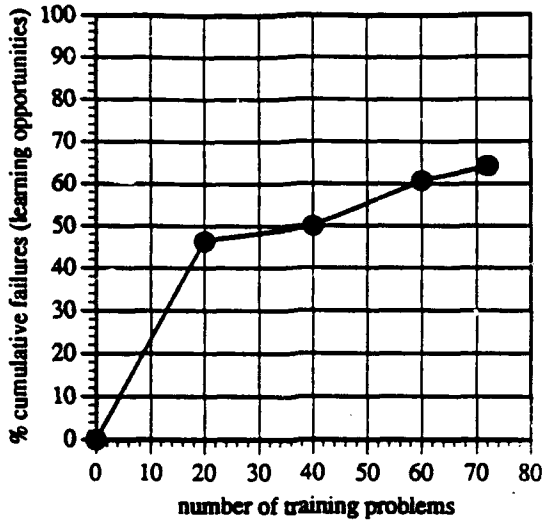


(a)

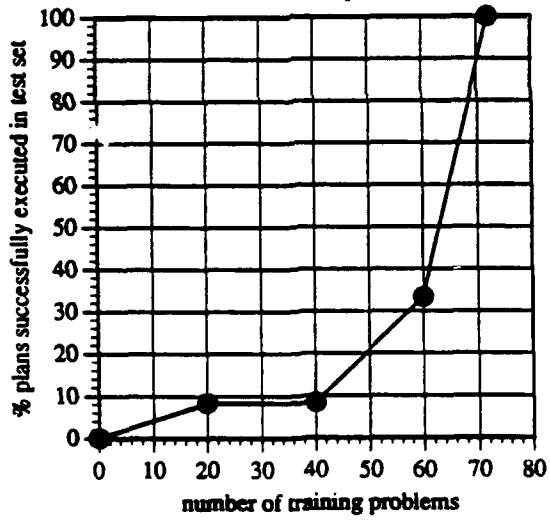


(b)

Figure 6.1: Effectiveness in the robot planning domain with 20% of the preconditions missing (D'_{prec20}). (a) Cumulative number of failures in the execution of solutions to training problems encountered by EXPO as the size of the training set increases. Each failure presents an opportunity for learning. (b) The number of plans successfully executed in the test set increases as EXPO examines more failures. The number of additional plans successfully executed is indicative of the amount of knowledge acquired by EXPO.

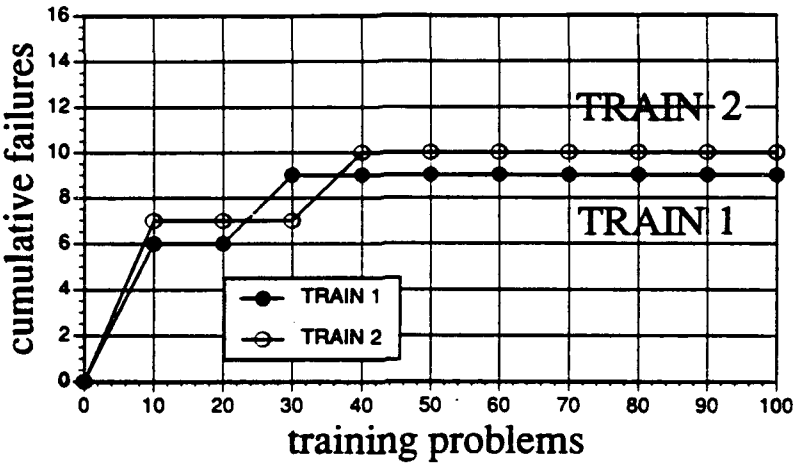


(a)

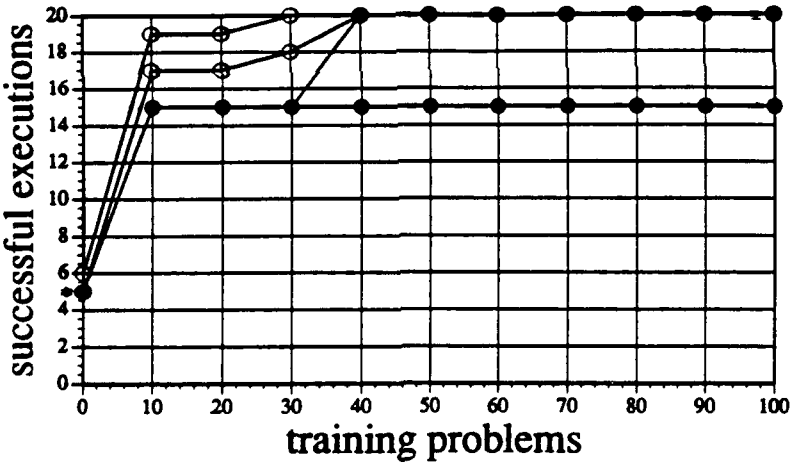


(b)

Figure 6.2: Effectiveness in the robot planning domain with 50% of the preconditions missing (D'_{prec50}). (a) Cumulative number of failures in the execution of solutions to training problems encountered by EXPO as the size of the training set increases. Each failure presents an opportunity for learning. (b) The number of plans successfully executed in the test set increases as EXPO examines more failures. The number of additional plans successfully executed is indicative of the amount of knowledge acquired by EXPO.

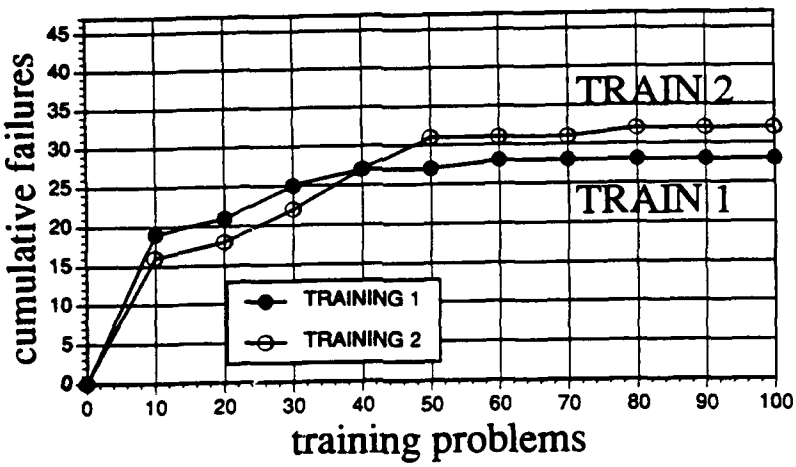


(a)

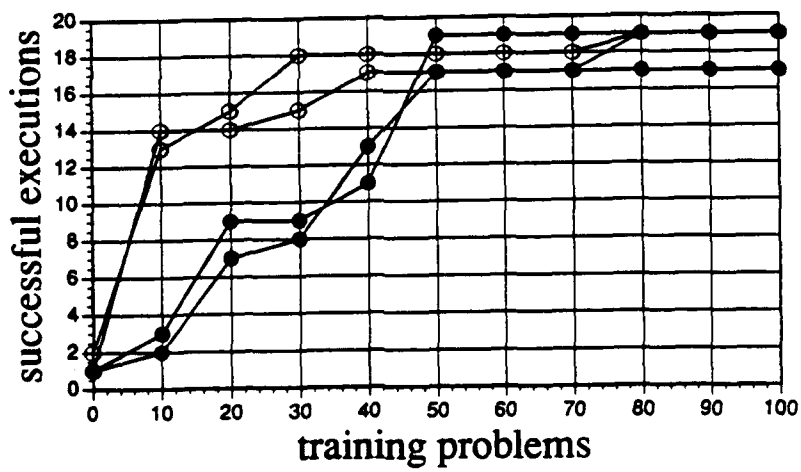


(b)

Figure 6.3: Effectiveness in the process planning domain with 10% of the preconditions missing (D'_{prec10}). (a) Cumulative number of failures in the execution of solutions to training problems encountered by EXPO as the size of the training set increases. Each failure presents an opportunity for learning. (b) The number of plans successfully executed in the test set increases as EXPO examines more failures. The number of additional plans successfully executed is indicative of the amount of knowledge acquired by EXPO.



(a)



(b)

Figure 6.4: Effectiveness in the process planning domain with 30% of the preconditions missing (D'_{prec30}). (a) Cumulative number of failures in the execution of solutions to training problems encountered by EXPO as the size of the training set increases. Each failure presents an opportunity for learning. (b) The number of plans successfully executed in the test set increases as EXPO examines more failures. The number of additional plans successfully executed is indicative of the amount of knowledge acquired by EXPO.

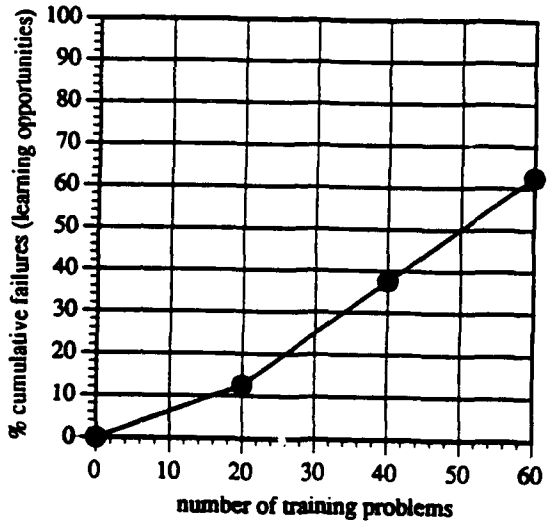
```
(GRIND
 (preconditions
   (and
        (is-a <machine> GRINDER)
        (is-a <tool> GRINDING-WHEEL)
        (is-a <part> PART)
     *  (is-clean <part>)
     *  (~ (has-burrs <part>))
     *  (has-fluid <machine>)
        (~ (same <dim> DIAMETER))
        (holding-tool <machine> <tool>)
        (side-up-for-machining <dim> <side>)
        (holding <machine> <holding-device> <part> <side>)))
 (effects (
     *  (del (is-clean <part>))
     *  (add (has-burrs <part>))
     *  (del (has-fluid <machine>))
     *  (del (surface-finish <part> <side> <s-q>))
        (del (size-of <part> <dim> <value-old>))
        (add (size-of <part> <dim> <value>)))))
```

Figure 6.5: A More Complete Model of Grinding

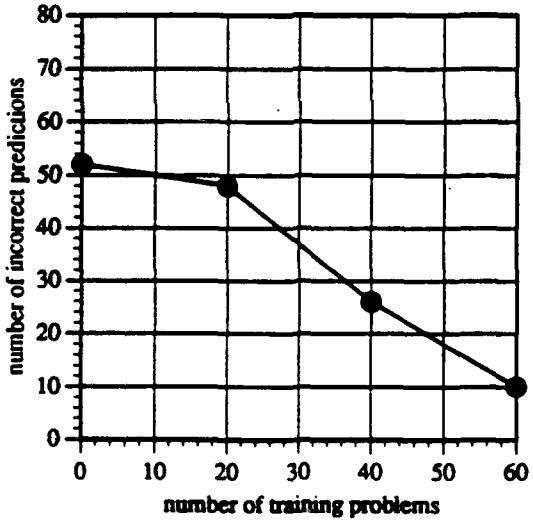
6.2 Efficiency

The previous section showed that EXPO is indeed able to acquire new knowledge through experimentation. So the techniques presented in this thesis are effective in that they do lead EXPO to the cause and repair of the failures it encounters. But this is not the only desirable property of this type of learning. In fact, as we discussed in Chapter 4, minimising the number of experiments is another important concern. This section takes a close look at the efficiency of the experimentation process.

Figures 6.8 and 6.9 present the number of experiments that are required to recover from the failures shown in Figures 6.1(a) and 6.2(a) respectively. The heuristics used are represented by a letter: g for generalization, s for structural similarity, and l for locality. Without any of our hypothesis-selection heuristics, many experiments are needed. The other curves show how effective each heuristic is individually and in combination with others. Each heuristic contributes in its own way to reducing the number of experiments. Notice that although the divide and conquer experimentation does a smaller number of experiments than some of the heuristics used in isolation, every experiment requires a larger number of goal statements to satisfy, as explained in Section 4.2. For 20% incompleteness, the three heuristics combined yield the best results. For 50% incompleteness, gl is about as good as gls. This is because when the operators are very incomplete similar

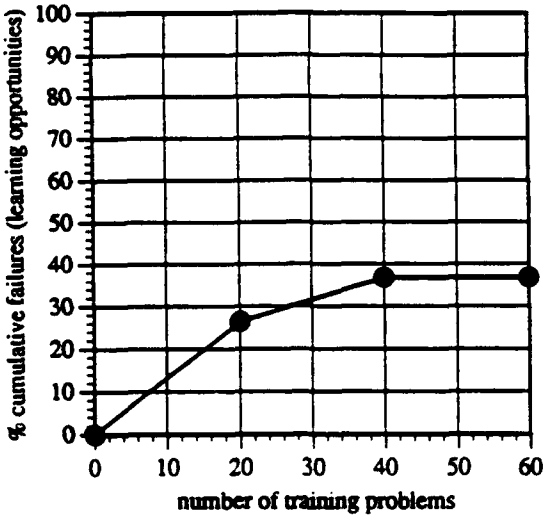


(a)

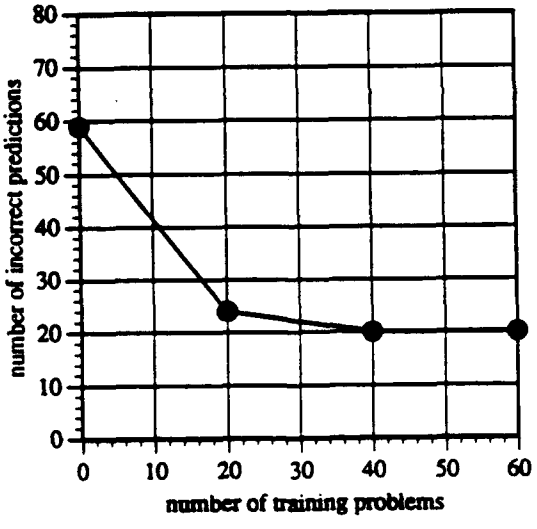


(b)

Figure 6.6: Acquisition of new effects in the robot planning domain with 20% of the effects missing (D'_{post20}). (a) Cumulative number of failures in the execution of training problems encountered by EXPO as the size of the training set increases. Each failure presents an opportunity for learning. (b) The number of incorrectly predicted literals in the test set decreases as EXPO examines more failures. This is indicative of the amount of new effects of operators acquired by EXPO.



(a)



(b)

Figure 6.7: Acquisition of new effects in the robot planning domain with 50% of the effects missing (D'_{post50}). (a) Cumulative number of failures in the execution of training problems encountered by EXPO as the size of the training set increases. Each failure presents an opportunity for learning. (b) The number of incorrectly predicted literals in the test set decreases as EXPO examines more failures. This is indicative of the amount of new effects of operators acquired by EXPO.

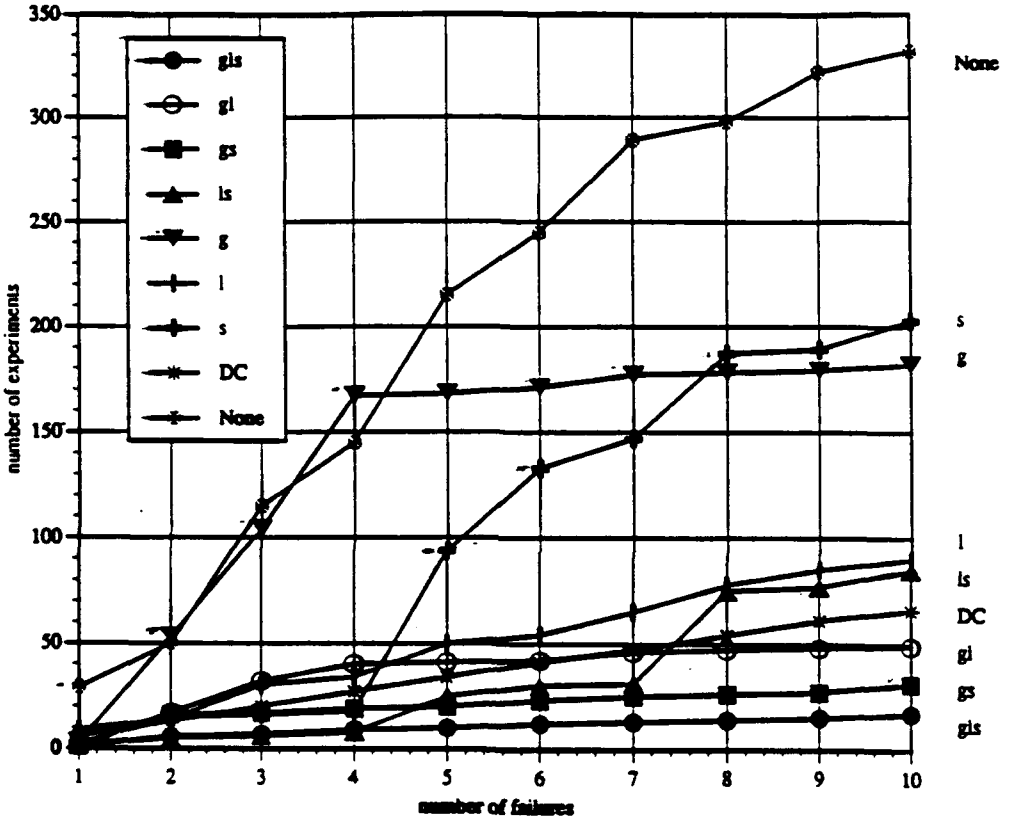


Figure 6.8: Given D'_{prec20} number of experiments that are necessary with all the combinations of the three hypotheses-selection heuristics: generalization of experience (g), locality (l), and structural similarity (s). The number of experiments needed is greatly reduced when the three of them are used.

operators may be missing the same conditions, so s is not very helpful. The effectiveness of s improves as new knowledge is added to the domain, this can be seen in the numbers of the last rows of the tables presented next.

The following tables show the numerical results that are summarized in Figures 6.8

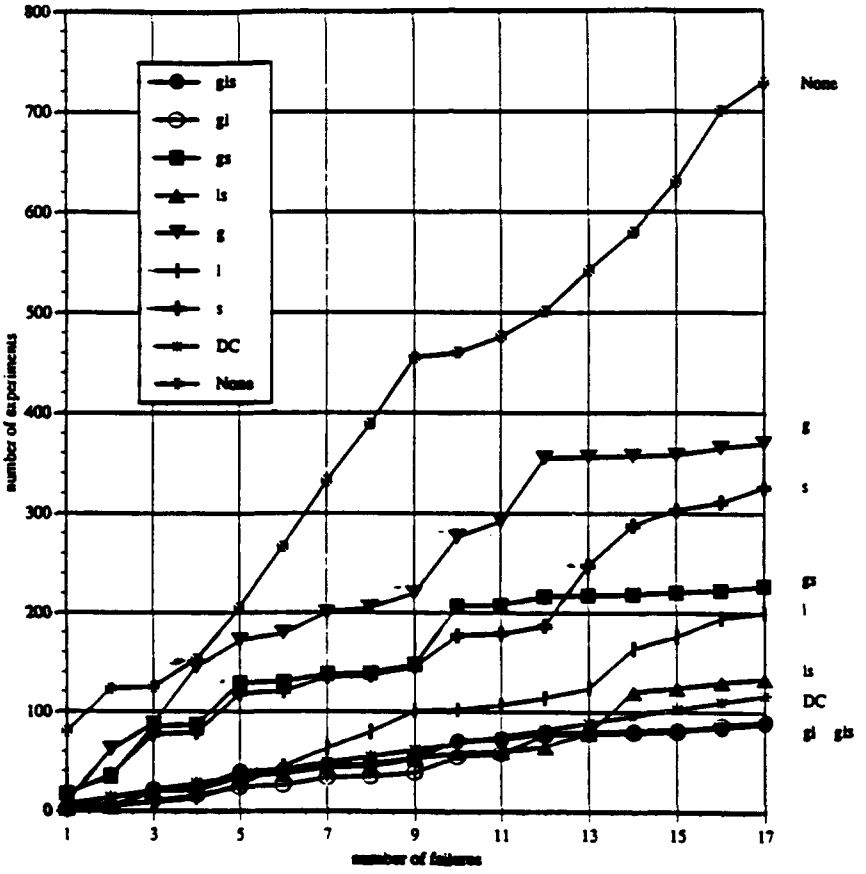


Figure 6.9: Given D'_{prec50} number of experiments that are necessary with all the combinations of the three hypotheses-selection heuristics: generalization of experience (g), locality (l), and structural similarity (s). The number of experiments needed is greatly reduced when the three of them are used.

and 6.9. The number of experiments needed with each combination of heuristics is shown for each failure. Also shown is the number of experiments needed if no heuristics are used, which corresponds to the ranking by default of the missing condition in the list of hypotheses. The last column shows the total number of hypotheses in the set of candidates.

With D'_{prec20}:

number of experiments								*total number of*
gls	*gl*	*gs*	*ls*	*g*	*l*	*s*	*no heuristics*	*of hypotheses*
1	2	1	2	3	9	2	29	59
5	15	15	3	50	5	13	21	67
1	15	1	1	51	16	1	65	66
2	8	2	2	63	4	2	30	77
1	1	1	17	1	16	76	70	77
2	1	3	5	3	4	39	30	78
1	4	2	1	6	11	14	44	60
1	1	1	44	1	13	40	9	86
1	1	1	2	1	7	2	24	68
2	1	4	7	3	5	14	10	24

With D'_{prec50}:

number of experiments								*total number of*
gls	*gl*	*gs*	*ls*	*g*	*l*	*s*	*no heuristics*	*of hypotheses*
2	1	17	2	9	5	18	80	81
2	3	18	2	53	2	18	43	82
18	6	50	16	26	1	41	2	73
1	6	2	1	57	5	2	28	76
17	8	42	14	27	14	39	52	70
1	2	1	2	7	18	2	62	69
5	8	8	7	21	18	15	65	68
1	1	1	1	5	17	1	57	59
7	4	9	8	15	21	10	66	67
17	16	59	6	56	1	31	5	56
1	3	1	1	16	5	2	16	66
7	19	9	5	63	7	7	25	66
1	1	1	14	1	10	61	41	74
1	1	1	41	1	40	40	38	81
1	1	2	4	1	12	15	50	87
2	7	2	6	7	19	8	70	86
5	4	5	4	5	6	15	28	73

Let us examine this last table more closely, and look at the effects of each heuristic in the ranking of candidate conditions. As we pointed out before, the heuristic of structural similarity is increasingly more effective, since the operators in the hierarchy become more complete through learning. The predicate (**inroom <key> <room>**) is added as a new precondition of LOCK in the 7th failure (row 7 in the table), and also as a new precondition of UNLOCK in the 16th failure (row 16). In the 7th failure, the similarity heuristic does not find similar operators with this condition, so it ranks it low. In the 16th failure, LOCK is found very close to UNLOCK in the hierarchy and it has the precondition (**inroom <key> <room>**), so this candidate is ranked high. The generalization of past experience also becomes more effective when more executions of the operators are examined. Row 14 corresponds to the new precondition (**arm-empty**) of the operator PICKUP-OBJ. Notice that the new precondition does not have any of the parameters of the operator, and as a result, the locality heuristic ranks this candidate very low.

In summary, the combination of the three heuristics (generalization of experience, structural similarity, and locality) reduces dramatically the number of experiments required, and yileds the best performance. A divide and conquer strategy over the set of candidates requires many more experiments that also have more complex setups.

Chapter 7

Conclusions and Future Work

This chapter summarizes the contributions and limitations of this thesis, and outlines some areas of future work.

7.1 Summary of the Approach and Results

The thesis presents a general framework and an effective and efficient approach to the practical implementation of learning by experimentation. The methods presented are domain independent, and do not require any knowledge other than the domain defined by the user for planning. The thesis shows that it is possible to recover from knowledge-level impasses autonomously without need of causal explanations of the failure. Automated learning by experimentation is a desirable capability of autonomous systems and it relieves humans of much work in the engineering of knowledge, taking over the burden of ensuring knowledge completeness and maintenance once an initial knowledge base is constructed. This thesis presents a step in that direction.

The work in this thesis is applicable to a wide range of planning problems in which the following items are feasible:

- discrete-valued features describe the state of the world.
- actions are axiomatizable as deterministic operators in terms of the features that describe the state.
- reliable observations are available on demand.
- noise-free sensors.

- no other agents are present whose actions interfere with the planner's.

Future work includes extensions in all these areas, and is discussed in Section 7.3.

7.2 Contributions

The theoretical contributions described in this thesis are:

- A closed-loop integration of planning and learning from the environment by experimentation where new knowledge is immediately incorporated, tested, and used by the planner
- Systematic augmentation of a given incomplete domain by directed experimentation, triggered each time that there is a knowledge impasse
- Acquisition of domain knowledge of a planner so it is able to solve problems it could not solve before learning
- Computationally effective methodology for correcting incomplete domain knowledge
- Exploration of methods for learning by experimentation, including hypothesis generation, filtering, prioritization, and empirical validation.
- Domain-independent heuristics for finding relevant hypotheses
- Efficient and customizable experimentation control strategies maximizing convergence on identification of missing knowledge
- A framework for the interaction between the main planning space and the experimentation planning space

EXPO's implementation of the above presents the following practical contributions:

- An implementation that demonstrates the synergistic interactions between a planning system and an active learner that acquires domain knowledge from the environment
- An empirical evaluation of methods with various degrees of initial incompleteness in the domain, and with different sets of experimentation heuristics to identify the sources of power and extensibility of the approach.
- Multi-domain generality (robot planning and complex process planning)

7.3 EXPO's Limitations and Future Work

This section describes the limitations of this thesis and some suggestions for future work. The section is organized under three major areas: the specific methods for learning by experimentation, the interaction with the environment, and the global framework for experimentation.

7.3.1 Extensions to the Learning Methods

EXPO's current implementation for learning new preconditions, described in Chapter 4, is limited to acquiring a new conjunct which is an observable predicate. Every member of the set of candidate new preconditions is an observable predicate. EXPO considers as hypotheses only the members of that set, and tests them through experiments. If the experiments show that none of the predicates in the set is a new precondition, EXPO gives up on acquiring the precondition autonomously: it notifies the user that it knows that the operator is missing a precondition and that it cannot find it. EXPO considers only the inclusion of additional conjunctive predicates (the most common and useful scenario). Other possible hypotheses to be considered as candidate conditions are:

- Disjunctive expressions of predicates
- Inferred predicates deduced from a state by theorem proving (or other inferential processes)
- Quantified expressions of some predicates
- Predicates that are never observed because they were not needed for planning before (i.e., the weight of a box)
- A functional relation of several predicate arguments

EXPO examines the hypotheses produced by the method. If the experiments show that the missing condition is not one of them, then it should consider the above possibilities. However, to simplify the implementation, EXPO abandons learning and continues plan execution.

Using a more sophisticated concept learning algorithm for generalizations would expand EXPO's capabilities to acquire expressions other than conjunctive ones, including disjunctions and quantified expressions. Functional relations between predicate arguments require an algorithm with the capability to construct new functions, such as BACON [Langley *et al.*, 1987].

Learning preconditions that are inferred or unobserved predicates is an open research question. EXPO could expand its set of hypotheses to inferred and unobserved predicates, and deduce or observe their value during the experiments. This solution would be very inefficient because a large number of predicates may belong to this group.

EXPO assumes an initially incomplete knowledge base, but many other types of imperfections are possible, as described in Section 3.3. The domain knowledge can be incorrect, inadequate, or intractable. Section 5.5 outlined some possibilities to address these different types of imperfections.

We described in Chapter 5 how experimentation is needed to collect observations from the state. When we can't observe directly if a door is locked or unlocked, we can experiment on opening it and we know immediately the answer. Robotics systems may benefit enormously from using this capablity of experimentation.

Expanding the system's vocabulary by learning new features about objects in the state is an open area. [Shen, 1989] addressed this problem in the LIVE system, which could detect hidden features and learn their value. Research on constructive learning is expanding horizons in this direction, and the area of autonomous learning from the environment should benefit from that.

In short, whereas this dissertation makes a substantial contribution to learning by experimentation, there is a vast open space of additional research topics in proactive experimentation.

7.3.2 Interaction with the Environment

The work in this thesis has a limited form of interaction with the environment. The assumption of noise-free sensors allows the algorithms to count on reliable feedback, but it is not a very realistic assumption for some domains. Work on inductive learning from noisy data could be applied if sensors were unreliable. Experience on robotics research leads us to believe that this is not a simple problem.

The presence of other agents that can change the environment and inadvertently may cause the internal state to diverge from the external world. Their differences would cause failures that are not due to a fault in the knowledge base. A solution to the problem of determining the cause of divergence could be a more sophisticated credit-assignment system for failures. Nondeterminism in the actions would cause a similar problem.

Learning by experimentation autonomously from the environment is not as direct for many applications outside planning. Other intelligent systems are focused on tasks where the interaction with the environment is expensive, impractical, or simply impossible to obtain. Medical diagnosis systems are a good example. However, it is conceivable to

use EXPO's strategies in such systems to produce experiments that would translate into questions for an expert, or a request for additional data gathering.

7.3.3 Toward a Framework for Learning by Experimentation

Figure 7.1 summarizes the framework for learning from the environment by experimentation presented in this thesis. Given a goal, a plan to achieve it is executed while the external environment is monitored. Any differences with the internal state are detected by various methods that suggest a type of fault in the domain knowledge that may have caused the expectation failure. The methods also construct a set of concrete hypotheses to repair the fault. After being heuristically filtered, one hypothesis is tested at a time with an experiment. After the experiment's requirements are designed, a plan is constructed to achieve the situation desired. After the execution of the plan and the experiment, observations are collected to conclude if the experiment was successful or not. Upon success, the hypothesis is confirmed and the domain knowledge is adjusted. Upon failure, the experimentation process is iterated until success or until no more hypotheses are left to be considered. This framework has shown to be an effective way to address experimentation but also raises many issues.

The learning methods are not completely independent, and may be triggered by the same failure. For example, suppose that a known effect of an operator does not occur upon execution. This triggers two methods that suggest different adjustments to the domain knowledge: either the effect of the operator is conditional, or a precondition is missing. Another example of the strong interaction between methods is raised by a problem that the planner cannot solve. It may be unsolvable because an existing operator is incomplete (i.e., missing an effect) or because the domain is missing one operator, or simply be unsolvable regardless of completeness of knowledge. A framework to address the interdependencies of the methods is needed. One method is chosen to be the first, and if the experiments do not uncover the knowledge fault the other method is tried. This issue suggests that intelligent shift of attention would be very advantageous.

In fact, intelligent shift of attention is necessary at all levels of the experimentation process, as shown in Figure 7.1. If the current hypothesis (general or particular) has taken enough time, another hypothesis may be chosen for consideration. If no satisfactory plan is found for an experiment, the experiment design may be changed. And if a reasonable amount of time and resources have been spent on studying a failure, the study may be suspended and continued in the future when more information becomes available.

Learning from the environment is a necessary capability for autonomous intelligent agents that must solve tasks in the real world. This thesis presents a step towards the autonomous refinement of knowledge through experimentation.

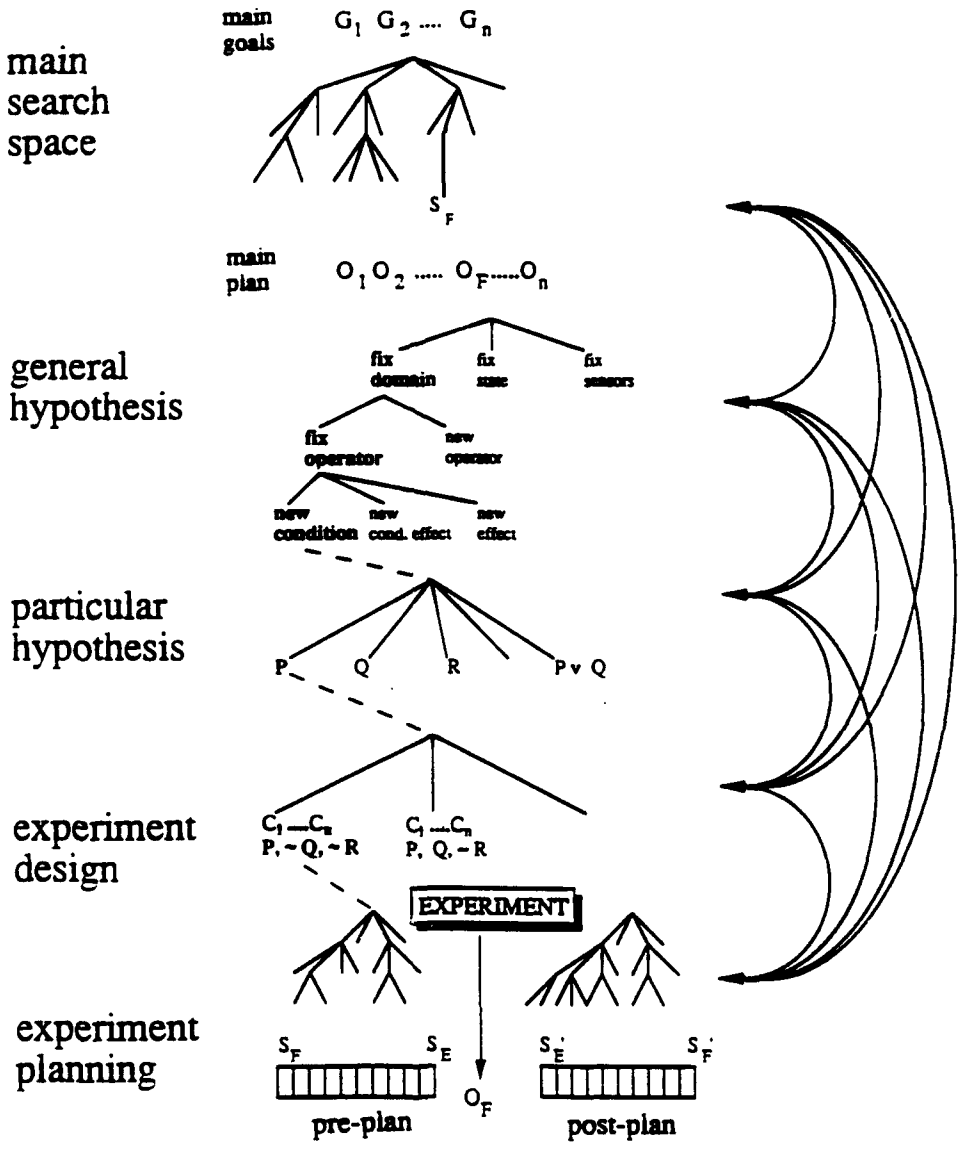


Figure 7.1: Toward a framework for learning by experimentation. Failures in the execution of a plan trigger learning. A general cause for the failure is hypothesized, then instantiated to a particular hypothesis. The design of experiments includes planning the experimental setup. A flexible framework for experimentation would include intelligent shift of attention at all levels of the process, as indicated by the arrows on the right of the figure.

Appendix A

The Robot Planning Domain

This appendix describes the robot planning domain implemented in the PRODIGY architecture used for examples and empirical tests in this thesis.

First, it includes a description of a domain and a quantitative and qualitative characterization. Then, the implementation of this domain in the PRODIGY architecture is listed. The rest of the appendix includes the incomplete versions and problems used in the empirical tests, and the numerical results obtained that were used in the graphs for Chapter 6.

This domain was chosen to test EXPO because of its realistic description of a robot task, its medium size, and because it has been used extensively for testing other learning methods [Minton, 1988; Etzioni, 1990; Knoblock, 1991; Pérez and Etzioni, 1992]. The domain is essentially the same used in these references, except that variable types have been added to the preconditions. PRODIGY needs generators for every variable, and the original domain used the predicates in the conditions as such. If a predicate that is a generator is missing from the preconditions, PRODIGY could not use the operators for planning.

A.1 Description of the Domain

This domain is an extension of the one used for STRIPS [Fikes and Nilsson, 1971]. In the original domain, a robot could move between rooms and transport boxes. In this domain, the robot can also open and close doors, and if it is holding the right key it can lock and unlock doors. Boxes are carriable or pushable, and all keys are carriable. Boxes and keys are objects. Only carriable objects may be held by the robot for transportation, other objects must be pushed to be moved. The actions available are: pickup an object,

put down and object, put down an object next to another one, push an object to a door, push an object through a door to another room, go through a door to another room, go next to a door, push an object, go next to an object, and open, close, lock, and unlock a door.

The domain can be qualitatively and quantitatively described as follows:

- Some quantitative features are:
  - There are 14 operators.
  - There are 11 predicates: `connects`, `carriable`, `pushable`, `is-room`, `is-object`, `is-door`, `is-key`, `dr-to-rm`, `inroom`, `next-to`, `holding`. Only 3 of them (`inroom`, `next-to`, `holding`) are changed by the operators.
  - There are four types of variables: object, room, door, and key.
  - The average number of parameters for an operator is 2.
  - The average number of preconditions of operators is 4.
  - The average number of effects of operators is 4.
  - 57 preconditions and 38 effects are learnable, a total of 95 learnable items.
- All the operators' effects are reversible.
- There are no inference rules for deducing new facts about a given state.
- There are no functions that compute the value of a predicate.
- The precondition expression of all the operators is a conjunction of predicates that are included in the state (i.e., are not to be derived or computed through a function, as explained in Section 3.6.).
- There are no negations in the precondition expressions.
- All effects of all operators are unconditional, i.e., their occurence is not dependent on the context given by the state at application time (as explained in Section 3.6).

All preconditions that are not type specifications are learnable by EXPO. The type specifications must be present in the operator as generators for PRODIGY 2.0, the version of the system on which EXPO is implemented (for more details on generators see [Minton *et al.*, 1989b]). However, this is not a deficiency of EXPO, but of PRODIGY 2.0, one that is being corrected in later versions of the system [Veloso, 1989; Carbonell *et al.*, 1992]. Only the effects used for backchaining are not learnable by EXPO. The reason for this

is that an operator must be used by the planner in order for EXPO to observe the outcomes of its execution. When operators are written by a human, they express an action or change in the world, so it is reasonable to assume that the operators initially given to EXPO have some effect.

A.2 Domain Operators

```
(PICKUP-OBJ
 (params (<object>))
 (preconds (and
   (arm-empty)
   (next-to robot <object>)
   (is-object <object>)
   (carriable <object>)))
 (effects (
   (del (arm-empty))
   (del (next-to <object> <*other-ob30>))
   (del (next-to <*other-ob31> <object>))
   (add (holding <object>)))))

(PUTDOWN
 (params (<object>))
 (preconds (and
   (holding <object>)
   (is-object <object>)))
 (effects (
   (del (holding <*other-ob35>))
   (add (next-to robot <object>))
   (add (arm-empty)))))

(PUTDOWN-NEXT-TO
 (params (<object> <other-ob> <room>))
 (preconds (and
   (holding <object>)
   (is-object <object>)
   (is-object <other-ob>)
   (inroom <other-ob> <room>)
   (is-room <room>)
   (inroom <object> <room>)
   (next-to robot <other-ob>)))
 (effects (
   (del (holding <*other-ob35>))
   (add (next-to <object> <other-ob>))
   (add (next-to robot <object>))
   (add (next-to <other-ob> <object>))
   (add (arm-empty)))))

(PUSH-TO-DR
 (params (<object> <door> <room>))
 (preconds (and
   (is-door <door>)
   (dr-to-rm <door> <room>)
   (is-room <room>)
   (inroom <object> <room>)
   (is-object <object>)
   (next-to robot <object>)
   (pushable <object>)))
 (effects (
   (del (next-to robot <*other-ob3>))
   (del (next-to <object> <*other-ob5>))
   (del (next-to <*other-ob13> <object>))
   (add (next-to <object> <door>))
   (add (next-to robot <object>)))))

(PUSH-THRU-DR
 (params (<object> <door> <room> <other-room>))
 (preconds (and
   (is-room <room>)
   (dr-to-rm <door> <room>)
   (is-door <door>)
   (dr-open <door>)
   (next-to <object> <door>)
   (next-to robot <object>)
   (is-object <object>)
   (pushable <object>)
   (connects <door> <room> <other-room>)
   (is-room <other-room>)
   (inroom <object> <other-room>)))
 (effects (
   (del (next-to robot <*other-ob1>))
   (del (next-to <object> <*other-ob12>))
   (del (next-to <*other-ob7> <object>))
   (del (inroom robot <*other-ob21>))
   (del (inroom <object> <*other-ob22>))
   (add (inroom robot <room>))
   (add (inroom <object> <room>))
   (add (next-to robot <object>)))))

(GO-THRU-DR
 (params (<door> <room> <other-room>))
 (preconds (and
   (arm-empty)
   (is-room <room>)
   (dr-to-rm <door> <room>)
   (is-door <door>)
   (dr-open <door>)
   (next-to robot <door>)
   (connects <door> <room> <other-room>)
   (is-room <other-room>)
   (inroom robot <other-room>)))
 (effects (
   (del (next-to robot <*other-ob19>))
   (del (inroom robot <*other-ob20>))
   (add (inroom robot <room>)))))

(CARRY-THRU-DR
 (params (<object> <door> <room> <other-room>))
 (preconds (and
   (is-room <room>)
   (dr-to-rm <door> <room>)
   (is-door <door>)
   (dr-open <door>)
   (is-object <object>)
   (holding <object>)
   (connects <door> <room> <other-room>)
   (is-room <other-room>)
   (inroom <object> <other-room>)
   (inroom robot <other-room>)
   (next-to robot <door>)))
 (effects (
```

```
   (del (next-to robot <*other-ob48>))
   (del (inroom robot <*other-ob41>))
   (del (inroom <object> <*other-ob42>))
   (add (inroom robot <room>))
   (add (inroom <object> <room>))))))

(GOTO-DR
 (params (<door> <room>))
 (preconds (and
   (is-door <door>)
   (dr-to-rm <door> <room>)
   (inroom robot <room>)
   (is-room <room>)))
 (effects (
   (del (next-to robot <*other-ob18>))
   (add (next-to robot <door>))))))

(PUSH-BOX
 (params (<object> <other-ob> <room>))
 (preconds (and
   (is-object <object>)
   (is-object <other-ob>)
   (inroom <other-ob> <room>)
   (is-room <room>)
   (inroom <object> <room>)
   (pushable <object>)
   (next-to robot <object>)))
 (effects (
   (del (next-to robot <*other-ob14>))
   (del (next-to <object> <*other-ob5>))
   (del (next-to <*other-ob6> <object>))
   (add (next-to robot <object>))
   (add (next-to robot <other-ob>))
   (add (next-to <object> <other-ob>))
   (add (next-to <other-ob> <object>))))))

(GOTO-OBJ
 (params (<object> <room>))
 (preconds (and
   (is-object <object>)
   (inroom <object> <room>)
   (is-room <room>)
   (inroom robot <room>)))
 (effects (
   (add (next-to robot <object>))
   (del (next-to robot <*other-ob109>))))))

(OPEN
 (params (<door>))
 (preconds (and
   (is-door <door>)
   (unlocked <door>)
   (next-to robot <door>)
   (dr-closed <door>)))
 (effects (
   (del (dr-closed <door>))
   (add (dr-open <door>))))))

(CLOSE
 (params (<door>))
 (preconds (and
   (is-door <door>)
   (next-to robot <door>)
   (dr-open <door>)))
 (effects (
   (del (dr-open <door>))
   (add (dr-closed <door>))))))

(LOCK
 (params (<door> <key> <room>))
 (preconds (and
   (is-door <door>)
   (is-key <door> <key>)
   (holding <key>)
   (dr-to-rm <door> <room>)
   (is-room <room>)
   (inroom <key> <room>)
   (next-to robot <door>)
   (dr-closed <door>)
   (unlocked <door>)))
 (effects (
   (del (unlocked <door>))
   (add (locked <door>))))))

(UNLOCK
 (params (<door> <key> <room>))
 (preconds (and
   (is-door <door>)
   (is-key <door> <key>)
   (holding <key>)
   (dr-to-rm <door> <room>)
   (is-room <room>)
   (inroom <key> <room>)
   (inroom robot <room>)
   (next-to robot <door>)
   (locked <door>)))
 (effects (
   (del (locked <door>))
   (add (unlocked <door>))))))
```

A.3 Incomplete Domains

The 12 preconditions missing in D'_{prec20} are the following:

operator	*precondition*
pickup-obj	(arm-empty)
push-to-dr	(dr-to-rm <door> <room>)
go-thru-dr	(dr-open <door>)
carry-thru-dr	(connects <door> <room> <other-room>)
carry-thru-dr	(next-to robot <door>)
goto-obj	(inroom robot <room>)
open	(unlocked <door>)
open	(next-to robot <door>)
lock	(next-to robot <door>)
unlock	(holding <key>)
unlock	(inroom robot <room>)
unlock	(next-to robot <door>)

The 8 effects missing in D'_{post20} are the following:

operator	*postcondition*
pickup-obj	(del (next-to <*other-ob31> <object>))
putdown-next-to	(del (holding <*other-ob35>))
push-thru-dr	(del (next-to <*other-ob7> <object>))
push-thru-dr	(del (inroom robot <*other-ob21>))
carry-thru-dr	(del (inroom robot <*other-ob41>))
carry-thru-dr	(del (inroom <object> <*other-ob42>))
open	(del (dr-closed <door>))
close	(del (dr-open <door>))

The 28 preconditions missing in D'_{prec50} are the following:

operator	*precondition*
pickup-obj	(arm-empty)
pickup-obj	(next-to robot <object>)
putdown	(holding <object>)
putdown-next-to	(inroom <object> <room>)
push-to-dr	(inroom <object> <room>)
push-to-dr	(next-to robot <object>)
push-thru-dr	(dr-to-rm <door> <room>)
push-thru-dr	(dr-open <door>)
push-thru-dr	(next-to <object> <door>)
push-thru-dr	(inroom <object> <other-room>)
carry-thru-dr	(next-to robot <door>)
goto-dr	(dr-to-rm <door> <room>)
push-box	(pushable <object>)
push-box	(next-to robot <object>)
goto-obj	(inroom <object> <room>)
open	(dr-closed <door>)
close	(next-to robot <door>)
close	(dr-open <door>)
lock	(holding <key>)
lock	(dr-to-rm <door> <other-room>)
lock	(inroom <key> <other-room>)
lock	(next-to robot <door>)
lock	(dr-closed <door>)
unlock	(holding <key>)
unlock	(dr-to-rm <door> <room>)
unlock	(inroom <key> <room>)
unlock	(inroom robot <room>)
unlock	(locked <door>)

The 19 effects missing in D'_{post50} are the following:

operator	*postcondition*
putdown	(add (next-to robot <object>))
putdown-next-to	(del (holding <*other-ob35>))
putdown-next-to	(add (next-to robot <object>))
push-to-dr	(add (next-to <object> <*other-ob5>))
push-to-dr	(add (next-to robot <object>))
push-thru-dr	(del (next-to robot <*other-ob1>))
push-thru-dr	(del (next-to <object> <*other-ob12>))
push-thru-dr	(del (inroom <object> <*other-ob22>))
push-thru-dr	(add (next-to robot <object>))
go-thru-dr	(del (next-to robot <*other-ob19>))
carry-thru-dr	(del (inroom <object> <*other-ob42>))
push-box	(del (next-to <object> <*other-ob5>))
push-box	(del (next-to <*other-ob6> <object>))
push-box	(add (next-to robot <object>))
push-box	(add (next-to robot <other-ob>))
goto-obj	(del (next-to robot <*other-ob109>))
close	(del (dr-open <door>))
lock	(del (unlocked <door>))
unlock	(del (locked <door>))

A.4 Training and Test Problems

The problems used to test this domain are a subset of those used in [Minton, 1988]. The problems were generated randomly, following a procedure described in the reference mentioned. We used 60 training problems and 12 test problems. Problems 1 to 20 are taken from ps0 and ps1, and are called *Train*1. Problems 21 to 40 are taken from ps2 and ps3 and form *Train*2. Problems 41 to 60 are taken from ps5 and ps6, and are called *Train*3. The twelve test problems are in ps4.

A.5 Tables of Results

This section presents the numerical results that were used for the graphs in Chapter 6.

A.5.1 Missing 20% of the Preconditions

The following table shows the numerical results that are summarized in Figure 6.1 (20% incompleteness):

number of training problems	*cumulative number of learning opportunities*	*number of plans successfully executed in test set*
0	0	2
20	8	10
40	9	12
60	10	12

Notice that after training with *Train*2, 100% of the test problems can be solved. However, the domain knowledge is still not complete, so EXPO continues learning new facts in subsequent training problems.

New preconditions for D'_{prec20} were learned by EXPO in the following order:

1. (**next-to robot <door>**) of CARRY-THRU-DR
2. (**next-to robot <door>**) of UNLOCK
3. (**holding <key>**) of UNLOCK
4. (**next-to robot <door>**) of OPEN
5. (**inroom robot <room>**) of GOTO-OBJ
6. (**unlocked <door>**) of OPEN
7. (**dr-open <door>**) of GO-THRU-DR
8. (**arm-empty**) of PICKUP-OBJ
9. (**next-to robot <door>**) of LOCK
10. (**dr-to-rm <door> <room>**) of PUSH-TO-DR

A.5.2 Missing 50% of the Preconditions

The following table shows the numerical results that are summarized in Figure 6.2 (50% incompleteness):

number of training problems	*cumulative number of learning opportunities*	*number of plans successfully executed in test set*
0	0	0
20	13	1
40	14	1
60	17	4
Testset	18	12

In this case, 4 of the 12 test problems cannot be successfully executed after training with all the training sets. This is due to the nature of the training sets, which may not uncover all the necessary failures. This is shown by training EXPO with the test set, after which all the test problems can be solved.

New preconditions for D'_{prec50} were learned by EXPO in the following order:

1. **(next-to robot <object>)** of PICKUP-OBJ
2. **(holding <object>)** of PUTDOWN
3. **(dr-closed <door>)** of LOCK
4. **(next-to robot <door>)** of CLOSE
5. **(holding <key>)** of LOCK
6. **(next-to robot <door>)** of LOCK
7. **(inroom <key> <room>)** of LOCK
8. **(next-to robot <door>)** of CARRY-THRU-DR
9. **(next-to robot <object>)** of PUSH-TO-DR
10. **(next-to <object> <door>)** of PUSH-THRU-DR
11. **(inroom <object> <room>)** of PUSH-TO-DR

12. (holding <key>) of UNLOCK
13. (inroom <object> <room>) of GOTO-OBJ
14. (arm-empty) of PICKUP-OBJ
15. (dr-to-rm <door> <room>) of GOTO-DR
16. (inroom robot <room>) of UNLOCK
17. (dr-open <door>) of PUSH-THRU-DR

A.5.3 Missing 20% of the Effects

The following table shows the numerical results obtained from EXPO that are summarized in Figure ?? for the domain with 20% incompleteness.

number of training problems	*cumulative number of learning opportunities*	*number of incorrect predictions*
0	0	52
20	1	48
40	3	26
60	5	10

The postconditions learned by EXPO given D'_{post20} are (in this order):

1. (del (dr-open <door>)) of CLOSE
2. (del (dr-closed <door>)) of OPEN
3. (del (inroom robot <other-room>)) of CARRY-THRU-DR
4. (del (next-to robot <object>)) of PICKUP-OBJ
5. (del (inroom robot <*var>)) of PUSH-THRU-DR

Notice that items 3 and 4 are more specific than the effects that actually appear in the original domain. But in fact, in item 3 the effect (del (inroom robot <other-room>)) learned by EXPO is the correct one for the operator: (del (inroom robot <*other-ob41>)) is overly general since the robot is leaving the room <other-room>. In item 4, (del

(next-to robot <object>)) is overly specific because <object> ceases to be next to anything else besides the robot. In this case, EXPO can learn that fact by adding another effect (del (next-to <*var> <object>)).

A.5.4 Missing 50% of the Effects

The following table shows the numerical results obtained from EXPO that are summarized in Figure ?? for the domain with 50% incompleteness.

number of training problems	*cumulative number of learning opportunities*	*number of incorrect predictions*
0	0	59
20	5	24
40	7	20
60	7	20

The postconditions learned by EXPO given D'_{post50} are (in this order):

1. (add (next-to robot <object>)) of PUSH-TO-DR
2. (add (next-to robot <object>)) of PUSH-THRU-DR
3. (del (next-to robot <*var>)) of GOTO-OBJ
4. (del (dr-open <door>)) of CLOSE
5. (del (inroom <object> <*var>)) of PUSH-THRU-DR
6. (del (next-to robot <*var>)) of GO-THRU-DR
7. (add (next-to robot <other-ob>)) of PUSH-BOX

Appendix B

The Process Planning Domain

This appendix describes the process planning domain used in the examples and in the empirical test of this thesis. This domain is different from the scheduling domain used in other work in PRODIGY. First, the appendix describes a domain and gives a quantitative and qualitative characterization of it. Then, the implementation in the PRODIGY system is listed. The rest of the appendix includes the incomplete versions and problems used in the empirical tests, and the numerical results obtained that were used in the graphs for Chapter 6.

A more complete description of the technical content of this process planning specification can be found in [Gil, 1991].

This domain was chosen to test EXPO because it is very elabotate and knowledge intensive. The variety of alternative processes, their complexity, and their interactions make the planning task very complex.

B.1 Description of the Domain

Process planning is a major component of product manufacturing. A product is designed to satisfy some desired set of specifications. A product is typically made of several components, also called parts. When the design is completed, production continues by planning the sequences of processes to be performed on raw material to produce a part. This process planning includes operations to machine, join and finish parts. Machining processes include cutting the part to a certain size, inflicting a feature such as a hole, and producing a certain roughness on a surface. Joining operations include bolting and welding parts. Finishing operations give the part a certain surface coating, such as a rust resistant finish.

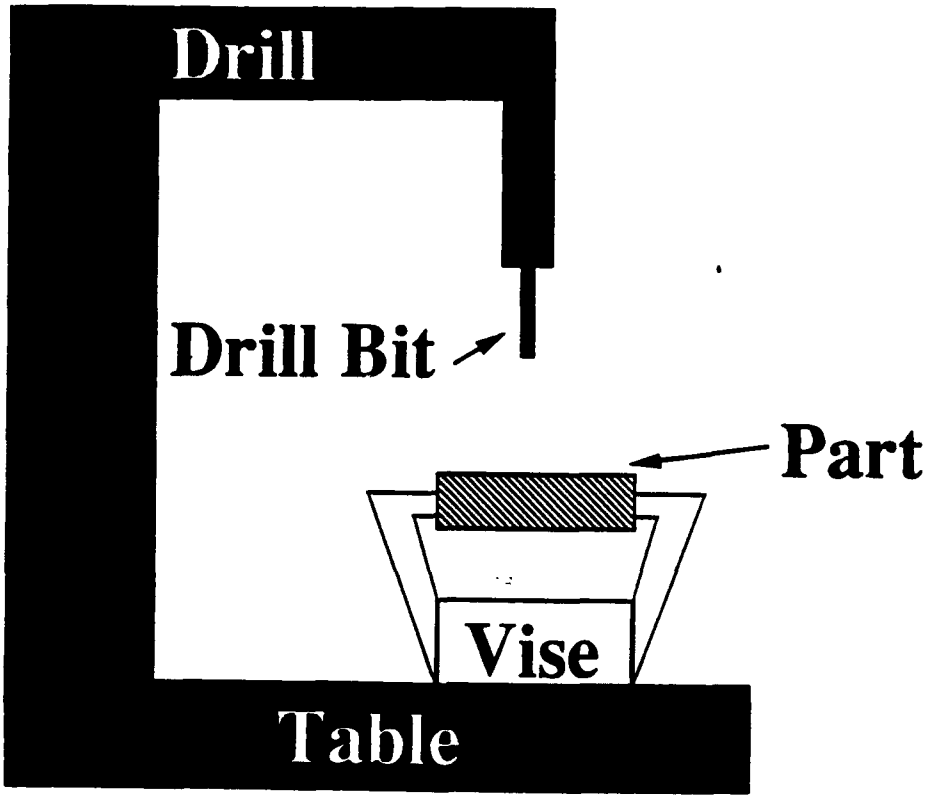


Figure B.1: The Setup for a Drilling Operation

Each operation involves a machine, a holding device to grasp the part, and a tool. Figure B.1 depicts a setup for drilling a hole.

A drilling machine holds a tool called a drill bit, and on its table there is a holding device called a vise that is grasping the part.

There are many constraints for the tools and holding devices that can be used with each machine.

An expert machinist assisted in the construction of the domain, and helped with the description of real machine setups and sample parts for constructing problems. Figure B.2 shows an actual request. It is one of the examples included in [Hayes, 1990], selected from a job shop that serves the Mechanical Engineering Department of Carnegie Mellon University.

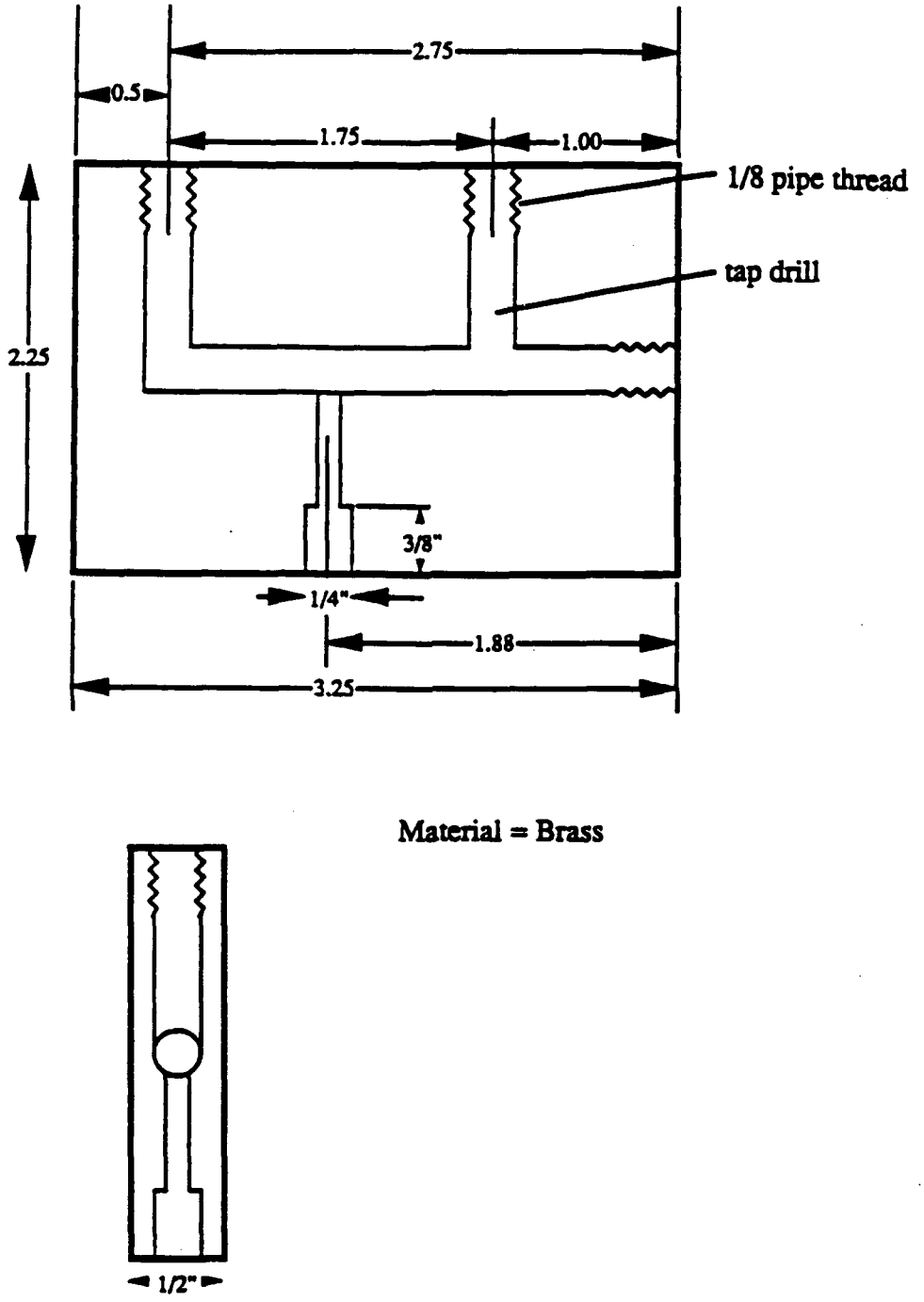


Figure B.2: An Example of a Request for a Part (from [Hayes, 1990])

This domain can be qualitatively and quantitatively described as follows:

- Some qualitative parameters are:
  - There are 117 rules, that include 73 operators and 44 inference rules.
  - There are 33 different types of objects.
  - There are 93 predicates. 55 of them are static (i.e., do not change during problem solving), 27 of them are closed world (i.e., appear in the effects of some operator but not in the effects of inference rules), 20 of them are open world (i.e., deduced by inference rules), and 7 are computed by Lisp functions.
  - The average number of parameters for an operator is 5.
  - The average number of preconditions for an operator is 8.
  - The average number of effects for an operator is 6.
  - 163 preconditions and 154 effects are learnable, a total of 317 learnable items.
- The effects of most operators are not reversible.
- The precondition expression of some operators involves facts not present in the state such as negations, predicates computed by functions, and predicates derived by inference rules.
- There are context-dependent effects in some operators.

As explained in Appendix A, type specifications and backchaining effects are not learnable by EXPO.

B.2 The Domain

The domain operators, inference rules, and function predicates are listed below.

B.2.1 Operators

```
;;;*****************************************************
; MACHINE: DRILL

; operators for making holes

(DRILL-WITH-SPOT-DRILL
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole> <side>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (is-a <drill-bit> SPOT-DRILL)
   (holding-tool <machine> <drill-bit>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (add (has-spot <part> <hole> <side> <loc-x>
                  <loc-y>)))))

(DRILL-WITH-TWIST-DRILL
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole> <side> <hole-depth>
          <hole-diam>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (same <drill-bit-diam> <hole-diam>)
   (diameter-of-drill-bit <drill-bit>
                          <drill-bit-diam>)
   (is-a <drill-bit> TWIST-DRILL)
   (has-spot <part> <hole> <side> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (has-spot <part> <hole> <side> <loc-x>
                  <loc-y>))
   (add (has-hole <part> <hole> <side> <hole-depth>
                  <hole-diam> <loc-x> <loc-y>)))))

(DRILL-WITH-HIGH-HELIX-DRILL
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole> <side> <hole-depth>
          <hole-diam>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (same <drill-bit-diam> <hole-diam>)
   (diameter-of-drill-bit <drill-bit>
                          <drill-bit-diam>)
   (is-a <drill-bit> HIGH-HELIX-DRILL)
   (has-fluid <machine> <fluid> <part>)
   (has-spot <part> <hole> <side> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (has-spot <part> <hole> <side> <loc-x>
                  <loc-y>))
   (add (has-hole <part> <hole> <side> <hole-depth>
                  <hole-diam> <loc-x> <loc-y>)))))

(DRILL-WITH-STRAIGHT-FLUTED-DRILL
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole> <side> <hole-depth>
          <hole-diam>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (same <drill-bit-diam> <hole-diam>)
   (diameter-of-drill-bit <drill-bit>
                          <drill-bit-diam>)
   (is-a <drill-bit> STRAIGHT-FLUTED-DRILL)
   (smaller <hole-depth> 2)
   (material-of <part> BRASS)
   (has-spot <part> <hole> <side> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (has-spot <part> <hole> <side> <loc-x>
                  <loc-y>))
   (add (has-hole <part> <hole> <side> <hole-depth>
                  <hole-diam> <loc-x> <loc-y>)))))

(DRILL-WITH-OIL-HOLE-DRILL
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole> <side> <hole-depth>
          <hole-diam>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (same <drill-bit-diam> <hole-diam>)
   (diameter-of-drill-bit <drill-bit>
                          <drill-bit-diam>)
   (is-a <drill-bit> OIL-HOLE-DRILL)
   (smaller <hole-depth> 20)
   (has-fluid <machine> <fluid> <part>)
   (has-spot <part> <hole> <side> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (has-spot <part> <hole> <side> <loc-x>
                  <loc-y>))
   (add (has-hole <part> <hole> <side> <hole-depth>
                  <hole-diam> <loc-x> <loc-y>)))))

(DRILL-WITH-GUN-DRILL
```

```
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole> <side> <hole-depth>
          <hole-diam>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (same <drill-bit-diam> <hole-diam>)
   (diameter-of-drill-bit <drill-bit>
                          <drill-bit-diam>)
   (is-a <drill-bit> GUN-DRILL)
   (has-fluid <machine> <fluid> <part>)
   (has-spot <part> <hole> <side> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (has-spot <part> <hole> <side> <loc-x>
                  <loc-y>))
   (add (has-hole <part> <hole> <side> <hole-depth>
                  <hole-diam> <loc-x> <loc-y>)))))


(DRILL-WITH-CENTER-DRILL
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole> <side> <drill-bit-diam>
          <loc-x> <loc-y>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (diameter-of-drill-bit <drill-bit>
                          <drill-bit-diam>)
   (same <drill-bit-diam> <hole-diam>)
   (is-a <drill-bit> CENTER-DRILL)
   (has-spot <part> <hole> <side> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (has-spot <part> <hole> <side> <loc-x>
                  <loc-y>))
   (add (has-hole <part> <hole> <side> 1/8
                  <hole-diam> <loc-x> <loc-y>))
   (add (has-center-hole <part> <hole> <side>
                         <loc-x> <loc-y>)))))

;; operators for finishing holes

(TAP
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (same <drill-bit-diam> <hole-diam>)
   (diameter-of-drill-bit <drill-bit>
                          <drill-bit-diam>)
   (is-a <drill-bit> TAP)
   (has-hole <part> <hole> <side> <hole-depth>
             <hole-diam> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (- (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <hol      av> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (is-reamed <part> <hole> <side> <hole-depth>
                   <hole-diam> <loc-x> <loc-y>))
   (add (is-tapped <part> <hole> <side> <hole-depth>
                   <hole-diam> <loc-x> <loc-y>)))))

(COUNTERSINK
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (angle-of-drill-bit <drill-bit> <angle>)
   (is-a <drill-bit> COUNTERSINK)
   (has-hole <part> <hole> <side> <hole-depth>
             <hole-diam> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (- (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (add (is-countersinked <part> <hole> <side>
              <hole-depth> <hole-diam> <loc-x>
              <loc-y> <angle>)))))

(COUNTERBORE
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> DRILL)
   (size-of-drill-bit <drill-bit> <counterbore-size>)
   (is-a <drill-bit> COUNTERBORE)
   (has-hole <part> <hole> <side> <hole-depth>
             <hole-diam> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (- (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (add (is-counterbored <part> <hole> <side>
             <hole-depth> <hole-diam> <loc-x>
             <loc-y> <counterbore-size>)))))

(REAM
 (params (<machine> <drill-bit> <holding-dev> <part>
          <hole> <side> <hole-depth> <hole-diam>))
 (preconds (and
   (is-a <part> PART)
```

```
   (is-a <machine> DRILL)
   (same <drill-bit-diam> <hole-diam>)
   (diameter-of-drill-bit <drill-bit>
                          <drill-bit-diam>)
   (is-a <drill-bit> REAMER)
   (smaller <hole-depth> 2)
   (has-fluid <machine> <fluid> <part>)
   (has-hole <part> <hole> <side> <hole-depth>
             <hole-diam> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (is-tapped <part> <hole> <side> <hole-depth>
                   <hole-diam> <loc-x> <loc-y>))
   (add (is-reamed <part> <hole> <side> <hole-depth>
                   <hole-diam> <loc-x> <loc-y>)))))


;;;**********************************************

; MACHINE: MILLING MACHINE

(SIDE-MILL
 (params (<machine> <part> <milling-cutter>
          <holding-dev> <side> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> MILLING-MACHINE)
   (is-of-type <milling-cutter> MILLING-CUTTER)
   (or (same <dim> WIDTH)
       (same <dim> LENGTH))
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
   (smaller-than-2in  <value-old> <value>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <milling-cutter>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             ROUGH-MILL))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))

(FACE-MILL
 (params (<machine> <part> <milling-cutter>
          <holding-dev> <side> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> MILLING-MACHINE)
   (is-of-type <milling-cutter> MILLING-CUTTER)
   (same <dim> HEIGHT)
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <milling-cutter>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             ROUGH-MILL))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))

(DRILL-WITH-SPOT-DRILL-IN-MILLING-MACHINE
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole> <side>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> MILLING-MACHINE)
   (is-a <drill-bit> SPOT-DRILL)
   (holding-tool <machine> <drill-bit>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (add (has-spot <part> <hole> <side> <loc-x>
                  <loc-y>)))))

(DRILL-WITH-TWIST-DRILL-IN-MILLING-MACHINE
 (params (<machine> <drill-bit> <holding-dev>
          <part> <hole> <side> <hole-depth>
          <hole-diam>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> MILLING-MACHINE)
   (same <drill-bit-diam> <hole-diam>)
   (diameter-of-drill-bit <drill-bit>
                          <drill-bit-diam>)
   (is-a <drill-bit> TWIST-DRILL)
   (has-spot <part> <hole> <side> <loc-x> <loc-y>)
   (holding-tool <machine> <drill-bit>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (has-spot <part> <hole> <side> <loc-x>
                  <loc-y>))
   (add (has-hole <part> <hole> <side> <hole-depth>
                  <hole-diam> <loc-x> <loc-y>)))))

;;;**********************************************
; MACHINE: LATHE

(ROUGH-TURN-RECTANGULAR-PART
 (params (<machine> <part> <toolbit> <holding-dev>
          <diameter-new>))
 (preconds (and
```

```
   (is-a <machine> LATHE)
   (is-a <toolbit> ROUGH-TOOLBIT)
   (shape-of <part> RECTANGULAR)
   (size-of <part> HEIGHT <h>)
   (size-of <part> WIDTH <w>)
   (smaller <diameter-new> <h>)
   (smaller <diameter-new> <w>)
   (holding-tool <machine> <toolbit>)
   (side-up-for-machining DIAMETER <side>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (size-of <part> HEIGHT <h>))
   (del (size-of <part> WIDTH <w>))
   (add (size-of <part> DIAMETER <diameter-new>))
   (del (surface-coating-side <part> SIDE1
                              <*surface-coating>))
   (del (surface-coating-side <part> SIDE2
                              <*surface-coating>))
   (del (surface-coating-side <part> SIDE4
                              <*surface-coating>))
   (del (surface-coating-side <part> SIDE5
                              <*surface-coating>))
   (del (surface-coating-side <part> SIDE0
                              <*surface-coating>))
   (del (surface-finish-side <part> SIDE1 <sf1>))
   (del (surface-finish-side <part> SIDE2 <sf2>))
   (del (surface-finish-side <part> SIDE4 <sf4>))
   (del (surface-finish-side <part> SIDE5 <sf5>))
   (add (surface-finish-side <part> SIDE0
                             ROUGH-TURN)))))

(ROUGH-TURN-CYLINDRICAL-PART
 (params (<machine> <part> <toolbit> <holding-dev>
          <diameter-new>))
 (preconds (and
   (is-a <machine> LATHE)
   (is-a <toolbit> ROUGH-TOOLBIT)
   (shape-of <part> CYLINDRICAL)
   (size-of <part> DIAMETER <diam>)
   (smaller <diameter-new> <diam>)
   (holding-tool <machine> <toolbit>)
   (side-up-for-machining DIAMETER <side>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (size-of <part> DIAMETER <diam>))
   (add (size-of <part> DIAMETER <diameter-new>))
   (del (surface-coating-side <part> SIDE0
                              <*surface-coating>))
   (del (surface-finish-side <part> SIDE0 <sf>))
   (add (surface-finish-side <part> SIDE0
                             ROUGH-TURN)))))

(FINISH-TURN
 (params (<machine> <part> <toolbit> <holding-dev>
          <diameter-new>))
 (preconds (and
   (is-a <machine> LATHE)
   (is-a <toolbit> FINISH-TOOLBIT)
   (shape-of <part> CYLINDRICAL)
   (size-of <part> DIAMETER <diam>)
   (finishing-size <diam> <diameter-new>)
   (holding-tool <machine> <toolbit>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> SIDE0)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (size-of <part> DIAMETER <diam>))
   (add (size-of <part> DIAMETER <diameter-new>))
   (del (surface-coating-side <part> SIDE0
                              <*surface-coating>))
   (del (surface-finish-side <part> SIDE0 <sf>))
   (add (surface-finish-side <part> SIDE0
                             FINISH-TURN)))))

(MAKE-THREAD-WITH-LATHE
 (params (<machine> <part> <holding-dev> <side>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> LATHE)
   (is-a <toolbit> V-THREAD)
   (shape-of <part> CYLINDRICAL)
   (holding-tool <machine> <toolbit>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> SIDE0)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> SIDE0
                              <*surface-coating>))
   (del (surface-finish-side <part> SIDE0 <sf>))
   (add (surface-finish-side <part> SIDE0 TAPPED)))))

(MAKE-KNURL-WITH-LATHE
 (params (<machine> <part> <holding-dev> <side>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> LATHE)
   (is-a <toolbit> KNURL)
   (shape-of <part> CYLINDRICAL)
   (holding-tool <machine> <toolbit>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> SIDE0)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> SIDE0
                              <*surface-coating>))
   (del (surface-finish-side <part> SIDE0 <sf>))
   (add (surface-finish-side <part> SIDE0
                             KNURLED)))))

(FILE-WITH-LATHE
```

```
 (params (<machine> <part> <holding-dev>
          <lathe-file> <diameter-new>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> LATHE)
   (is-a <lathe-file> LATHE-FILE)
   (shape-of <part> CYLINDRICAL)
   (size-of <part> DIAMETER <diam>)
   (finishing-size <diam> <diameter-new>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> SIDE0)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (size-of <part> DIAMETER <diam>))
   (add (size-of <part> DIAMETER <diameter-new>))
   (del (surface-coating-side <part> SIDE0
                              <*surface-coating>))
   (del (surface-finish-side <part> SIDE0 <sf>))
   (add (surface-finish-side <part> SIDE0
                             ROUGH-GRIND)))))

(POLISH-WITH-LATHE
 (params (<machine> <part> <holding-dev> <cloth>
          <diameter-new>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> LATHE)
   (is-a <cloth> ABRASIVE-CLOTH)
   (material-of-abrasive-cloth <cloth> EMERY)
   (shape-of <part> CYLINDRICAL)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> SIDE0)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> SIDE0
                              <*surface-coating>))
   (del (surface-finish-side <part> SIDE0 <*s-q>))
   (add (surface-finish-side <part> SIDE0
                             POLISHED)))))

;;;*****************************************************
; MACHINE: SHAPER

(ROUGH-SHAPE
 (params (<machine> <part> <cutting-tool>
          <holding-dev> <side> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> SHAPER)
   (is-a <cutting-tool> ROUGHING-CUTTING-TOOL)
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <cutting-tool>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             ROUGH-SHAPED))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))

(FINISH-SHAPE
 (params (<machine> <part> <cutting-tool>
          <holding-dev> <side> <dim> <value>))
  (preconds (and
   (is-a <part> PART)
   (is-a <machine> SHAPER)
   (is-a <cutting-tool> FINISHING-CUTTING-TOOL)
   (size-of <part> <dim> <value-old>)
   (finishing-size <value-old> <value>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <cutting-tool>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> <side>)))
  (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             FINISH-SHAPED))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))

;;;*****************************************************
; MACHINE: PLANER

(ROUGH-SHAPE-WITH-PLANER
 (params (<machine> <part> <cutting-tool>
          <holding-dev> <side> <dim> <value>))
  (preconds (and
   (is-a <part> PART)
   (is-a <machine> PLANER)
   (is-a <cutting-tool> ROUGHING-CUTTING-TOOL)
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <cutting-tool>)
   (holding <machine> <holding-dev> <part> <side>)))
  (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             ROUGH-PLANED))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))
```

```
(FINISH-SHAPE-WITH-PLANER
 (params (<machine> <part> <cutting-tool>
          <holding-dev> <side> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> PLANER)
   (is-a <cutting-tool> FINISHING-CUTTING-TOOL)
   (size-of <part> <dim> <value-old>)
   (finishing-size <value-old> <value>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <cutting-tool>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             FINISH-PLANED))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))

;;;**************************************************
; MACHINE: GRINDER

(ROUGH-GRIND-WITH-HARD-WHEEL
 (params (<machine> <part> <wheel> <holding-dev>
          <side> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> GRINDER)
   (is-a <wheel> GRINDING-WHEEL)
   (has-fluid <machine> <fluid> <part>)
   (hardness-of-wheel <wheel> HARD)
   (hardness-of <part> SOFT)
   (~ (material-of <part> BRONZE))
   (~ (material-of <part> COPPER))
   (grit-of-wheel <wheel> COARSE-GRIT)
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <wheel>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             ROUGH-GRIND))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))

(ROUGH-GRIND-WITH-SOFT-WHEEL
 (params (<machine> <part> <wheel> <holding-dev>
          <side> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> GRINDER)
   (is-a <wheel> GRINDING-WHEEL)
   (has-fluid <machine> <fluid> <part>)
   (hardness-of-wheel <wheel> SOFT)
   (hardness-of <part> HARD)
   (grit-of-wheel <wheel> COARSE-GRIT)
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <wheel>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             ROUGH-GRIND))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))

(FINISH-GRIND-WITH-HARD-WHEEL
 (params (<machine> <part> <wheel> <holding-dev>
          <side> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> GRINDER)
   (is-a <wheel> GRINDING-WHEEL)
   (has-fluid <machine> <fluid> <part>)
   (hardness-of-wheel <wheel> HARD)
   (hardness-of <part> SOFT)
   (~ (material-of <part> BRONZE))
   (~ (material-of <part> COPPER))
   (grit-of-wheel <wheel> FINE-GRIT)
   (size-of <part> <dim> <value-old>)
   (finishing-size <value-old> <value>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <wheel>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             FINISH-GRIND))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))


(FINISH-GRIND-WITH-SOFT-WHEEL
 (params (<machine> <part> <wheel> <holding-dev>
          <side> <dim> <value>))
```

```
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> GRINDER)
   (is-a <wheel> GRINDING-WHEEL)
   (has-fluid <machine> <fluid> <part>)
   (hardness-of-wheel <wheel> SOFT)
   (hardness-of <part> HARD)
   (grit-of-wheel <wheel> FINE-GRIT)
   (size-of <part> <dim> <value-old>)
   (finishing-size <value-old> <value>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <wheel>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             FINISH-GRIND))
   (add (size-of <part> <dim> <value>))
   (del (size-of <part> <dim> <value-old>)))))

;;;*****************************************************
; MACHINE: CIRCULAR-SAW

(CUT-WITH-CIRCULAR-COLD-SAW
 (params (<machine> <part> <attachment>
          <holding-dev> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> CIRCULAR-SAW)
   (is-a <attachment> COLD-SAW)
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <attachment>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             FINISH-MILL))
   (del (size-of <part> <dim> <value-old>))
   (add (size-of <part> <dim> <value>)))))

(CUT-WITH-CIRCULAR-FRICTION-SAW
 (params (<machine> <part> <attachment>
          <holding-dev> <dim> <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> CIRCULAR-SAW)
   (is-a <attachment> FRICTION-SAW)
   (has-fluid <machine> <fluid> <part>)
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <attachment>)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side>
                             ROUGH-MILL))
   (del (size-of <part> <dim> <value-old>))
   (add (size-of <part> <dim> <value>)))))

;;;*****************************************************
; MACHINE: BAND-SAW

(CUT-WITH-BAND-SAW
 (params (<machine> <part> <attachment> <dim>
          <value>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> BAND-SAW)
   (is-a <attachment> SAW-BAND)
   (size-of <part> <dim> <value-old>)
   (smaller <value> <value-old>)
  (side-up-for-machining <dim> <side>)
  (holding-tool <machine> <attachment>)
  (~ (has-burrs <part>))
  (is-clean <part>)
  (on-table <machine> <part>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side> <*s-q>))
   (add (surface-finish-side <part> <side> SAWCUT))
   (del (size-of <part> <dim> <value-old>))
   (add (size-of <part> <dim> <value>)))))

(POLISH-WITH-BAND-SAW
 (params (<machine> <part> <attachment> <side>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> BAND-SAW)
   (is-a <attachment> SAW-BAND)
   (side-up-for-machining <dim> <side>)
   (holding-tool <machine> <attachment>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (on-table <machine> <part>)))
 (effects (
   (del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side>
                              <*surface-coating>))
   (del (surface-finish-side <part> <side>
```

```
                              <*old-sf-cond>))
   (add (surface-finish-side <part> <side>
                              POLISHED)))))


;;;**********************************************
; MACHINE: WELDER

(WELD-CYLINDERS-METAL-ARC
 (params (<machine> <part1> <part2> <part>
          <electrode> <holding-dev> <length>))
 (preconds (and
   (is-a <part1> PART)
   (is-a <part2> PART)
   (~ (same <part1> <part2>))
   (is-a <machine> METAL-ARC-WELDER)
   (is-a <electrode> ELECTRODE)
   (material-of <part1> <material1>)
   (material-of <part2> <material2>)
   (shape-of <part1> CYLINDRICAL)
   (shape-of <part2> CYLINDRICAL)
   (~ (exists (<hole>)
        (has-hole <part1> <hole> <*side> <*depth>
                  <*diam> <*loc-x> <*loc-y>)))
   (~ (exists (<hole>)
        (has-hole <part2> <hole> <*side> <*depth>
                  <*diam> <*loc-x> <*loc-y>)))
   (size-of <part1> DIAMETER <diameter1>)
   (size-of <part2> DIAMETER <diameter2>)
   (same <diameter1> <diameter2>)
   (size-of <part1> LENGTH <length1>)
   (size-of <part2> LENGTH <length2>)
   (new-size <length1> <length2> <length>)
   (new-part <part> <part1> <part2>)
   (new-material <material> <material1> <material2>)
   (holding-tool <machine> <electrode>)
   (holding <machine> <holding-dev> <part2> SIDE3)))
 (effects (
   (del (is-a <part1> PART))
   (del (is-a <part2> PART))
   (add (is-a <part> PART))
   (add (material-of <part> <material>))
   (add (size-of <part> DIAMETER <diameter1>))
   (add (size-of <part> LENGTH <length>))
   (add (surface-finish-side <part> SIDE0 SAWCUT))
   (if (surface-finish-side <part1> SIDE3 <sf31>)
       (add (surface-finish-side <part> SIDE3
                                 <sf31>)))
   (if (surface-finish-side <part2> SIDE6 <sf62>)
       (add (surface-finish-side <part> SIDE6
                                 <sf62>)))
   (del (holding <machine> <holding-dev> <part2>
                 SIDE3))
   (add (holding <machine> <holding-dev> <part>
                 SIDE3))
   (del (size-of <part1> DIAMETER <diam>))
   (del (size-of <part1> LENGTH <length1>))
   (del (size-of <part2> DIAMETER <diam>))
   (del (size-of <part2> LENGTH <length2>))
   (del (material-of <part1> <material1>))
   (del (material-of <part2> <material2>))
   (del (is-clean <part1>))
   (del (is-clean <part2>))
   (del (surface-coating-side <part1> <*sidea>
                              <*surf-coatinga>))
   (del (surface-coating-side <part2> <*sideb>
                              <*surf-coatingb>))
   (del (surface-finish-side <part1> <*sidec>
                             <*sfc>))
   (del (surface-finish-side <part2> <*sided>
                             <*sfd>)))))

(WELD-CYLINDERS-GAS
 (params (<machine> <part1> <part2> <part> <rod>
          <holding-dev> <length>))
 (preconds (and
   (is-a <part1> PART)
   (is-a <part2> PART)
   (~ (same <part1> <part2>))
   (is-a <machine> GAS-WELDER)
   (is-a <rod> WELDING-ROD)
   (is-a <torch> TORCH)
   (material-of <part1> <material1>)
   (material-of <part2> <material2>)
   (same <material1> <material2>)
   (shape-of <part1> CYLINDRICAL)
   (shape-of <part2> CYLINDRICAL)
   (~ (exists (<hole>)
        (has-hole <part1> <hole> <*side> <*depth>
                  <*diam> <*loc-x> <*loc-y>)))
   (~ (exists (<hole>)
        (has-hole <part2> <hole> <*side> <*depth>
                  <*diam> <*loc-x> <*loc-y>)))
   (size-of <part1> DIAMETER <diameter1>)
   (size-of <part2> DIAMETER <diameter2>)
   (same <diameter1> <diameter2>)
   (size-of <part1> LENGTH <length1>)
   (size-of <part2> LENGTH <length2>)
   (new-size <length1> <length2> <length>)
   (new-part <part> <part1> <part2>)
   (holding <machine> <holding-dev> <part2> SIDE3)))
 (effects (
   (del (is-a <part1> PART))
   (del (is-a <part2> PART))
   (add (is-a <part> PART))
   (add (material-of <part> <material1>))
   (add (size-of <part> DIAMETER <diameter1>))
   (add (size-of <part> LENGTH <length>))
   (add (surface-finish-side <part> SIDE0 SAWCUT))
   (if (surface-finish-side <part1> SIDE3 <sf31>)
       (add (surface-finish-side <part> SIDE3
                                 <sf31>)))
   (if (surface-finish-side <part2> SIDE6 <sf62>)
       (add (surface-finish-side <part> SIDE6
                                 <sf62>)))
   (del (holding <machine> <holding-dev> <part2>
                 SIDE3))
   (add (holding <machine> <holding-dev> <part>
                 SIDE3))
   (del (size-of <part1> DIAMETER <diam>))
```

```
   (del (size-of <part1> LENGTH <length1>))
   (del (size-of <part2> DIAMETER <diam>))
   (del (size-of <part2> LENGTH <length2>))
   (del (material-of <part1> <material1>))
   (del (material-of <part2> <material2>))
   (del (is-clean <part1>))
   (del (is-clean <part2>))
   (del (surface-coating-side <part1> <*sidea>
                              <*surface-coatinga>))
   (del (surface-coating-side <part2> <*sideb>
                              <*surface-coatingb>))
   (del (surface-finish-side <part1> <*sidec>
                             <*sfc>))
   (del (surface-finish-side <part2> <*sided>
                             <*sfd>)))))

;;;*****************************************************
; METAL-COATING

(METAL-SPRAY-COATING-CORROSION-RESISTANT
 (params (<machine> <wire> <part> <side>
          <another-machine> <holding-dev>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> ELECTRIC-ARC-SPRAY-GUN)
   (is-a <wire> SPRAYING-METAL-WIRE)
   (material-of-wire <wire> STAINLESS-STEEL)
   (~ (material-of-wire <wire> TUNGSTEN))
   (~ (material-of-wire <wire> MOLYBDENUM))
   (is-clean <part>)
   (~ (has-burrs <part>))
   (surface-coating-side <part> <side> FUSED-METAL)
   (is-of-type <another-machine> MACHINE)
   (holding <another-machine> <holding-dev> <part>
            <side>)))
 (effects (
   (add (surface-coating-side <part> <side>
                              CORROSION-RESISTANT))
   (del (surface-coating-side <part> <side>
                              FUSED-METAL)))))

(METAL-SPRAY-COATING-HEAT-RESISTANT
 (params (<machine> <wire> <part> <side>
          <another-machine> <holding-dev>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> ELECTRIC-ARC-SPRAY-GUN)
   (is-a <wire> SPRAYING-METAL-WIRE)
   (material-of-wire <wire> ZIRCONIUM-OXIDE)
   (~ (material-of-wire <wire> TUNGSTEN))
   (~ (material-of-wire <wire> MOLYBDENUM))
   (is-clean <part>)
   (~ (has-burrs <part>))
   (surface-coating-side <part> <side> FUSED-METAL)
   (is-of-type <another-machine> MACHINE)
   (holding <another-machine> <holding-dev> <part>
            <side>)))
 (effects (
   (add (surface-coating-side <part> <side>
                              HEAT-RESISTANT))
   (del (surface-coating-side <part> <side>
                              FUSED-METAL)))))


(METAL-SPRAY-COATING-WEAR-RESISTANT
 (params (<machine> <wire> <part> <side>
          <another-machine> <holding-dev>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> ELECTRIC-ARC-SPRAY-GUN)
   (is-a <wire> SPRAYING-METAL-WIRE)
   (material-of-wire <wire> ALUMINUM-OXIDE)
   (~ (material-of-wire <wire> TUNGSTEN))
   (~ (material-of-wire <wire> MOLYBDENUM))
   (is-clean <part>)
   (~ (has-burrs <part>))
   (surface-coating-side <part> <side> FUSED-METAL)
   (is-of-type <another-machine> MACHINE)
   (holding <another-machine> <holding-dev> <part>
            <side>)))
 (effects (
   (add (surface-coating-side <part> <side>
                              WEAR-RESISTANT))
   (del (surface-coating-side <part> <side>
                              FUSED-METAL)))))

(METAL-SPRAY-PREPARE
 (params (<machine> <wire> <part> <side>
          <another-machine> <holding-dev>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> ELECTRIC-ARC-SPRAY-GUN)
   (is-a <wire> SPRAYING-METAL-WIRE)
   (has-high-melting-point <wire>)
   (is-clean <part>)
   (~ (has-burrs <part>))
   (is-of-type <another-machine> MACHINE)
   (holding <another-machine> <holding-dev> <part>
            <side>)))
 (effects (
   (del (surface-coating-side <part> <side> <*s-f>))
   (add (surface-coating-side <part> <side>
                              FUSED-METAL)))))

;;;*****************************************************
; OTHER OPERATIONS

(CLEAN
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (is-available-part <part>)))
 (effects (
   (add (is-clean <part>)))))

(REMOVE-BURRS
 (params (<part> <brush>))
 (preconds (and
   (is-a <part> PART)
   (is-a <brush> BRUSH)
```

```
   (is-available-part <part>)))
 (effects (
   (del (is-clean <part>))
   (del (has-burrs <part>)))))


;;;*****************************************************
;;;*****************************************************
; operators for preparing the machines

;;;*****************************************************
; tools in machines

(PUT-TOOL-ON-MILLING-MACHINE
 (params (<machine> <attachment>))
 (preconds (and
   (is-a <machine> MILLING-MACHINE)
   (or (is-of-type <attachment> MILLING-CUTTER)
       (is-of-type <attachment> DRILL-BIT))
   (is-available-tool-holder <machine>)
   (is-available-tool <attachment>)))
 (effects (
   (add (holding-tool <machine> <attachment>)))))

(PUT-IN-DRILL-SPINDLE
 (params (<machine> <drill-bit>))
 (preconds (and
   (is-a <machine> DRILL)
   (is-of-type <drill-bit> DRILL-BIT)
   (is-available-tool-holder <machine>)
   (is-available-tool <drill-bit>)))
 (effects (
   (add (holding-tool <machine> <drill-bit>)))))

(PUT-TOOLBIT-IN-LATHE
 (params (<machine> <toolbit>))
 (preconds (and
   (is-a <machine> LATHE)
   (is-of-type <toolbit> LATHE-TOOLBIT)
   (is-available-tool-holder <machine>)
   (is-available-tool <toolbit>)))
 (effects (
   (add (holding-tool <machine> <toolbit>)))))

(PUT-CUTTING-TOOL-IN-SHAPER-OR-PLANER
 (params (<machine> <cutting-tool>))
 (preconds (and
   (or (is-a <machine> SHAPER)
       (is-a <machine> PLANER))
   (is-of-type <cutting-tool> CUTTING-TOOL)
   (is-available-tool-holder <machine>)
   (is-available-tool <cutting-tool>)))
 (effects (
   (add (holding-tool <machine> <cutting-tool>)))))

(PUT-WHEEL-IN-GRINDER
 (params (<machine> <wheel>))
 (preconds (and
   (is-a <machine> GRINDER)
   (is-a <wheel> GRINDING-WHEEL)
   (is-available-tool-holder <machine>)
   (is-available-tool <wheel>)))
 (effects (
   (add (holding-tool <machine> <wheel>)))))

(PUT-CIRCULAR-SAW-ATTACHMENT-IN-CIRCULAR-SAW
 (params (<machine> <attachment>))
 (preconds (and
   (is-a <machine> CIRCULAR-SAW)
   (is-of-type <attachment> CIRCULAR-SAW-ATTACHMENT)
   (is-available-tool-holder <machine>)
   (is-available-tool <attachment>)))
 (effects (
   (add (holding-tool <machine> <attachment>)))))

(PUT-BAND-SAW-ATTACHMENT-IN-BAND-SAW
 (params (<machine> <attachment>))
 (preconds (and
   (is-a <machine> BAND-SAW)
   (is-of-type <attachment> BAND-SAW-ATTACHMENT)
   (is-available-tool-holder <machine>)
   (is-available-tool <attachment>)))
 (effects (
   (add (holding-tool <machine> <attachment>)))))

(PUT-ELECTRODE-IN-WELDER
 (params (<machine> <electrode>))
 (preconds (and
   (is-a <machine> METAL-ARC-WELDER)
   (is-a <electrode> ELECTRODE)
   (is-available-tool-holder <machine>)
   (is-available-tool <electrode>)))
 (effects (
   (add (holding-tool <machine> <electrode>)))))

(REMOVE-TOOL-FROM-MACHINE
 (params (<machine> <tool>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-of-type <tool> MACHINE-TOOL)
   (holding-tool <machine> <tool>)))
 (effects (
   (del (holding-tool <machine> <tool>)))))

;;;*****************************************************
; holding devices in machines

(PUT-HOLDING-DEVICE-IN-MILLING-MACHINE
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-a <machine> MILLING-MACHINE)
   (or
       (is-a <holding-dev> 4-JAW-CHUCK)
       (is-a <holding-dev> V-BLOCK)
       (is-a <holding-dev> VISE)
       (is-a <holding-dev> COLLET-CHUCK)
       (is-a <holding-dev> TOE-CLAMP))
   (is-available-table <machine> <holding-dev>)
   (is-available-holding-device <holding-dev>)))
 (effects (
```

```
   (add (has-device <machine> <holding-dev>)))))

(PUT-HOLDING-DEVICE-IN-DRILL
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-a <machine> DRILL)
   (or
       (is-a <holding-dev> 4-JAW-CHUCK)
       (is-a <holding-dev> V-BLOCK)
       (is-a <holding-dev> VISE)
       (is-a <holding-dev> TOE-CLAMP))
   (is-available-table <machine> <holding-dev>)
   (is-available-holding-device <holding-dev>)))
 (effects (
   (add (has-device <machine> <holding-dev>)))))

(PUT-HOLDING-DEVICE-IN-LATHE
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-a <machine> LATHE)
   (or (is-a <holding-dev> CENTERS)
       (is-a <holding-dev> 4-JAW-CHUCK)
       (is-a <holding-dev> COLLET-CHUCK))
   (is-available-table <machine> <holding-dev>)
   (is-available-holding-device <holding-dev>)))
 (effects (
   (add (has-device <machine> <holding-dev>)))))

(PUT-HOLDING-DEVICE-IN-SHAPER
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-a <machine> SHAPER)
   (is-a <holding-dev> VISE)
   (is-available-table <machine> <holding-dev>)
   (is-available-holding-device <holding-dev>)))
 (effects (
   (add (has-device <machine> <holding-dev>)))))

(PUT-HOLDING-DEVICE-IN-PLANER
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-a <machine> PLANER)
   (is-a <holding-dev> TOE-CLAMP)
   (is-available-table <machine> <holding-dev>)
   (is-available-holding-device <holding-dev>)))
 (effects (
   (add (has-device <machine> <holding-dev>))`)))

(PUT-HOLDING-DEVICE-IN-GRINDER
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-a <machine> GRINDER)
   (or (is-a <holding-dev> MAGNETIC-CHUCK)
       (is-a <holding-dev> V-BLOCK)
       (is-a <holding-dev> VISE))
   (is-available-table <machine> <holding-dev>)
   (is-available-holding-device <holding-dev>)))
 (effects (
   (add (has-device <machine> <holding-dev>)))))

(PUT-HOLDING-DEVICE-IN-CIRCULAR-SAW
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-a <machine> CIRCULAR-SAW)
   (or (is-a <holding-dev> VISE)
       (is-a <holding-dev> V-BLOCK))
   (is-available-table <machine> <holding-dev>)
   (is-available-holding-device <holding-dev>)))
 (effects (
   (add (has-device <machine> <holding-dev>)))))

(PUT-HOLDING-DEVICE-IN-WELDER
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-of-type <machine> WELDER)
   (or (is-a <holding-dev> VISE)
       (is-a <holding-dev> TOE-CLAMP))
   (is-available-table <machine> <holding-dev>)
   (is-available-holding-device <holding-dev>)))
 (effects (
   (add (has-device <machine> <holding-dev>)))))


(REMOVE-HOLDING-DEVICE-FROM-MACHINE
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-of-type <holding-dev> HOLDING-DEVICE)
   (has-device <machine> <holding-dev>)
   (is-empty-holding-device <holding-dev>
                            <machine>)))
 (effects (
   (del (has-device <machine> <holding-dev>)))))

;;;*************************************************
; cutting fluid in machines

(ADD-SOLUBLE-OIL
 (params (<machine> <fluid>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (or (material-of <part> STEEL)
       (material-of <part> ALUMINUM))
   (is-a <fluid> SOLUBLE-OIL)))
 (effects (
   (add (has-fluid <machine> <fluid> <part>)))))

(ADD-MINERAL-OIL
 (params (<machine> <fluid>))
 (preconds  and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-a <fluid> MINERAL-OIL)
   (material-of <part> IRON)))
 (effects (
   (add (has-fluid <machine> <fluid> <part>)))))

(ADD-ANY-CUTTING-FLUID
 (params (<machine> <fluid>))
```

```
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (or (material-of <part> BRASS)
       (material-of <part> BRONZE)
       (material-of <part> COPPER))
   (is-of-type <fluid> CUTTING-FLUID)))
 (effects (
   (add (has-fluid <machine> <fluid> <part>)))))


;;;*****************************************************
;;;*****************************************************
; operators for holding parts with a device in
;  a machine

(PUT-ON-MACHINE-TABLE
 (params (<machine> <part>))
 (preconds (and
   (is-a <part> PART)
   (is-of-type <machine> MACHINE)
   (~ (is-a <machine> SHAPER))
   (is-available-part <part>)
   (is-available-machine <machine>)))
 (effects (
   (del (on-table <another-machine> <part>))
   (add (on-table <machine> <part>)))))


(PUT-ON-SHAPER-TABLE
 (params (<machine> <part>))
 (preconds (and
   (is-a <part> PART)
   (is-a <machine> SHAPER)
   (size-of-machine <machine> <shaper-size>)
   (size-of <part> LENGTH <part-size>)
   (smaller <part-size> <shaper-size>)
   (is-available-part <part>)
   (is-available-machine <machine>)))
 (effects (
   (del (on-table <another-machine> <part>))
   (add (on-table <machine> <part>)))))


(HOLD-WITH-V-BLOCK
 (params (<machine> <holding-dev> <part> <side>))
 (preconds
   (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-a <holding-dev> V-BLOCK)
   (has-device <machine> <holding-dev>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (on-table <machine> <part>)
   (shape-of <part> CYLINDRICAL)
   (same <side> SIDE0)
   (is-empty-holding-device <holding-dev> <machine>)
   (is-available-part <part>)))
 (effects (
   (del (on-table <machine> <part>))
   (add (holding-weakly <machine> <holding-dev>
                        <part> <side>)))))

(HOLD-WITH-VISE
 (params (<machine> <holding-dev> <part> <side>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-a <holding-dev> VISE)
   (has-device <machine> <holding-dev>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (on-table <machine> <part>)
   (is-empty-holding-device <holding-dev> <machine>)
   (is-available-part <part>)))
 (effects (
   (del (on-table <machine> <part>))
   (if (shape-of <part> CYLINDRICAL)
       (add (holding-weakly <machine> <holding-dev>
                            <part> <side>)))
   (if (shape-of <part> RECTANGULAR)
       (add (holding <machine> <holding-dev>
                     <part> <side>))))))

(HOLD-WITH-TOE-CLAMP
 (params (<machine> <holding-dev> <part> <side>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-a <holding-dev> TOE-CLAMP)
   (has-device <machine> <holding-dev>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (or (shape-of <part> RECTANGULAR)
       (same <side> SIDE3)
       (same <side> SIDE6))
   (on-table <machine> <part>)
   (is-empty-holding-device <holding-dev> <machine>)
   (is-available-part <part>)))
 (effects (
   (del (on-table <machine> <part>))
   (add (holding <machine> <holding-dev> <part>
                 <side>)))))

(SECURE-WITH-TOE-CLAMP
 (params (<machine> <holding-dev> <part> <side>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-a <holding-dev> TOE-CLAMP)
   (has-device <machine> <holding-dev>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (shape-of <part> CYLINDRICAL)
   (holding-weakly <machine> <another-holding-device>
                   <part> <side>)
   (is-empty-holding-device <holding-dev>
                            <machine>)))
 (effects (
```

```
   (del (on-table <machine> <part>))
   (add (holding <machine> <holding-dev> <part>
                 <side>)))))

(HOLD-WITH-CENTERS
 (params (<machine> <holding-dev> <part> <side>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-a <holding-dev> CENTERS)
   (has-device <machine> <holding-dev>)
   (has-center-holes <part>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (on-table <machine> <part>)
   (shape-of <part> CYLINDRICAL)
   (is-empty-holding-device <holding-dev> <machine>)
   (is-available-part <part>)))
 (effects (
   (del (on-table <machine> <part>))
   (add (holding <machine> <holding-dev> <part>
                 <side>)))))

(HOLD-WITH-4-JAW-CHUCK
 (params (<machine> <holding-dev> <part> <side>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-a <holding-dev> 4-JAW-CHUCK)
   (has-device <machine> <holding-dev>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (on-table <machine> <part>)
   (is-empty-holding-device <holding-dev> <machine>)
   (is-available-part <part>)))
 (effects (
   (del (on-table <machine> <part>))
   (add (holding <machine> <holding-dev> <part>
                 <side>)))))

(HOLD-WITH-COLLET-CHUCK
 (params (<machine> <holding-dev> <part> <side>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-a <holding-dev> COLLET-CHUCK)
   (has-device <machine> <holding-dev>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (on-table <machine> <part>)
   (shape-of <part> CYLINDRICAL)
   (is-empty-holding-device <holding-dev> <machine>)
   (is-available-part <part>)))
 (effects (
   (del (on-table <machine> <part>))
   (add (holding <machine> <holding-dev> <part>
                 <side>)))))

(HOLD-WITH-MAGNETIC-CHUCK
 (params (<machine> <holding-dev> <part> <side>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-a <holding-dev> MAGNETIC-CHUCK)
   (has-device <machine> <holding-dev>)
   (~ (has-burrs <part>))
   (is-clean <part>)
   (on-table <machine> <part>)
   (is-empty-holding-device <holding-dev> <machine>)
   (is-available-part <part>)))
 (effects (
   (del (on-table <machine> <part>))
   (add (holding <machine> <holding-dev> <part>
                 <side>)))))

(RELEASE-FROM-HOLDING-DEVICE
 (params (<machine> <holding-dev> <part> <side>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-of-type <holding-dev> HOLDING-DEVICE)
   (holding <machine> <holding-dev> <part> <side>)))
 (effects (
   (del (holding <machine> <holding-dev> <part>
                 <side>))
   (add (on-table <machine> <part>)))))

(RELEASE-FROM-HOLDING-DEVICE-WEAK
 (params (<machine> <holding-dev> <part> <side>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-a <part> PART)
   (is-of-type <holding-dev> HOLDING-DEVICE)
   (holding-weakly <machine> <holding-dev> <part>
                   <side>)))
 (effects (
   (del (holding-weakly <machine> <holding-dev>
                        <part> <side>))
   (add (on-table <machine> <part>)))))
```

B.2.2 Inference Rules

```
(HAS-CENTER-HOLES
 (params (<part> <x2> <y2>))
 (preconds (and
   (is-a <part> PART)
   (or (and
         (shape-of <part> RECTANGULAR)
         (size-of <part> WIDTH <x>)
         (size-of <part> HEIGHT <y>))
     (and
         (shape-of <part> CYLINDRICAL)
         (size-of <part> DIAMETER <x>)
         (size-of <part> DIAMETER <y>)))
   (half-of <x> <x2>)
   (half-of <y> <y2>)
   (has-center-hole <part> CENTER-HOLE-SIDE3 SIDE3
                    <x2> <y2>)
```

```
   (is-countersinked <part> CENTER-HOLE-SIDE3 SIDE3
                    1/8 1/16 <x2> <y2> 60)
   (has-center-hole <part> CENTER-HOLE-SIDE6 SIDE6
                   <x2> <y2>)
   (is-countersinked <part> CENTER-HOLE-SIDE6 SIDE6
                    1/8 1/16 <x2> <y2> 60)))
 (effects (
   (add (has-center-holes <part>)))))

;;;*****************************************************

(SIDE-UP-FOR-MACHINING-LENGTH
 (params (<side>))
 (preconds (and
   (same <dim> LENGTH)
   (or (same <side> SIDE3)
     (same <side> SIDE6))))
 (effects (
   (add (side-up-for-machining <dim> <side>)))))

(SIDE-UP-FOR-MACHINING-WIDTH
 (params (<side>))
 (preconds (and
   (same <dim> WIDTH)
   (or (same <side> SIDE2)
     (same <side> SIDE5))))
 (effects (
   (add (side-up-for-machining <dim> <side>)))))

(SIDE-UP-FOR-MACHINING-HEIGHT
 (params (<side>))
 (preconds (and
   (same <dim> HEIGHT)
   (or (same <side> SIDE1)
     (same <side> SIDE4))))
 (effects (
   (add (side-up-for-machining <dim> <side>)))))

(SIDE-UP-FOR-MACHINING-DIAMETER
 (params (<side>))
 (preconds (and
   (same <dim> DIAMETER)
   (or (and
         (shape-of <part> RECTANGULAR)
         (same <side> SIDE1))
     (and
         (shape-of <part> CYLINDRICAL)
         (same <side> SIDE0)))))
 (effects (
   (add (side-up-for-machining <dim> <side>)))))

;;;*****************************************************
; inference rules for availability

(MACHINE-AVAILABLE
 (params (<machine>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (- (exists (<other-part>)
         (on-table <machine> <other-part>))))
 (effects (
   (add (is-available-machine <machine>)))))

(TOOL-HOLDER-AVAILABLE
 (params (<machine>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (- (exists (<tool>)
         (holding-tool <machine> <tool>)))))
 (effects (
   (add (is-available-tool-holder <machine>)))))

(TOOL-AVAILABLE
 (params (<tool>))
 (preconds (and
   (is-of-type <tool> MACHINE-TOOL)
   (- (exists (<machine>)
         (holding-tool <machine> <tool>)))))
 (effects ( (add (is-available-tool <tool>)))))

(TABLE-AVAILABLE
 (params (<machine>))
 (preconds (and
   (is-of-type <machine> MACHINE)
   (is-of-type <holding-dev> HOLDING-DEVICE)
   (or
     (- (exists (<another-holding-device>)
           (has-device <machine>
                       <another-holding-device>)))
     (is-a <holding-dev> TOE-CLAMP))))
 (effects (
   (add (is-available-table <machine>
                            <holding-dev>)))))

(HOLDING-DEVICE-AVAILABLE
 (params (<machine> <holding-dev>))
 (preconds (and
   (is-of-type <holding-dev> HOLDING-DEVICE)
   (- (exists (<machine>)
         (has-device <machine> <holding-dev>)))))
 (effects (
   (add (is-available-holding-device
                                 <holding-dev>)))))

(PART-AVAILABLE
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (- (exists (<machine>)
         (holding-weakly <machine> <*holding-dev>
                         <part> <*side>)))
   (- (exists (<machine>)
         (holding <machine> <*another-holding-dev>
                  <part> <*side>)))))
 (effects (
   (add (is-available-part <part>)))))

(HOLDING-DEVICE-EMPTY
 (params (<machine> <holding-dev>))
 (preconds (and
```

```
   (is-of-type <machine> MACHINE)
   (is-of-type <holding-dev> HOLDING-DEVICE)
   (~ (exists (<part>)
         (holding-weakly <machine> <holding-dev>
                         <part> <side>)))
   (~ (exists (<another-part>)
         (holding <machine> <holding-dev>
                  <another-part> <side>)))))
 (effects (
   (add (is-empty-holding-device <holding-dev>
                                 <machine>)))))

;;;*****************************************************
; inference rules for shape

(IS-RECTANGULAR
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (size-of <part> LENGTH <l>)
   (size-of <part> WIDTH <w>)
   (size-of <part> HEIGHT <h>)))
 (effects (
   (add (shape-of <part> RECTANGULAR)))))

(IS-CYLINDRICAL
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (size-of <part> LENGTH <l>)
   (size-of <part> DIAMETER <d>)))
 (effects (
   (add (shape-of <part> CYLINDRICAL)))))

(ARE-SIDES-OF-RECTANGULAR-PART
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> RECTANGULAR)))
 (effects (
   (add (side-of <part> SIDE1))
   (add (side-of <part> SIDE2))
   (add (side-of <part> SIDE3))
   (add (side-of <part> SIDE4))
   (add (side-of <part> SIDE5))
   (add (side-of <part> SIDE6)))))


(ARE-SIDES-OF-CYLINDRICAL-PART
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> CYLINDRICAL)))
 (effects (
   (add (side-of <part> SIDE0))
   (add (side-of <part> SIDE3))
   (add (side-of <part> SIDE6)))))

;;;*****************************************************

; inference rules for surface finish

(IS-MACHINED-SURFACE-QUALITY
 (params (<part> <side>))
 (preconds (and
   (is-a <part> PART)
   (or
     (surface-finish-side <part> <side> ROUGH-MILL)
     (surface-finish-side <part> <side> ROUGH-TURN)
     (surface-finish-side <part> <side> ROUGH-SHAPED)
     (surface-finish-side <part> <side> ROUGH-PLANED)
     (surface-finish-side <part> <side> FINISH-PLANED)
     (surface-finish-side <part> <side> COLD-ROLLED)
     (surface-finish-side <part> <side> FINISH-MILL)
     (surface-finish-side <part> <side> FINISH-TURN)
     (surface-finish-quality-side <part> <side>
                                  GROUND))))
 (effects (
   (add (surface-finish-quality-side <part> <side>
                                     MACHINED)))))

(IS-GROUND-SURFACE-QUALITY
 (params (<part> <side>))
 (preconds (and
   (is-a <part> PART)
   (or
     (surface-finish-side <part> <side> ROUGH-GRIND)
     (surface-finish-side <part> <side>
                          FINISH-GRIND))))
 (effects (
   (add (surface-finish-quality-side <part> <side>
                                     GROUND)))))


(HAS-SURFACE-FINISH-RECTANGULAR-PART
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> RECTANGULAR)
   (surface-finish-side <part> SIDE1 <surface-finish>)
   (surface-finish-side <part> SIDE2 <surface-finish>)
   (surface-finish-side <part> SIDE3 <surface-finish>)
   (surface-finish-side <part> SIDE4 <surface-finish>)
   (surface-finish-side <part> SIDE5 <surface-finish>)
   (surface-finish-side <part> SIDE6 <surface-finish>)))
 (effects (
   (add (surface-finish <part> <surface-finish>)))))

(HAS-SURFACE-FINISH-CYLINDRICAL-PART
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> CYLINDRICAL)
   (surface-finish-side <part> SIDE0 <surface-finish>)
   (surface-finish-side <part> SIDE3 <surface-finish>)
   (surface-finish-side <part> SIDE6 <surface-finish>)))
 (effects (
   (add (surface-finish <part> <surface-finish>)))))

(HAVE-SURFACE-FINISH-RECTANGULAR-PART-SIDES
```

```
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> RECTANGULAR)
   (surface-finish <part> <surf-fin>)))
 (effects (
   (add (surface-finish-side <part> SIDE1 <surf-fin>))
   (add (surface-finish-side <part> SIDE2 <surf-fin>))
   (add (surface-finish-side <part> SIDE3 <surf-fin>))
   (add (surface-finish-side <part> SIDE4 <surf-fin>))
   (add (surface-finish-side <part> SIDE5 <surf-fin>))
   (add (surface-finish-side <part> SIDE6 <surf-fin>)))))

(HAVE-SURFACE-FINISH-CYLINDRICAL-PART-SIDES
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> CYLINDRICAL)
   (surface-finish <part> <surf-fin>)))
 (effects (
   (add (surface-finish-side <part> SIDE0 <surf-fin>))
   (add (surface-finish-side <part> SIDE3 <surf-fin>))
   (add (surface-finish-side <part> SIDE6 <surf-fin>)))))

;;;*****************************************************
;  inference rules for surface-coating


(HAS-SURFACE-COATING-RECTANGULAR-PART
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> RECTANGULAR)
   (surface-coating-side <part> SIDE1 <surf-coat>)
   (surface-coating-side <part> SIDE2 <surf-coat>)
   (surface-coating-side <part> SIDE3 <surf-coat>)
   (surface-coating-side <part> SIDE4 <surf-coat>)
   (surface-coating-side <part> SIDE5 <surf-coat>)
   (surface-coating-side <part> SIDE6 <surf-coat>)))
 (effects (
   (add (surface-coating <part> <surf-coat>)))))

(HAS-SURFACE-COATING-CYLINDRICAL-PART
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> CYLINDRICAL)
   (surface-coating-side <part> SIDE0 <surf-coat>)
   (surface-coating-side <part> SIDE3 <surf-coat>)
   (surface-coating-side <part> SIDE6 <surf-coat>)))
 (effects (
   (add (surface-coating <part> <surf-coat>)))))


(HAVE-SURFACE-COATING-RECTANGULAR-PART-SIDES
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> RECTANGULAR)
   (surface-coating <part> <surf-coat>)))
 (effects (
   (add (surface-coating-side <part> SIDE1 <surf-coat>))
   (add (surface-coating-side <part> SIDE2 <surf-coat>))
   (add (surface-coating-side <part> SIDE3 <surf-coat>))
   (add (surface-coating-side <part> SIDE4 <surf-coat>))
   (add (surface-coating-side <part> SIDE5 <surf-coat>))
   (add (surface-coating-side <part> SIDE6 <surf-coat>)))))

(HAVE-SURFACE-COATING-CYLINDRICAL-PART-SIDES
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (shape-of <part> CYLINDRICAL)
   (surface-coating <part> <surf-coat>)))
 (effects (
   (add (surface-coating-side <part> SIDE0 <surf-coat>))
   (add (surface-coating-side <part> SIDE3 <surf-coat>))
   (add (surface-coating-side <part> SIDE6 <surf-coat>)))))

;;;*****************************************************
(MATERIAL-FERROUS
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (or
       (material-of <part> STEEL)
       (material-of <part> IRON))))
 (effects (
   (add (alloy-of <part> FERROUS)))))

(MATERIAL-NON-FERROUS
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (or
       (material-of <part> BRASS)
       (material-of <part> COPPER)
       (material-of <part> BRONZE))))
 (effects (
   (add (alloy-of <part> NON-FERROUS)))))

(HARDNESS-OF-MATERIAL-SOFT
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (or
       (material-of <part> ALUMINUM)
       (alloy-of <part> NON-FERROUS))))
 (effects (
   (add (hardness-of <part> SOFT)))))

(HARDNESS-OF-MATERIAL-HARD
 (params (<part>))
 (preconds (and
   (is-a <part> PART)
   (alloy-of <part> FERROUS)))
 (effects (
   (add (hardness-of <part> HARD)))))
```

```
(HIGH-MELTING-POINT
 (params (<wire>))
 (preconds (and
   (is-a <wire> SPRAYING-METAL-WIRE)
   (or
      (material-of-wire <wire> TUNGSTEN)
      (material-of-wire <wire> MOLYBDENUM))))
 (effects (
   (add (has-high-melting-point <wire>)))))


;;;**************************************************
; inference rules for types

(IS-MACHINE
 (params (<machine>))
 (preconds
   (or
       (is-a <machine> DRILL)
       (is-a <machine> LATHE)
       (is-a <machine> SHAPER)
       (is-a <machine> PLANER)
       (is-a <machine> GRINDER)
       (is-a <machine> BAND-SAW)
       (is-a <machine> CIRCULAR-SAW)
       (is-a <machine> MILLING-MACHINE)
       (is-of-type <machine> WELDER)))
 (effects (
   (add (is-of-type <machine> MACHINE)))))

(IS-WELDER
 (params (<machine>))
 (preconds
   (or
       (is-a <machine> METAL-ARC-WELDER)
       (is-a <machine> GAS-WELDER)))
 (effects (
   (add (is-of-type <machine> WELDER)))))

(IS-TOOL
 (params (<tool>))
 (preconds
   (or
       (is-of-type <tool> MACHINE-TOOL)
       (is-of-type <tool> OPERATOR-TOOL)))
 (effects (
   (add (is-of-type <tool> TOOL)))))

(IS-MACHINE-TOOL
 (params (<attachment>))
 (preconds
    (or
        (is-of-type <attachment> DRILL-BIT)
        (is-of-type <attachment> LATHE-TOOLBIT)
        (is-of-type <attachment> CUTTING-TOOL)
        (is-a <attachment> GRINDING-WHEEL)
        (is-of-type <attachment> BAND-SAW-ATTACHMENT)
        (is-of-type <attachment>
                    CIRCULAR-SAW-ATTACHMENT)
        (is-of-type <attachment> MILLING-CUTTER)
        (is-a <attachment> ELECTRODE)))
 (effects (
   (add (is-of-type <attachment> MACHINE-TOOL)))))

(IS-DRILL-BIT
 (params (<drill-bit>))
 (preconds
   (or
       (is-a <drill-bit> SPOT-DRILL)
       (is-a <drill-bit> CENTER-DRILL)
       (is-a <drill-bit> TWIST-DRILL)
       (is-a <drill-bit> STRAIGHT-FLUTED-DRILL)
       (is-a <drill-bit> HIGH-HELIX-DRILL)
       (is-a <drill-bit> OIL-HOLE-DRILL)
       (is-a <drill-bit> GUN-DRILL)
       (is-a <drill-bit> CORE-DRILL)
       (is-a <drill-bit> TAP)
       (is-a <drill-bit> COUNTERSINK)
       (is-a <drill-bit> COUNTERBORE)
       (is-a <drill-bit> REAMER)))
 (effects (
   (add (is-of-type <drill-bit> DRILL-BIT)))))

(IS-LATHE-TOOLBIT
 (params (<toolbit>))
 (preconds
   (or
       (is-a <toolbit> ROUGH-TOOLBIT)
       (is-a <toolbit> FINISH-TOOLBIT)
       (is-a <toolbit> V-THREAD)
       (is-a <toolbit> KNURL)))
 (effects (
   (add (is-of-type <toolbit> LATHE-TOOLBIT)))))

(IS-CUTTING-TOOL
 (params (<cutting-tool>))
 (preconds
   (or
       (is-a <cutting-tool> ROUGHING-CUTTING-TOOL)
       (is-a <cutting-tool> FINISHING-CUTTING-TOOL)))
 (effects (
   (add (is-of-type <cutting-tool> CUTTING-TOOL)))))

(IS-CIRCULAR-SAW-ATTACHMENT
 (params (<attachment>))
 (preconds
   (or
       (is-a <attachment> COLD-SAW)
       (is-a <attachment> FRICTION-SAW)))
 (effects (
   (add (is-of-type <attachment>
                    CIRCULAR-SAW-ATTACHMENT)))))


(IS-BAND-SAW-ATTACHMENT
 (params (<attachment>))
 (preconds
   (or
       (is-a <attachment> SAW-BAND)
```

```
      (is-a <attachment> BAND-FILE)))
 (effects (
   (add (is-of-type <attachment>
                    BAND-SAW-ATTACHMENT)))))

(IS-MILLING-CUTTER
 (params (<milling-cutter>))
 (preconds
   (or
       (is-a <milling-cutter> PLAIN-MILL)
       (is-a <milling-cutter> END-MILL)))
 (effects (
   (add (is-of-type <milling-cutter>
                    MILLING-CUTTER)))))

(IS-OPERATOR-TOOL
 (params (<tool>))
 (preconds
   (or
       (is-a <tool> LATHE-FILE)
       (is-a <tool> ABRASIVE-CLOTH)
       (is-a <tool> TORCH)
       (is-a <tool> WELDING-ROD)
       (is-a <tool> SPRAYING-METAL-WIRE)
       (is-a <tool> BRUSH)))
 (effects (
   (add (is-of-type <tool> OPERATOR-TOOL)))))

(IS-CUTTING-FLUID
 (params (<cutting-fluid>))
 (preconds
   (or
       (is-a <cutting-fluid> SOLUBLE-OIL)
       (is-a <cutting-fluid> MINERAL-OIL)))
 (effects (
   (add (is-of-type <cutting-fluid> CUTTING-FLUID)))))

(IS-HOLDING-DEVICE
 (params (<holding-dev>))
 (preconds
   (or
       (is-a <holding-dev> V-BLOCK)
       (is-a <holding-dev> VISE)
       (is-a <holding-dev> TOE-CLAMP)
       (is-a <holding-dev> CENTERS)
       (is-a <holding-dev> 4-JAW-CHUCK)
       (is-a <holding-dev> COLLET-CHUCK)
       (is-a <holding-dev> MAGNETIC-CHUCK)))
 (effects (
   (add (is-of-type <holding-dev> HOLDING-DEVICE)))))
```

B.2.3 Functions

```
(defun same (x y)
   (cond ((is-variable x)
          (return-binding x y))
         ((is-variable y)
          (return-binding y x))
         (t
          (equal x y))))

(defun half-of (x y)
   (cond ((is-variable x)
          'no-match-attempted)
         ((is-variable y)
          (return-binding y (/ x 2)))
         ((= (/ x 2) y) t )))

(defun smaller (x y)
   (cond ((is-variable x)
          (if (> (- y .5) 0)
              (return-binding x (- y .5))))
         ((is-variable y)
          (return-binding y (+ x .5)))
         ((< x y) t )))

(defun smaller-than-2in (x y)
   (cond ((is-variable x)
          'no-match-attempted)
         ((is-variable y)
          'no-match-attempted)
         (t
          (<= (- x y) 2))))


; Function used for finish operations.
;
(defun finishing-size (x y)
   (cond ((and (is-variable x)
               (is-variable y))
          'no-match-attempted)
         ((is-variable x)
          (return-binding x (+ y 0.002)))
         ((is-variable y)
          (if (> (- x 0.002) 0)
              (return-binding y (- x 0.002))))
         (t
          (<= (abs (- x y)) 0.003))))

; Functions for generating new values when two
;   parts are welded together.

(defun new-size (d1 d2 d)
   (cond ((is-variable d1)
          'no-match-attempted)
         ((is-variable d2)
          'no-match-attempted)
         ((is-variable d)
          (return-binding d (+ d1 d2)))
         (t
          (= d (+ d1 d2)))))

(defun new-part (part part1 part2)
   (cond ((is-variable part)
          (return-binding part (new-name part1 part2)))
         (t)))

(defun new-material (material material1 material2)
```

```
 (if (is-variable material)
   (cond ((same material1 material2)
          (return-binding material material1))
         (t
          (return-binding
                    material
                    (new-name material1 material2))))
   t))

(defun new-name (name1 name2)
  (intern (concatenate 'string
                       (symbol-name name1)
                       "-"
                       (symbol-name name2))))

;;;**********c**********************************************

; Return a PRODIGY binding: variable var is bound
;  to value val.
(defun return-binding (var val)
  (list (list (list var val))))
```

B.3 Incomplete Domains

The 16 preconditions missing in D'_{prec10} are the following:

operator	*precondition*
drill-with-high-helix-drill	(holding-tool <machine> <drill-bit>)
drill-with-gun-drill	(has-spot <part> <hole> <side> <loc-x> <loc-y>)
drill-with-center-drill	(has-spot <part> <hole> <side> <loc-x> <loc-y>)
tap	(holding-tool <machine> <drill-bit>)
tap	(is-clean <part>)
counterbore	(holding-tool <machine> <drill-bit>)
ream	(has-fluid <machine> <fluid> <part>)
drill-with-twist-drill-in-milling-machine	(holding-tool <machine> <drill-bit>)
make-knurl-with-lathe	(holding-tool <machine> <toolbit>)
make-knurl-with-lathe	(- (has-burrs <part>))
finish-shape	(- (has-burrs <part>))
cut-with-circular-friction-saw	(holding-tool <machine> <attachment>)
cut-with-band-saw	(- (has-burrs <part>))
hold-with-v-block	(on-table <machine> <part>)
hold-with-centers	(on-table <machine> <part>)
hold-with-magnetic-chuck	(- (has-burrs <part>))

The 44 preconditions missing in D'_{prec30} are the following:

operator	*precondition*
drill-with-twist-drill	(holding-tool <machine> <drill-bit>)
drill-with-high-helix-drill	(has-fluid <machine> <fluid> <part>)
drill-with-high-helix-drill	(holding-tool <machine> <drill-bit>)
drill-with-straight-fluted-drill	(has-spot <part> <hole> <side> <loc-x> <loc-y>)
drill-with-oil-hole-drill	(has-fluid <machine> <fluid> <part>)
drill-with-gun-drill	(holding-tool <machine> <drill-bit>)
tap	(holding-tool <machine> <drill-bit>)
countersink	(has-hole <part> <hole> <side> <depth> <diam> <loc-x> <loc-y>)
countersink	(- (has-burrs <part>))
counterbore	(holding-tool <machine> <drill-bit>)
counterbore	(- (has-burrs <part>))
side-mill	(holding-tool <machine> <milling-cutter>)
finish-turn	(is-clean <part>)
make-thread-with-lathe	(is-clean <part>)
make-knurl-with-lathe	(holding-tool <machine> <toolbit>)
make-knurl-with-lathe	(is-clean <part>)
polish-with-lathe	(material-of-abrasive-cloth <cloth> EMERY)
finish-shape	(holding-tool <machine> <cutting-tool>)
finish-shape	(- (has-burrs <part>))
finish-shape-with-planer	(holding-tool <machine> <cutting-tool>)
rough-grind-with-hard-wheel	(- (material-of <part> BRONZE))
rough-grind-with-hard-wheel	(- (material-of <part> COPPER))
rough-grind-with-hard-wheel	(holding-tool <machine> <wheel>)
rough-grind-with-soft-wheel	(hardness-of-wheel <wheel> SOFT)
rough-grind-with-soft-wheel	(grit-of-wheel <wheel> COARSE-GRIT)
finish-grind-with-hard-wheel	(has-fluid <machine> <fluid> <part>)
finish-grind-with-hard-wheel	(grit-of-wheel <wheel> FINE-GRIT)
finish-grind-with-soft-wheel	(grit-of-wheel <wheel> FINE-GRIT)
finish-grind-with-soft-wheel	(is-clean <part>)
cut-with-circular-friction-saw	(holding-tool <machine> <attachment>)
polish-with-band-saw	(is-clean <part>)
metal-spray-coating-wear-resistant	(- (material-of-wire <wire> TUNGSTEN))
metal-spray-coating-wear-resistant	(- (material-of-wire <wire> MOLYBDENUM))
metal-spray-coating-wear-resistant	(is-clean <part>)
metal-spray-prepare	(is-clean <part>)
hold-with-v-block	(is-clean <part>)
hold-with-v-block	(on-table <machine> <part>)
hold-with-toe-clamp	(is-clean <part>)
secure-with-toe-clamp	(is-clean <part>)
hold-with-centers	(- (has-burrs <part>))
hold-with-centers	(on-table <machine> <part>)
hold-with-collet-chuck	(has-device <machine> <holding-device>)
hold-with-collet-chuck	(- (has-burrs <part>))
hold-with-magnetic-chuck	(is-clean <part>)

B.4 Problem Sets

The problems used to train and test EXPO were generated randomly as follows. A random number of goals is chosen between 1 and 9. The goals are chosen from a list of machining goals that include size, surface finish, surface coating, and holes. Then a start state is generated from a machine shop description that contains a set of machines, tools, holding devices, and raw materials. The solutions of the problems average one hundred steps.

EXPO was tested with two different training sets of 100 problems each. Two test sets of 20 problems each were used.

B.5 Tables of Results

This section presents the numerical results that were used for the graphs in Chapter 6.

B.5.1 Missing 10% of the Preconditions

The following tables show the numerical results that are summarized in Figure 6.3 (10% incompleteness):

number of	*failures found*	
training problems	*training set 1*	*training set 2*
0	0	0
10	6	7
20	6	7
30	9	7
40	9	10
50	9	10
60	0	0
70	0	0
80	0	0
90	0	0
100	0	0

number of training problems	*successfully executed solutions*			
	training set 1		*training set 2*	
	test set 1	*test set 2*	*test set 1*	*test set 2*
0	5	5	5	6
10	15	15	17	19
20	15	15	17	19
30	15	15	18	20
40	15	20	20	20
50	15	20	20	20
60	15	20	20	20
70	15	20	20	20
80	15	20	20	20
90	15	20	20	20
100	15	20	20	20

New preconditions for D'_{prec10} were learned by EXPO with the first training set in the following order:

1. (~ (**has-burrs** <**part**>)) of operator cut-with-band-saw
2. (**holding-tool** <**machine**> <**drill-bit**>) of operator drill-with-high-helix-drill
3. (**holding-tool** <**machine**> <**drill-bit**>) of operator tap
4. (**is-clean** <**part**>) of operator tap
5. (**has-fluid** <**machine**> <**fluid**> <**part**>) of operator ream
6. (**holding-tool** <**machine**> <**attachment**>) of operator cut-with-circular-friction-saw
7. (**on-table** <**machine**> <**part**>) of operator hold-with-v-block
8. (~ (**has-burrs** <**part**>)) of operator hold-with-magnetic-chuck
9. (~ (**has-burrs** <**part**>)) of operator finish-shape

New preconditions for D'_{prec10} were learned by EXPO with the second training set in the following order:

1. (**holding-tool** <**machine**> <**drill-bit**>) of operator drill-with-high-helix-drill

2. (holding-tool <machine> <drill-bit>) of operator tap
3. (is-clean <part>) of operator tap
4. (holding-tool <machine> <drill-bit>) of operator counterbore
5. (has-fluid <machine> <fluid> <part>) of operator ream
6. (holding-tool <machine> <attachment>) of operator cut-with-circular-friction-saw
7. (~ (has-burrs <part>)) of operator cut-with-band-saw
8. (on-table <machine> <part>) of operator hold-with-v-block
9. (~ (has-burrs <part>)) of operator hold-with-magnetic-chuck
10. (~ (has-burrs <part>)) of operator finish-shape

B.5.2 Missing 30% of the Preconditions

The following tables show the numerical results that are summarized in Figure 6.4 (30% incompleteness):

number of	*failures found*	
training problems	*training set 1*	*training set 2*
0	0	0
10	19	16
20	2	2
30	4	4
40	2	5
50	0	4
60	1	0
70	0	0
80	0	1
90	0	0
100	0	0

number of training problems	*successfully executed solutions*			
	training set 1		*training set 2*	
	test set 1	*test set 2*	*test set 1*	*test set 2*
0	1	2	1	1
10	3	13	2	14
20	9	15	7	14
30	9	18	8	15
40	11	18	13	17
50	19	18	17	17
60	19	18	17	17
70	19	18	17	17
80	19	19	17	19
90	19	19	17	19
100	19	19	17	19

New preconditions for D'_{prec30} were learned by EXPO with the first training set in the following order:

1. (**is-clean <part>**) of operator polish-with-band-saw
2. (**is-clean <part>**) of operator hold-with-toe-clamp
3. (**holding-tool <machine> <drill-bit>**) of operator drill-with-twist-drill
4. (**has-fluid <machine> <fluid> <part>**) of operator drill-with-high-helix-drill
5. (**holding-tool <machine> <drill-bit>**) of operator drill-with-high-helix-drill
6. (**has-hole <part> <hole> <side> <depth> <diam> <loc-x> <loc-y>**) of operator countersink
7. (**~ (has-burrs <part>)**) of operator countersink
8. (**holding-tool <machine> <drill-bit>**) of operator counterbore
9. (**~ (has-burrs <part>)**) of operator counterbore
10. (**hardness-of-wheel <wheel> SOFT**) of operator rough-grind-with-soft-wheel
11. (**is-clean <part>**) of operator secure-with-toe-clamp
12. (**holding-tool <machine> <drill-bit>**) of operator tap

13. (holding-tool <machine> <attachment>) of operator cut-with-circular-friction-saw
14. (holding-tool <machine> <wheel>) of operator rough-grind-with-hard-wheel
15. (is-clean <part>) of operator metal-spray-prepare
16. (has-fluid <machine> <fluid> <part>) of operator drill-with-oil-hole-drill
17. (is-clean <part>) of operator hold-with-v-block
18. (on-table <machine> <part>) of operator hold-with-v-block
19. (is-clean <part>) of operator hold-with-magnetic-chuck
20. (holding-tool <machine> <cutting-tool>) of operator finish-shape
21. (~ (has-burrs <part>)) of operator finish-shape
22. (grit-of-wheel <wheel> FINE-GRIT) of operator finish-grind-with-soft-wheel
23. (is-clean <part>) of operator finish-grind-with-soft-wheel
24. (holding-tool <machine> <milling-cutter>) of operator side-mill

New preconditions for D'_{prec30} were learned by EXPO with the second training set in the following order:

1. (holding-tool <machine> <drill-bit>) of operator drill-with-twist-drill
2. (has-fluid <machine> <fluid> <part>) of operator drill-with-high-helix-drill
3. (holding-tool <machine> <drill-bit>) of operator tap
4. (holding-tool <machine> <drill-bit>) of operator drill-with-high-helix-drill
5. (holding-tool <machine> <drill-bit>) of operator counterbore
6. (~ (has-burrs <part>)) of operator counterbore
7. (is-clean <part>) of operator metal-spray-prepare
8. (has-fluid <machine> <fluid> <part>) of operator finish-grind-with-hard-wheel
9. (grit-of-wheel <wheel> FINE-GRIT) of operator finish-grind-with-hard-wheel

10. (holding-tool <machine> <wheel>) of operator rough-grind-with-hard-wheel
11. (is-clean <part>) of operator hold-with-toe-clamp
12. (has-hole <part> <hole> <side> <depth> <diam> <loc-x> <loc-y>) of operator countersink
13. (~ (has-burrs <part>)) of operator countersink
14. (is-clean <part>) of operator secure-with-toe-clamp
15. (holding-tool <machine> <attachment>) of operator cut-with-circular-friction-saw
16. (~ (material-of-wire <wire> TUNGSTEN)) of operator metal-spray-coating-wear-resistant
17. (~ (material-of-wire <wire> MOLYBDENUM)) of operator metal-spray-coating-wear-resistant
18. (is-clean <part>) of operator metal-spray-coating-wear-resistant
19. (has-fluid <machine> <fluid> <part>) of operator drill-with-oil-hole-drill
20. (is-clean <part>) of operator hold-with-v-block
21. (on-table <machine> <part>) of operator hold-with-v-block
22. (is-clean <part>) of operator hold-with-magnetic-chuck
23. (holding-tool <machine> <cutting-tool>) of operator finish-shape
24. (~ (has-burrs <part>)) of operator finish-shape
25. (grit-of-wheel <wheel> FINE-GRIT) of operator finish-grind-with-soft-wheel
26. (is-clean <part>) of operator finish-grind-with-soft-wheel
27. (holding-tool <machine> <milling-cutter>) of operator side-mill

Appendix C

EXPO's Implementation of Experimentation Policies

The control rules below implement EXPO's experimentation strategies for PRODIGY, as described in Section 4.4.1. The meta predicates that are used by these control rules are described afterwards.

C.1 Policies

Search Depth and Plan Length

```
(AVOID-DEEP-NODES
    (lhs (and (primary-candidate-node  <node>)
              (below-exp-depth-limit <node>)))
    (rhs (reject node <node>)))

(AVOID-LONG-PLANS
    (lhs (and (primary-candidate-node <node>)
              (current-plan <node> <plan>)
              (is-too-long-plan <plan>)))
    (rhs (reject node <node>)))

(PREFER-SHORT-PLANS
    (priority 10)
    (lhs (and (candidate-node <node1>)
              (candidate-node <node2>)
              (node-pref-not-cached <node1> <node2>)
              (current-plan <node1> <pl1>)
              (current-plan <node2> <pl2>)
              (is-longer <pl2> <pl1>)))
    (rhs (prefer node <node1> <node2>)))

(PREFER-PLANS-WITH-FEWER-STATE-CHANGES
```

```
    (priority 10)
    (lhs (and (candidate-node <node1>)
              (candidate-node <node2>)
              (node-pref-not-cached <node1> <node2>)
              (current-state <node1> <state1>)
              (current-state <node2> <state2>)
              (has-fewer-changes <state1> <state2>)))
    (rhs (prefer node <node1> <node2>)))
```

Goal Interactions

```
(SUPPORT-TOP-GOAL-CONCORD
    (priority 10)
     (lhs (and (candidate-node <node1>)
               (candidate-node <node2>)
               (node-pref-not-cached <node1> <node2>)
               (current-goal <node1> <goal1>)
               (current-goal <node2> <goal2>)
               (does-top-goal-concord <goal1>)
               (not-does-top-goal-concord <goal2>)))
     (rhs (prefer node <node1> <node2>)))

(AVOID-TOP-PROTECTION-VIOLATION
    (priority 10)
    (lhs (and (candidate-node <node1>)
              (candidate-node <node2>)
              (node-pref-not-cached <node1> <node2>)
              (current-goal <node1> <goal1>)
              (current-goal <node2> <goal2>)
              (does-top-protection-violation <goal2>)
              (not-does-top-protection-violation <goal1>)))
     (rhs (prefer node <node1> <node2>)))

(AVOID-TOP-PREREQUISITE-VIOLATION
    (priority 10)
    (lhs (and (candidate-node <node1>)
              (candidate-node <node2>)
              (node-pref-not-cached <node1> <node2>)
              (current-goal <node1> <goal1>)
              (current-goal <node2> <goal2>)
              (does-top-prerequisite-violation <goal2>)
              (not-does-top-prerequisite-violation <goal1>)))
     (rhs (prefer node <node1> <node2>)))
```

Operators

```
(REJECT-IRREVERSIBLE-OPS
    (lhs (and (current-node <node>)
              (candidate-op <node> <op>)
              (not-is-reversible <op>)))
    (rhs (reject operator <op>)))

(PREFER-OPS-WITH-FEWER-STATE-CHANGES
    (priority 10)
    (lhs (and (current-node <node>)
              (candidate-op <node> <op1>)
```

```
            (candidate-op <node> <op2>)
            (are-effects-of <op1> <eff1>)
            (are-effects-of <op2> <eff2>)
            (is-longer <eff2> <eff1>)))
    (rhs (prefer operator <op1> op2>)))

(PREFER-RELIABLE-OPS
    (priority 10)
    (lhs (and (current-node <node>)
            (candidate-op <node> <op1>)
            (candidate-op <node> <op2>)
            (is-more-reliable <op1> <op2>)))
    (rhs (prefer operator <op1> <op2>)))


(PREFER-UNRELIABLE-OPS
    (priority 10)
    (lhs (and (current-node <node>)
            (candidate-op <node> <op1>)
            (candidate-op <node> <op2>)
            (not-reliable <op1>)))
    (rhs (prefer operator <op1> <op2>)))

(PREFER-REVERSIBLE-OPS
    (priority 10)
    (lhs (and (current-node <node>)
            (candidate-op <node> <op1>)
            (candidate-op <node> <op2>)
            (is-reversible <op1>)
            (not-is-reversible <op2>)))
    (rhs (prefer operator <op1> <op2>)))
```

Binding Interactions

```
(PREFER-NO-OBJS-VERY-HIGH-PROTECTION
 (priority 10)
 (lhs (and (current-node <node>)
          (new-candidate-bindings <node> <binding-list-1>)
          (new-candidate-bindings <node> <binding-list-2>)
          (not-equal-lists <binding-list-1> <binding-list-2>)
          (has-objs-used-very-high-protection <binding-list-2>)
          (not-has-objs-used-very-high-protection <binding-list-1>)))
 (rhs (prefer bindings <binding-list-1> <binding-list-2>)))

(PREFER-LEAST-OBJS-VERY-HIGH-PROTECTION
 (priority 10)
 (lhs (and (current-node <node>)
          (new-candidate-bindings <node> <binding-list-1>)
          (new-candidate-bindings <node> <binding-list-2>)
          (not-equal-lists <binding-list-1> <binding-list-2>)
          (num-objs-used-very-high-protection <binding-list-1> <n1>)
          (num-objs-used-very-high-protection <binding-list-2> <n2>)
          (smaller <n1> <n2>)))
 (rhs (prefer bindings <binding-list-1> <binding-list-2>)))
```

C.2 Metapredicates

The meta-predicates defined for EXPO are the following:

- `(BELOW-EXP-DEPTH-LIMIT <node>)`

  Tests whether a node is below a user-defined depth.

- `(CURRENT-PS <ps>)`

  Used to set a context for the activation of the rule. Tests the current problem solving context. Two contexts are currently defined: *main* and *experimentation.*

- `(NODE-LEVEL <node> <level>)`

  Returns the depth of a node.

- `(IS-CURRENT-STATE <node> <state>)`

  Returns the current state at that node.

- `(CURRENT-PLAN <node> <plan>)`

  Returns the current plan at that node.

- `(IS-TOO-LONG-PLAN <plan>)`

  Tests whether the plan is longer than a user-defined length.

- `(HAS-FEWER-CHANGES <state1> <state2>)`

  Tests whether the number of differences with the initial state is smaller for ¡state1¿ than for ¡state2¿.

- `(DOES-TOP-GOAL-CONCORD <goal>)`

  `(NOT-DOES-TOP-GOAL-CONCORD <goal>)`

  Test whether the goal is the same as any pending goals in the main plan.

- `(DOES-TOP-PROTECTION-VIOLATION <goal>)`

  `(NOT-DOES-TOP-PROTECTION-VIOLATION <goal>)`

  Test whether the goal clobbers a goal previosly achieved for the main plan.

- (DOES-TOP-PREREQUISITE-VIOLATION <goal>)

  (NOT-DOES-TOP-PREREQUISITE-VIOLATION <goal>)

  Test whether the goal clobbers a predicate needed for later steps of the main plan.

- (HAS-OBJS-USED-VERY-HIGH-PROTECTION <obj>)

  (NOT-HAS-OBJS-USED-VERY-HIGH-PROTECTION <obj>)

  Test whether any of the objects is of a very high protection type.

- (NUM-OBJS-USED-VERY-HIGH-PROTECTION <objs> <n>)

  Returns how many objects are of a very high protection type.

- (IS-MORE-RELIABLE <op1> <op2>)

  (NOT-RELIABLE <op1> <op2>)

  Test whether one operator is more reliable than another. The reliability is computed as the ratio of the number of successful and the number of failed executions.

- (IS-REVERSIBLE <op>)

  (NOT-IS-REVERSIBLE <op>)

  Test whether the operator is reversible.

- (ARE-EFFECTS-OF <op> <effects-list>)

  Returns the effects list of the operator.

The meta level predicates used by EXPO that are provided by PRODIGY are the following:

- (CANDIDATE-NODE <node>)

  Should be used in selecting, rejecting, and preferring nodes. Tests whether a node is among the candidate set of nodes in the search tree.

- (CURRENT-NODE <node>)

  Tests whether <node> has been chosen as current node in this decision phase.

- **(CANDIDATE-OP <node> <op>)**

  Tests whether **<op>** is a member of the relevant operators being considered at the current **<node>**.

- **(CURRENT-OP <node> <op>)**

  Tests whether **<op>** is the current ¡op¿ for the current goal at the current node.

- **(CANDIDATE-BINDINGS <bindings> <node>)**

  Tests whether **<bindings>** is a member of the default set of candidate bindings for the current operator, goal, and node.

- **(KNOWN <node> <expression>)**

  Tests if an expression is true in the current state at the node.

- **(IS-EQUAL <x> <y>)**

  (NOT-EQUAL <x> <y>)

  These test for equality and inequality.

References

Amsterdam, Jonathan. 1988. Extending the Valiant learning model. In *Proceedings of the Fifth International Joint Conference on Machine Learning.* Ann Arbor, MI.

Angluin, Dana. 1987. Queries and concept learning. *Machine Learning* 2(4):319–342.

Boose, John H. 1992. Knowledge acquisition. In *Encyclopedia of Artificial Intelligence*, ed. Stuart C. Shapiro. New York, NY: John Wiley & Sons.

ed. Michael Brady. 1982. *Robot Motion: Planning and Control.* Cambridge, MA: MIT Press.

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

Bylander, Tom and Michael A. Weintraub. 1988. A corrective learning procedure using different explanatory types. In *AAAI Spring Symposium on Explanation-Based Learning.* Stanford, CA.

Carbonell, J. G., J. Blythe, O. Etzioni, Y. Gil, R. L. Joseph, D. Kahn, C. A. Knoblock, S. Minton, M. A. Pérez, S. Reilly, M. M. Veloso, , and X. Wang. 1992. PRODIGY4.0: *The Manual and Tutorial.* Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University.

Carbonell, Jaime G. and Yolanda Gil. 1990. Learning by experimentation: The operator refinement method. In *Machine Learning, An Artificial Intelligence Approach, Volume III.* San Mateo, CA: Morgan Kaufmann.

Carbonell, Jaime G., Yolanda Gil, R. L. Joseph, Craig A. Knoblock, Steve Minton, and Manuela M. Veloso. 1990. Designing an integrated architecture: The PRODIGY view. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society.* Boston, MA.

Carbonell, Jaime G., Craig A. Knoblock, and Steven Minton. 1991. PRODIGY: An integrated architecture for planning and learning. In *Architectures for Intelligence*, ed. Kurt VanLehn. Hillsdale, NJ: Lawrence Erlbaum Associates.

Chang, Tien C. and Richard A. Wysk. 1985. *An Introduction to Automated Process Planning Systems.* Englewood Cliffs, NJ: Prentice Hall.

Chapman, David. 1987. Planning for conjunctive goals. *Artificial Intelligence* 32(3):333–377.

Cheng, Peter C-H. 1990. *Modelling Scientific Discovery.* PhD thesis, The Open University, Milton Keynes, England.

Chien, Steve A. 1990. *An Explanation-Based Learning Approach to Incremental Planning.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL.

Christensen, Jens. 1991. *Automatic Abstraction in Planning.* PhD thesis, Department of Computer Science, Stanford University, Stanford, CA.

Christiansen, Alan D. 1992. *Automatic Acquisition of Task Theories for Robotic Manipulation.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

Danyluk, Andrea D. 1991. *Extraction and Use of Contextual Attributes of Theory Completion: An Integration of Explanation-Based and Similarity-Based Learning.* PhD thesis, Columbia University, New York, NY.

Davis, Randall. 1976. *Meta-level Knowledge for Construction and Maintenance of Large Knowledge Bases.* PhD thesis, Stanford University, Stanford, CA.

Davis, Randall. 1984. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence* 24(1-3):347–410.

Davis, Randall, Howard Shrobe, Walter Hamscher, Karen Wieckert, Mark Shirley, and Steve Polit. 1982. Diagnosis based on descriptions of structure and function. In *Proceedings of the National Conference on Artificial Intelligence.* Pittsburgh, PA.

DeJong, Gerald F. and Raymond Mooney. 1986. Explanation-based learning: An alternative view. *Machine Learning* 1(2):145–176.

Dent, Lisa and Jeffrey C. Schlimmer. 1990. Learning from indifferent agents. In *Proceedings of the Twelfth Annual Conference of the Coynitive Science Society.* Cambridge, MA.

Dietterich, Thomas G. 1986. Learning at the knowledge level. *Machine Learning* 1(3):287–316.

Ellman, Thomas. 1989. *Integrated Analytic and Empirical Learning of Approximations for Intractable Theories.* PhD thesis, Columbia University, New York, NY.

Etzioni, Oren. 1990. *A Structural Theory of Explanation-Based Learning.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Falkenheiner, Brian. 1989. *Learning from Physical Analogies: A Study in Analogy and the Explanation Process.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL.

Falkenheiner, Brian and Shankar A. Rajamoney. 1988. The interdependencies of theory formation, revision, and experimentation. In *Proceedings of the Fifth International Conference on Machine Learning.* Ann Arbor, MI.

Falkenheiner, Brian C. and Ryszard S. Michalski. 1986. Integrating quantitative and qualitative discovery: the ABACUS system. *Machine Learning* 1(4):367–401.

Fikes, R. E., P. E. Hart, and N. J. Nilsson. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3(4):251–288.

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

Flann, Nicholas S. 1990. Applying abstraction and simplification to learn in intractable domains. In *Proceedings of the Seventh International Conference on Machine Learning.* Austin, TX.

Genesereth, Michael R. 1984. The use of design descriptions in automated diagnosis. *Artificial Intelligence* 24(1-3):411–436.

Genest, Jean, Stan Matwin, and Boris Plante. 1990. Explanation-based learning with incomplete theories: A three-step approach. In *Proceedings of the Seventh International Conference on Machine Learning.* Austin, TX.

Gil, Yolanda. 1991. *A Specification of Process Planning for* PRODIGY. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University.

Gross, Klaus P. 1988. Incremental multiple concept learning using experiments. In *Proceedings of the Fifth International Conference on Machine Learning.* Ann Arbor, MI.

Gross, Klaus P. 1991. *Concept Acquistion through Attribute Evolution and Experiment Selection.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

Hall. Robert J. 1988. Learning by failure to explain: Using partial explanation to learn in incomplete or intractable domains. *Machine Learning* 3(1):45–78.

Hammond, Chris J. 1986. *Case-based Planning: An Integrated Theory of Planning, Learning, and Memory.* PhD thesis, Yale University, New Haven, CN.

Haussler, David. 1988. Quantifying inductive bias: Valiant's learning framework. *Artificial Intelligence* 36(2):177–221.

Haussler, David. 1989. Learning conjunctive concepts in structural domains. *Machine Learning* 34(1):7–40.

Hayes, Caroline. 1990. *Machining Planning: a Model of an Expert Level Planning Process.* PhD thesis, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

Huffman, Scott B., Douglas J. Pearson, and John E. Laird. 1992. Correcting imperfect domain theories: A knowledge-level analysis. In *Machine Learning: Induction, Analogy and Discovery.* Boston, MA: Kluman Academic Press.

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

Joseph, Robert L. 1992. *Graphical Knowledge Acquistion for Visual Planning Domain.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

Kaelbling, Leslie P. 1990. *Learning in Embedded Systems.* PhD thesis, Stanford University.

Kedar, Smadar T., John L. Bresina, and C. Lisa Dent. 1991. The blind leading the blind: Mutual refinement of approximate theories. In *Proceedings of the Eight Machine Learning Workshop.* Evanston, IL.

Knoblock, Craig A. 1991. *Automatically Generating Abstractions for Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

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

Kolodner, Janet L. 1980. *Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model.* PhD thesis, Yale University, New Haven, CN.

Korf, Richard E. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1):35–77.

Kuhn, Thomas S. 1977. *The Essential Tension: Selected Studies in Scientific Tradition and Change.* Chicago, IL: University of Chicago Press.

Kulkarni, Deepak S. 1988. *The Process of Scientific Research: The Strategy of Experimentation.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Kuokka, Dan R. 1990. *The Deliberate Integration of Planning, Execution and Learning.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

Laird, John E. and Paul S. Rosenbloom. 1990. Integrating execution, planning and learning in soar for external environments. In *Proceedings of the Eighth National Conference on Artificial Intelligence.* Boston, MA.

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

Laird, John E., Eric S. Yager, Christopher M. Tuck, and Michael Hucka. 1989. Learning in tele-autonomous systems using Soar. In *Proceedings of the NASA Conference on Space Telerobotics.*

Langley, Pat. 1987. A general theory of discrimination learning. In *Production System Models of Learning and Development.* Cambridge, MA: MIT Press.

Langley, Pat, Herbert A. Simon, Gary L. Bradshaw, and Jan M. Żytkow. 1987. *Scientific Discovery: Computational Explorations of the Creative Processes.* Cambridge, MA: MIT Press.

Langley, Pat, Ken Thompson, Wayne Iba, John H. Gennari, and John A. Allen. In press. An integrated cognitive architecture for autonomous agents. In *Representation and Learning in Autonomous Agents*, ed. Walter Van De Velde. Amsterdam, The Netherlands: North Holland.

Ling, Xiaofeng. 1991. Inductive learning from good examples. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence.* Sydney, Australia.

Maes, Pattie. 1991. Adaptive action selection. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society.* Chicago, IL.

Maes, Pattie and Rodney A. Brooks. 1990. Learning to coordinate behaviors. In *Proceedings of the Eight National Conference on Artificial Intelligence.* Boston, MA.

Mahadevan, Sridhar. 1989. Using determinations in EBL: A solution to the incomplete theory problem. In *Proceedings of the Sixth International Workshop on Machine Learning.* Ithaca, NY.

Mahadevan, Sridhar and Jonathan Connell. 1992. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence* 55(2-3):311–365.

ed. Sandra Marcus. 1990. *Knowledge Acquisition: Selected Research and Commentary.* Kluwer Academic Publishers.

ed. Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell. 1983. *Machine Learning, An Artificial Intelligence Approach.* Palo Alto, CA: Tioga Press.

Minton, Steve. 1988. *Learning Search Control Knowledge: An Explanation-based Approach.* Boston, Massachusetts: Kluwer Academic Publishers.

Minton, Steve, Jaime G. Carbonell, Craig A. Knoblock, Dan R. Kuokka, Oren Etzioni, and Yolanda Gil. 1989. Explanation-based learning: A problem solving perspective. *Artificial Intelligence* 40(1-3):63–118.

Minton, Steve, Craig A. Knoblock, Dan R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. 1989. PRODIGY *2.0: The Manual and Tutorial.* Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Mitchell, Tom, Paul Utgoff, and Ranan Banerji. 1983. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning, An Artificial Intelligence Approach. Volume I.* Palo Alto, CA: Tioga Press.

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

Mitchell, Tom M., Richard M. Keller, and Smadar T. Kedar-Cabelli. 1986. Explanation-based learning: A unifying view. *Machine Learning* 1(1):47–80.

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

Newell, Allen. 1982. The knowledge level. *Artificial Intelligence* 18(1):87–127.

Newell, Allen and Herbert A. Simon. 1972. *Human Problem Solving.* New Jersey: Prentice-Hall.

Nordhausen, Bernd and Pat Langley. 1992. An integrated framework for empirical discovery. *Machine Learning* To appear.

Ourston, Dick and Raymond J. Mooney. 1990. Changing the rules: A comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence.* Boston, MA.

Patil, Ramesh S., Peter Szolovits, and William B. Schwartz. 1981. Causal understanding of patient illness in medical diagnosis. In *Proceedings of the Seventh International Conference of Artificial Intelligence.* Vancouver, B.C., Canada.

Pazzani, Michael J. 1988. *Learning Causal Relationships by Integrating Empirical and Explanation-based Learning Methods.* PhD thesis, University of California at Los Angeles, Los Angeles, CA.

Pazzani, Michael J. 1990. Learning fault diagnosis heuristics from device descriptions. In *Machine Learning: An Artificial Intelligence Approach, Vol III.* San Mateo, CA: Morgan Kaufmann.

Pérez, M. Alicia and Oren Etzioni. 1992. DYNAMIC: A new role for training problems in EBL. In *Proceedings of the Ninth International Conference on Machine Learning.* Aberdeen, Scotland.

Pitt, Leonard and Leslie Valiant. 1988. Computational limitations on learning from examples. *Journal of the ACM* 35(4):965–984.

Rajamoney, Shankar A. 1988. *Explanation-Based Theory Revision: An Approach to the Problems of Incomplete and Incorrect Theories.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL.

Rajamoney, Shankar A. 1992. The design of discrimination experiments. *Machine Learning* To appear.

Rajamoney, Shankar A. and Gerald F. DeJong. 1987. The classification, detection, and handling of imperfect theory problems. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence.* Milano, Italy.

Rivest, Ron L. and Robert Sloan. 1988. Learning complicated concepts reliably and usefully. In *Proceedings of the Workshop on Computational Learning Theory.* Pittsburgh, PA.

Ruff, Ritchey A. and Thomas G. Dietterich. 1989. What good are experiments? In *Proceedings of the Sixth International Workshop on Machine Learning.* Ithaca, NY.

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

Sacerdoti, Earl D. 1977. *A structure for Plans and Behavior.* New York, NY: American Elsevier.

Salzberg, Steven, Arhtur Delcher, David Heath, and Simon Kasif. 1991. Learning with a helpful teacher. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence.* Sydney, Australia.

Sammut, Claude and Ranan Banerji. 1986. Learning concepts by asking questions. In *Machine Learning: An Artificial Intelligence Approach, Volume II.* Los Altos, CA: Morgan Kaufmann.

Schoppers, Marcel J. 1989. *Representation and Automatic Synthesis of Reaction Plans.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL.

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

Sleeman, Derek, Haym Hirsh, Ian Ellery, and In-Yung Kim. 1990. Extending domain theories: Two case studies in student modeling. *Machine Learning* 5(1):11–37.

Subramanian, Devika and Joan Feigenbaum. 1986. Factorization in experiment generation. In *Proceedings of the Fifth National Conference on Artificial Intelligence.* Philadelphia, PA.

Sussman, Gerald J. 1975. *A Computer Model of Skill Acquisition.* New York, NY: American Elsevier.

Sutton, Richard S. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning.* Austin, TX.

Tadepalli, Prasad. 1989. Lazy-explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence.* Detroit, MI.

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

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

Veloso, Manuela M. 1992. *Learning by Analogical Reasoning in General Problem Solving.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

Watkins, Christopher. 1989. *Learning from Delayed Rewards.* PhD thesis, Kings College.

Wilensky, Robert. 1981. PAM. In *Inside Computer Understanding*, ed. R.C. Shank and C.K. Riesbeck. Hillsdale, NJ: Erlbaum.

Wilensky, Robert. 1983. *Planning and Understanding: A Computational Approach to Human Reasoning.* Reading, MA: Addison Wesley.

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

Winston, Patrick H. 1975. Learning structural descriptions from examples. In *The Psychology of Computer Vision*, ed. P.H. Winston. New York, NY: McGraw Hill.

Żytkow, Jan M., Jieming Zhu, and Abul Hussam. 1990. Automated discovery in a chemistry laboratory. In *Proceedings of the Eighth National Conference on Artificial Intelligence.* Boston, MA.