Northwestern University

The Institute for the Learning Sciences

FLEXIBLE LEARNING IN A MULTI-COMPONENT PLANNING SYSTEM

Technical Report # 46 • November 1993

Bruce Tepper Krulwich

Established in 1989 with the support of Andersen Consulting

NORTHWESTERN UNIVERSITY

Flexible Learning in a Multi-Component Planning System

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

by

Bruce Tepper Krulwich

EVANSTON, ILLINOIS

December 1993

Abstract

Flexible Learning in a Multi-Component Planning System

Bruce Tepper Krulwich

People are able to learn a wide range of lessons from a given experience, depending on which of their cognitive abilities seem to need improvement. A theory of learning to plan should account for how and why an intelligent agent can learn such a diversity of lessons. Such a theory must address not only *how* and *when* an agent learns, but also *what* the agent should learn, because lessons must be formulated appropriately for the skills being improved. This thesis argues that a machine learning system must be able to dynamically determine what to learn from an experience. Making this determination requires that the system possess explicit knowledge of its own decision-making procedures, and be able be apply this knowledge in learning.

This thesis describes the CASTLE system, which learns new rules for a variety of cognitive tasks in the domain of competitive games, in particular chess. CASTLE's tasks include detection of threats and opportunities, plan recognition, goal generation, planning, counterplanning, and plan selection. CASTLE uses knowledge of its planning procedures to determine which of its decision-making components are responsible for expectation failures, and uses an abstract model of planning to appropriately formulate new rules.

ACKNOWLEDGEMENTS

With thanks to the Almighty, who makes all things possible:

I first of all want to thank Larry Birnbaum, my advisor, for all of his guidance in the past six years. Larry not only directed my research and thesis-writing, but more importantly he conveyed to me an approach to AI that forms the cornerstone of my thinking. Most of all, his enthusiasm about AI and his enjoyment in thinking about it has made it fun to work with him, and has hopefully been contagious.

I also owe a great deal to Gregg Collins, the second member of my committee and collaborator on the project. Gregg's calm and clear way of working through complicated issues has influenced me more than he may think, and he's been a pleasure to work with.

Thanks also to Ken Forbus, my third committee member, for his help in clarifying a number of issues that had to be addressed in the thesis. And of course none of this would exist were it not for Roger Schank's establishing the AI lab at Yale and The Institute for the Learning Sciences at Northwestern, and imbuing them with his sense of excitement and purpose.

Most of all, I want to thank my wife Devorah, for making life wonderful, for being so supportive, and for making it all worthwhile. And Daniel and Sarah, for showing me how little any of us know about intelligence and learning.

I also want to thank my parents, grandparents, and brothers, who have been fantastically supportive of everything I've been doing, despite my occasional inability to explain it. And I couldn't have gotten through all these years without my friends in Bethesda, Pittsburgh, New Haven, and Chicago, particularly Victor Assal, Steve and Caryn Gale, and Yossi Prager.

My early interest in computer science was largely due to Richard Smith, G. Edwards, and Gary Knott. At Carnegie Mellon I learned alot from Mark Perlin and Dave Touretzky, both of whom had quite a bit to do with my interest in pursuing research. My time with Drew McDermott at Yale taught me alot of what I know about rigorous approaches to both formal and "informal" AI problems.

It's impossible for me to mention all the people I've worked with over the years, the fellow students and researchers at CMU, Yale, and ILS, and the people I've talked to at conferences and by e-mail. Rather than try to list you all, and undoubtedly leave someone

out, I'll simply thank you all, and trust that you know who you are. I do want to specifically mention the other members of the research group at ILS, Louise Pryor, Matt Brand, and Mike Freed, and endless conversations that I had with Matt Brand, Bill Ferguson, Eric Jones, Arman Maghbouleh, and Louise Pryor that helped form and focus my thoughts about AI as a whole.

Lastly, I want to thank Alan Meyrowitz for support for this work from the Office of Naval Research under contract N00014-89-J-3217, and Abe Waksman for support from the Air Force Office of Scientific Research under contract AFOSR-91-0341-DEF. Additional support came from the Advanced Research Projects Agency, monitored by AFOSR under contract F49620-88-C-0058 and by ONR under contract N00014-91-J-4092. The Institute for the Learning Sciences was established in 1989 with the support of Andersen Consulting. The Institute receives additional support from Ameritech and North West Water, Institute Partners.

Contents

Chapter 1

Flexible learning

A theory of learning must ultimately address three issues:

- when to learn,
- what to learn, and
- how to learn.

The overwhelming majority of research in machine learning has been concerned exclusively with the last of these questions, how to learn. This work ranges from work in purely inductive category formation (e.g., [Winston, 1975; Mitchell, 1982; Michalski, 1983; Quinlan, 1986]) to more knowledge-based approaches (e.g., [DeJong and Mooney, 1986; Mitchell *et al.*, 1986; Schank *et al.*, 1986]). The aim of this work has generally been to develop and explore algorithms for generalizing or specializing category definitions. The nature of the categories being defined—i.e., what is being learned—is rarely a consideration in the development of these algorithms. For purely inductive approaches, this is entirely a matter of the empirical data that serves as input to the learner. In explanation-based approaches (EBL), it is a matter of the user-defined "goal concept"—in other words, input of another sort. In neither case is the question of what is being learned taken to be within the purview of the model under development.[1]

Some work—in particular that in which learning has been addressed within the context of performing a task—has addressed the first question above, namely, when to learn. A common approach to this issue, known as *failure-driven learning,* is based on the idea that a system should learn in response to performance failures. The direct connection this establishes between learning and task performance has made this approach among the most widespread in learning to plan [Sussman, 1975; Hayes-Roth, 1982; Schank, 1982; Kolodner, 1987; Minton, 1988a; Hammond, 1989].

For the most part, however, even these models do not address the second question above, what to learn. In many cases, this is because the models are only capable of learning one type of lesson. What to learn is thus predetermined. For example, many systems that learn

[1] Although the need to address this question in the actual deployment of learning algorithms has been explicitly recognized [Mitchell *et al.*, 1986, p. 72].

exclusively from planner success always learn the same thing, namely a generalized form of the plan that was created (e.g., [Mitchell, 1990]). Similarly, many systems that learn from plan outcomes always learn the same type of planning knowledge—e.g., when it is feasible to make certain simplifying assumptions or to defer planning—from each situation (e.g., [Chien, 1990; DeJong *et al.*, 1993]). Even systems that can learn more than one thing generally do so in a predetermined and highly inflexible fashion. The CHEF system [Hammond, 1989], for instance, which can learn a variety of types of concepts, has trigger conditions for each type of concept to learn built into its control structure. Similarly, the PRODIGY system [Minton, 1988a; Minton *et al.*, 1989; Minton, 1990] has an explicit set of hard-wired *example recognizer rules* which are used to determine what can be learned in any situation.

While this type of solution can often be effective for any individual application setting, it fails to provide an account of how a learning system could determine for *itself* what to learn, and do so in a manner that is flexible enough to take account of the internal and external context of learning. For a system that is capable of learning a wide variety of types of concepts, in a wide variety of settings, the number of hard-wired mapping rules required to do this would be very large, and the rules themselves would get very difficult to manage or reason about, and may even be impossible to formulate.[2]

More importantly, however, is the fact hard-wired rules of this type do not provide a *theory* of determining what to learn. Just as a set of rules for actions can result in intelligent behavior without providing a foundation for the actions [Brooks, 1986; Agre and Chapman, 1987], hard-wired rules for determining what to learn can be effective but nonetheless do not necessarily provide a theory underlying these decisions. The point is that just as complex decisions about actions are very difficult to formulate using hard-wired rules, and thus require inference, so too complex decisions about what to learn require inference.

In sum, however effective in particular application settings, pre-determined approaches to what to learn are inherently incapable of accounting for the diversity and flexibility that a learning system must be able to demonstrate.

1.1 Flexible learning: An everyday example

Consider the case of a person cooking rice pilaf for the first time. The last step in the directions says to "cover the pot and cook for 25-30 minutes." Suppose the person starts the rice cooking and then goes off to do something else—say, clean up the house. In the interim, the pot boils over. When the person returns to the kitchen a half-hour later, the rice pilaf is ruined.

What should be learned from this sequence of events?[3] Intuitively we can imagine a number of lessons that might be learned:

[2]For example, in addition to using ***example-recognizers*** to identify what to learn, the PRODIGY system also requires procedurally encoded ***selection heuristics*** to filter out "uninteresting" learning opportunities [Minton, 1988a, sec. 4.2]. We will return to this point in section 8.2.

[3]Clearly an agent may decide that the incident was a fluke, and that it's not worth learning anything. We assume here that the agent wants to learn from the experience.

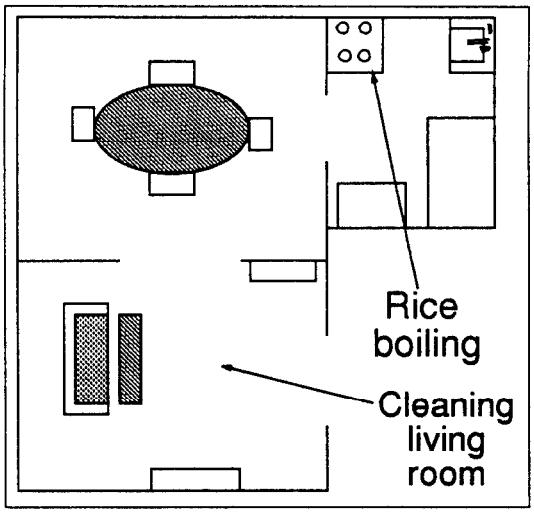


Figure 1.1: Cooking rice pilaf while cleaning

1. Whenever a covered pot containing liquid is on the stove, keep an ear peeled for the sound of the lid bouncing or the sound of the water bubbling.

2. Do not put a covered pot with liquid in it over a high flame, because it will boil over. The flame should be turned down.

3. When cooking over a high flame, leave the pot uncovered or the lid ajar.

4. Don't do loud things while cooking on the stove.

5. When cooking liquid in a covered pot, stay in the kitchen, because it's hard to hear a pot boiling over from the other rooms.

6. Don't cook over high flame when busy.

While all of these lessons are sensible, they are very disparate, in that they address very different issues. The lessons concern different aspects of behavior, refer to different portions of the agent's plan, and are expressed in different vocabularies. It is difficult to imagine how any learning process that did not distinguish among these alternatives in some way would be capable of such diverse behavior. Rather, it seems more likely that before the agent can undertake the task of learning from the mistake, he must select a lesson (or set of lessons) to learn.[4] In other words, given that the agent has decided to learn from the mistake, and given that he is capable of carrying out the learning task, he still has to first determine what to learn.

[4]Masculine pronouns are used throughout the discussion of this example because the incident occurred to a man.

Which of these lessons the agent should learn depends on his perceptual and plan execution abilities, the plans that he typically generates, and the constraints under which he operates. If the agent would in general be able to hear the pot boiling over from the other room, but simply had not attended to the soft sounds that he heard in this instance, then tuning his perceptual attention apparatus when a pot is on the stove is a good way to adapt to the newly discovered problem of pots boiling over. This is the first lesson listed above. If the agent would not be able to hear the pot boiling over however hard he listened, another lesson must be learned, either to prevent pots from boiling over by changing the parameters of the cooking process (e.g., by turning down the flame), or to avoid leaving the kitchen when a pot is on the stove. If the recipe will work properly with the flame turned very low, as is the case with rice, then the second lesson will suffice for the agent to plan properly in the future. If the recipe cannot be cooked over a low flame, but can be cooked uncovered (which is not usually the case with rice but is with other foods such as spaghetti), the third lesson is the one that should be learned. Alternatively, the other lessons could be learned to allow the agent to avoid boiling over in other ways.

We see, then, that the agent could learn several things in response to the rice pilaf boiling over. Which of the lessons the agent should learn, whether changes to the cooking methods, the idea of staying in the kitchen, or of tuning his perceptual apparatus, depend on the agent's perceptual and planning abilities, and on his knowledge of the domain. The key point is that many different lessons are possible. Any approach to determining what to learn must be flexible enough to account for this diversity.

1.2 Using cognitive tasks to determine what to learn

What would an appropriate theory of determining what to learn look like? Imagine the thought processes going on in the agent's head (consciously or subconsciously) in viewing his situation and considering what lesson to learn:

1. *Why was the rice ruined?*

   The rice boiled over.

2. *Could I have done something differently at the time I started the rice cooking to prevent the problem?*

   Yes, I could have lowered the flame or uncovered the pot.

3. *Without doing this, could I have prevented it from boiling over?*

   Yes, if I had heard it.

4. *Could I have heard it boiling over?*

   Maybe I could have, if I'd paid more attention.

5. *Why couldn't I hear it boiling over?*

   I was using the vacuum in the living room.

6. *Could I have planned things differently to enable me to hear?*

   Yes, I could have delayed vacuuming or stopped every few minutes to check the rice.

7. *Why didn't I?*

   I didn't think about the inability to hear from the other room.

The focus of this "dialogue" is on the decisions and actions of the agent that led to the rice burning, and particularly on what the agent could have considered or done to prevent the problem from arising. A self-dialogue of this sort is a means of analyzing the situation to explain what happened, and thereby focus learning from the experience [Chi *et al.*, 1989; Ram, 1989]. In this case the agent is performing *self-diagnosis,* trying to determine what mistake(s) he made that led to the rice burning. Put another way, the agent is considering possible lessons, and trying to determine which of them to learn. It is important to notice at this point that the dialogue is *not* specifically aimed at diagnosing the actions that the agent took to determine which action is to blame (although that may be involved as well). Rather, the dialogue is diagnosing the *decisions* that the agent made.

The key insight that will allow us to model this learning process is that each of the lessons that our agent can learn, and each of the questions in the hypothetical dialogue above, relates to a particular cognitive task that the agent was carrying out in the example:

1. Listen harder for bubbling or the lid bouncing: *Perceptual tuning*
2. Adjust the flame more carefully: *Plan step elaboration*
3. Leave the lid ajar: *Plan step elaboration*
4. Don't do loud things while cooking: *Scheduling/interleaving*
5. Stay in the kitchen while cooking: *Scheduling/interleaving*
6. Don't cook over high flame when busy: *Scheduling/interleaving*

By "cognitive tasks" we mean here the classes of decisions that the agent makes in the course of decision-making.[5] In our example, these cognitive tasks include such things as *plan step elaboration* (e.g., such as deciding how high to adjust the flame on the stove) and *perceptual tuning* (e.g., deciding what to listen for, in this case the sounds of the rice bubbling over). These tasks are themselves general cognitive abilities that are used frequently in goal-based behavior.

Given this insight, we can reformulate the learning problem posed above. Determining lessons to learn from a problem means first determining which cognitive tasks are relevant, and then determining what can be learned from the experience about how those particular tasks can be better carried out. In other words, we have transformed the problem of determining what to learn to two subproblems: determining what task to repair, and determining what aspect of the situation relates to that task, and how.

[5]Our notion of cognitive tasks is similar to the notion of *generic tasks* in [Chandrasekaran, 1987].

1.3 Modeling planner structure

In this thesis we will present an analytical approach to determining what to learn based on the notion that an agent should determine a lesson to learn from a situation by first selecting those cognitive tasks that pertain to performance in that situation, and then determining an appropriate lesson for each of them.

How can a computer system reason about its own decision-making and the cognitive tasks involved in that decision-making? The approach we will take is to design the system to facilitate this reasoning, by structuring its architecture in terms of *components* that carry out specific cognitive tasks. In other words, we partition the system into chunks, each of which is responsible for a particular cognitive task, and treat each chunk as a component of the architecture. The behavior of the system, and the opportunities to improve it, can then be analyzed in terms of the behavior of components and the interactions between them.[6]

To model this process in the rice pilaf example, we would design the agent model with components for the tasks of plan generation, plan step elaboration, plan step interleaving, perceptual tuning, and so forth. These components interact to produce the agent's decision-making behavior. When the rice is burned, the agent examines the behavior of each of its components leading up to the problem. Several of the components behaved incorrectly. The plan step interleaving component, for example, should have signaled the possible problem of not being able to hear the rice cooking while vacuuming the other room. The perceptual tuning component, on the other hand, should have focused more attention on the sounds coming from the kitchen. By examining how the components behaved, and reasoning about how they should have behaved, the agent can determine lessons to learn from the experience. Each lesson corresponds to a component that could profitably be repaired.

The next step, which answers the question of "what to learn," is to associate with each component information about its ideal (desired) behavior. The system can use this information to determine what can be learned from the situation about better carrying out the component's task. The perceptual tuning component, for example, will have an associated description of the task of adjusting the agent's perceptual apparatus in response to its goals and environment. In the rice pilaf example the system could realize, based on this information, that it can learn a lesson about attending to the rice while cleaning the living room. If this kind of information is provided for each component, the system can determine what to learn by retrieving the relevant information for each component potentially in need of repair. This approach to learning, and the implications of its implementation, forms the bulk of this thesis.

It is worth noting that the division of a planner architecture into components has been motivated for other reasons as well. One such purpose is to structure the use of specialized procedures in making complex decisions [Hayes-Roth and Hayes-Roth, 1979]. Structuring a planner into components provides a framework for incorporating diverse procedures for specific tasks which could not be easily incorporated into other, more uniform, planning

[6]This is the same approach that is often taken to reasoning about physical devices such as circuits and machines. In the "component model," devices are viewed as composed of separate components that are connected together [Davis, 1990, sec. 7.1]. The behavior of the device can then be analyzed by studying the behavior of the components and their interactions, using a model of the ideal behavior of each component.

paradigms. A closely related notion is that decomposing problem-solving tasks enables them to make use of specialized knowledge structures [Chandrasekaran, 1983; Chandrasekaran, 1987]. Another purpose of a component-structured planner is to facilitate reactivity and execution-time reasoning [McDermott, 1978; Collins *et al.*, 1991b; Simmons, 1991]. A component-based architecture shifts the view of plan execution from a general-purpose plan executor to a highly articulated set of components, each devoted to controlling a particular aspect of behavior.

1.4 Reasoning about justifications

Given this component-based approach to modeling the planning process, how can a computer system diagnose which component is responsible for a failure? The process of self-diagnosis, as discussed above, is aimed not at diagnosing the agent's actions (at least not directly), but rather at diagnosing the decision-making constructs that gave rise to these actions. In other words, the over-arching question is not "What action of mine led to the problem?" but is rather "What deficiency in the way I make decisions led to the problem?"

In contrast, previous research in learning to plan or solve problems in response to failures has generally been aimed at the first question, and has thus employed knowledge of the causal relations between the steps in the plan and the desired outcomes of those steps in diagnosis [Simmons, 1988a; Chien, 1990]. When a plan failure occurs, this information is used to see what step in the plan resulted in the failure. One obvious attraction of this approach is that this causal knowledge is useful for many other purposes (such as constructing the plans in the first place), and so it is reasonable to believe that an agent (or computer program) would already have this information available.

To diagnose failures in terms of faulty planner components, however, an agent requires analogous information concerning the causal relations between the decision-making processes used in planning, the actions taken, and the expected results. In other words, diagnosing the failure of a plan (or, more generally, of an expectation about the plan's performance) in terms of decision-making constructs requires information about the causal relations between the two. The agent needs to reason about the decision-making mechanisms that led to the belief that the plan would succeed, and the beliefs and assumptions upon which the decision-making was based.

More concretely, the agent must reason explicitly about his *justification* for his actions in terms of his own reasoning mechanisms. We consider our agent to know, or to be able to reconstruct, the reasons that he thought his decision-making was sound, and how his decision-making mechanisms led to the failure.[7] Introspective dialogues such as the one we saw above correspond to the agent's examination of this justification, and his consideration of where the faults lie.

In the rice pilaf example, the agent believed that his actions would lead to the rice being cooked and the other room being cleaned. The reason for believing this was roughly that:

[7]Justification structures have previously been used for a variety of purposes, including learning from experience by reapplication of decision-making sequences [Carbonell, 1986], diagnosis of physical devices [Davis, 1984], and truth maintenance [deKleer *et al.*, 1977; Doyle, 1979].

1. He had two good subplans, one for cooking the rice and one for cleaning the other room, and

2. He combined the two subplans properly, and

3. He could handle any problem that arose in executing the combined plan.[8]

Each of these is in turn believed for other reasons. The agent believes that he combined the two subplans well because *(1) The master plan included steps of both subplans;* and *(2) He handled all complications arising in combining the steps.* Similarly, the belief that he would be able to handle any problems that will arise is justified by the beliefs that *(1) He could detect all problems that arise;* and *(2) He could handle all problems that are detected.* These last two beliefs are assumptions about the abilities of two components in the agent's architecture. The other beliefs are similarly justified, in the end, by assumptions about the capabilities of the agent's components. The agent can determine which of his cognitive tasks are to blame for the problem by examining this justification for his actions. By modeling the process of self-diagnosis in terms of justifications, we have identified a specific form of knowledge that can enable the agent to determine the tasks that were performed badly.

Focusing the diagnosis process on the decisions made during planning has several advantages over focusing on the actions themselves. The most significant is that the agent can learn much more broadly applicable lessons if he thinks in terms of his decision-making rather than in terms of his particular plan. The lesson of tuning a perceptual apparatus to listen for problems that may potentially arise in a given situation, for example, may be applicable in situations far beyond those of cooking and cleaning. Another advantage of focusing on decisions rather than actions is that the lessons can relate to decisions that are not about particular actions, such as scheduling two subplans. Lastly, the lessons learned from examining decisions can be about decisions made at execution time in addition to those made at planning time.

Figure 1.2 shows a possible representation of the agent's justification of his decision-making in the rice pilaf example, in the format used later in the thesis. The diagram represents this justification structure as a circuit. Each "AND gate" represents a decision, in which a number of beliefs are considered (the beliefs that feed into the gate), and one or more beliefs are inferred. The beliefs that are not inferred by an AND gate are baseline assumptions that the system makes, and of these the ones in bold type refer to component capabilities. The numbers alongside some of the beliefs correspond to the self-questions in the dialogue discussed earlier. These justification structures are discussed in depth in chapter 4. We will return to the rice pilaf example again at the conclusion of the dissertation.

[8]The claim is not that intelligent agents truly "believe" such strong statements to be true, but rather that they must assume them to be "effectively true" in order to plan and reason about actions, and that these assumptions must be explicit in order to learn from their failure.

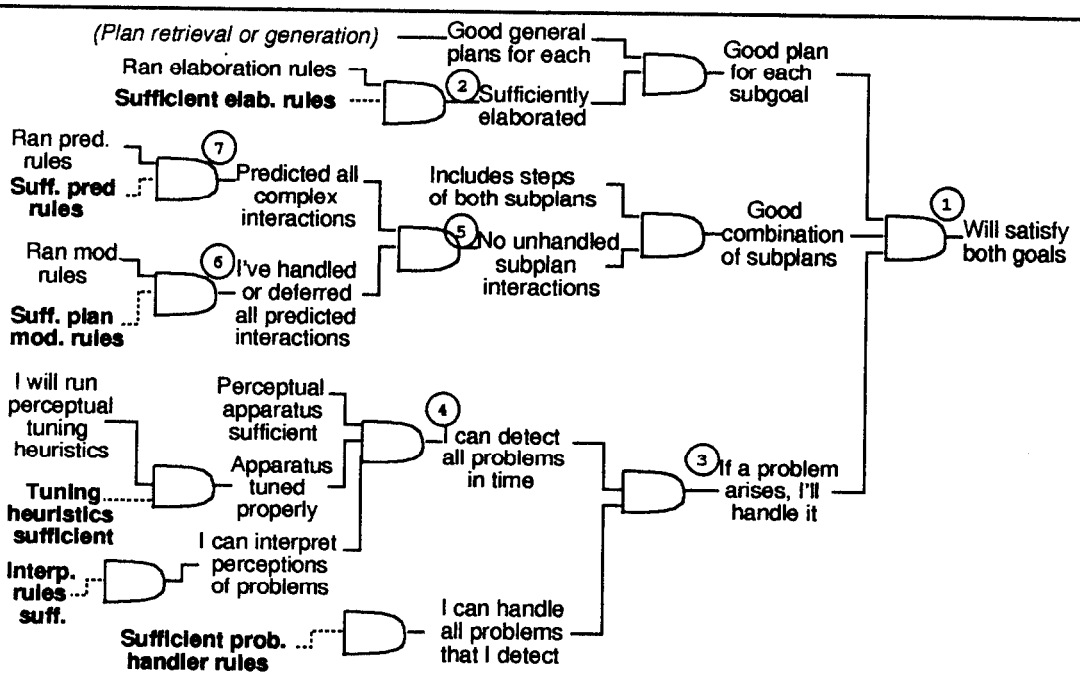


Figure 1.2: Justification structure for the rice pilaf example

1.5 Flexible learning: Putting it all together

We have seen that an intelligent agent must be able to dynamically determine what to learn, and that this process can require a significant amount of inference. By viewing the planning process as being composed of a variety of cognitive tasks, we transformed the problem of determining what to learn into two sub-problems: determining what cognitive task is at fault, and determining what could be learned to improve that task.

To implement this insight in a computer program, we proposed modeling cognitive tasks in terms of distinct components of a system's architecture, each of which performs a designated task or sub-task. Plan failures can then be diagnosed by examining justification structures, to find a component that did not perform its task correctly.

What we still need to specify is how this process is initiated in the first place. Our approach is for the system to maintain and monitor explicit *expectations* that reflect the assumptions made during planning [Schank, 1982; Doyle *et al.*, 1986; Ortony and Partridge, 1987]. These expectations carry with them justification structures that relate them to the decision-making processes and otherwise-implicit assumptions that underly their being expected in the first place. The failure of an expectation thus can directly lead to diagnosis of the relevant portions of the system's planning architecture.

This learning process is shown pictorially in figure 1.3. The planner considers the current situation and the active goals, and outputs a plan, along with a set of expectations to monitor. The failure of one of these expectations leads to diagnosis, which uses associated justification structures that represent part of the system's explicit self-model. The diagnosis

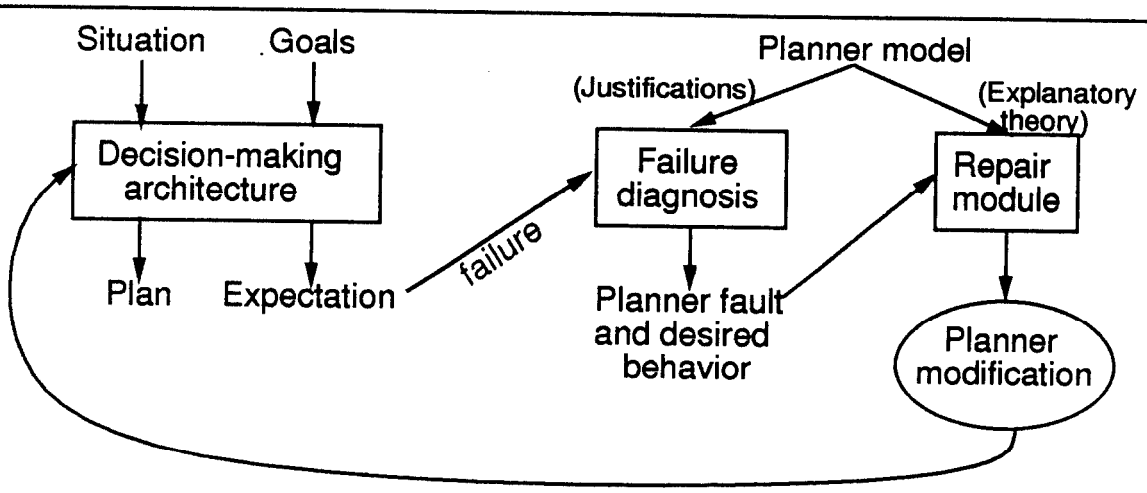


Figure 1.3: The learning process in CASTLE

process concludes which component is at fault, and how it should in fact have behaved. This information is passed to the repair module, which uses a model of the component's cognitive task to construct a repair.

1.6 The CASTLE system

The theory of learning and planning discussed in this thesis has been implemented in the CASTLE system, which operates in the domain of chess.[9] CASTLE's decision-making components include threat and opportunity detection, plan recognition, planning, counterplanning, and plan selection. It posts and monitors explicit expectations, and diagnoses and learns from expectation failures using the approach introduced earlier.

A snapshot of CASTLE's screen is shown in figure 1.4. The screen is composed of two main areas: the game board and the four program output windows.

The game board shows the human player (playing white) on the bottom, and allows pieces to be moved interactively.[10] There are four program output windows that are visible while the game is being played:

1. The *game record,* which lists the moves that have been made.
2. The *top-level events* window, which indicates the stage of decision-making in which CASTLE is currently engaged (e.g., planning, waiting for user's move, updating board, learning).
3. The *planning* window, which provides a transcript of the decisions involved in plan generation.

[9] CASTLE stands for *C*oncocting *A*bstract *S*trategies *T*hrough *L*earning from *E*xpectation-failures.

[10] CASTLE uses the XCHESS program as a graphical front-end. XCHESS is © 1986-1990 by the Free Software Foundation, Inc.

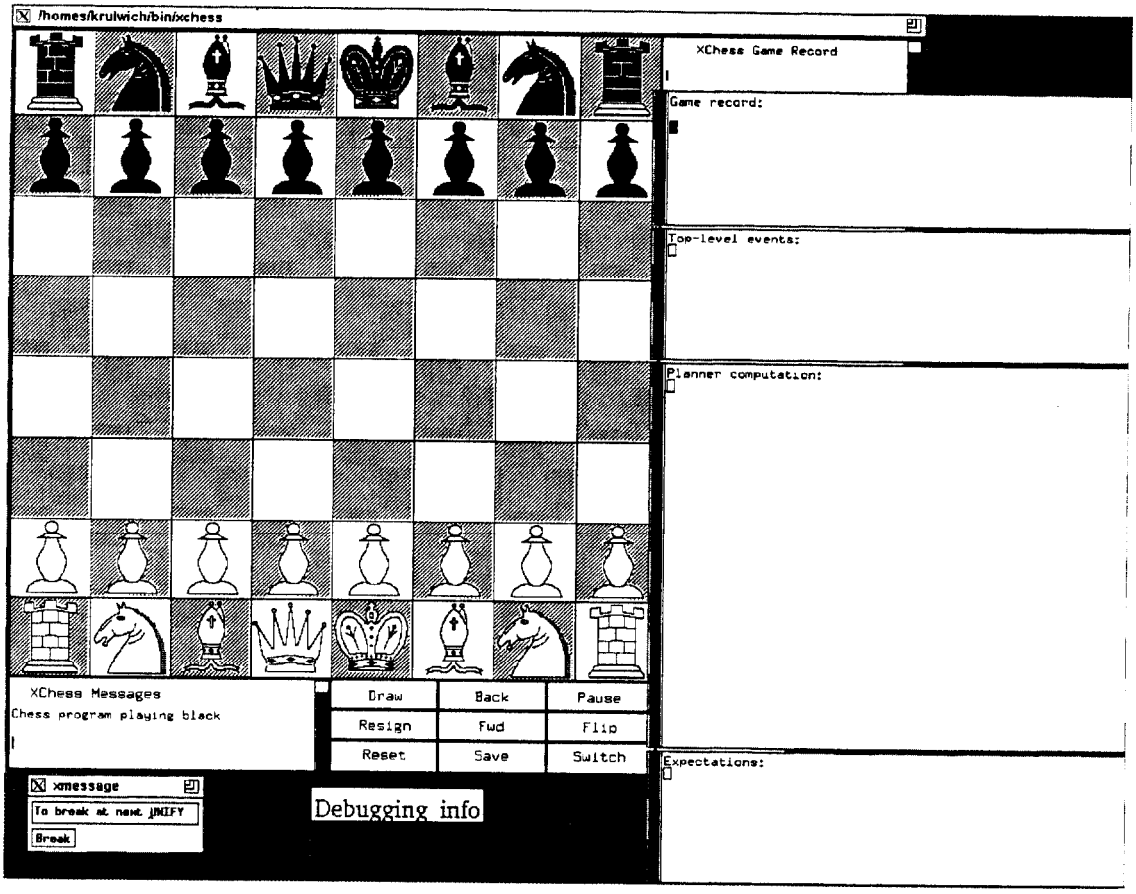


Figure 1.4: CASTLE's screen at the start of a game

4. The *expectations* window, which lists expectations as they are posted and as their status changes.

Additional windows pop up as appropriate during the execution of the program:

5. *Option selection* windows, which pop up in the course of program execution to give the user a choice of how to proceed. One such window pops up when an expectation failure occurs, describing the error and asking the user whether the system should attempt to learn from the failure. Others are used to select a particular example to run, to choose to restart the program after learning instead of continuing the game, and so on.

6. The *diagnosis* window, which provides a transcript of the diagnosis process whenever the system attempts to learn from an expectation failure.

7. The *rule construction* window, which similarly provides a transcript of the learning process.

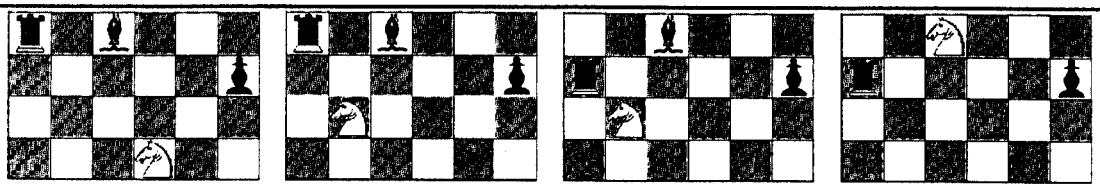


Figure 1.5: *Fork* example: Computer (black) to move

The reader should keep in mind that while CASTLE operates in the domain of chess, this is not a dissertation on chess *per se*. The focus throughout the thesis is on learning and planning, not on achieving quality chess play. No attempt will be made to relate CASTLE's decision-making architecture to those of master-level chess-playing systems, or to evaluate CASTLE's behavior in terms of becoming a chess expert. Rather, CASTLE is an attempt to model and implement flexible learning, and operates in the domain of chess for a combination of methodological and historical reasons (see, e.g., [Levinson *et al.*, 1991]).

CASTLE in action

As a quick example of CASTLE in action, suppose the system is operating without knowing the *fork* strategy, i.e., simultaneously attacking two opponent pieces from a single location. In the situation shown in figure 1.5, CASTLE is unaware in board (a) that the opponent can use the fork to capture its bishop or rook. When the opponent then applies the fork, resulting in board (b), the system has to choose which of its pieces to save. It decides in board (c) to save the rook at the expense of the bishop, and the opponent proceeds to capture the bishop in board (d). CASTLE must then find the fault in its decision-making mechanism that allowed it to get into such a situation, and learn how to avoid similar occurrences in the future.

As a result of this learning process, the system constructs a new rule, which is shown in figure 1.6. With this rule added, CASTLE will detect and respond to potential forks from the opponent. Figure 1.7 shows CASTLE (playing black) using the rule to realize that it should pass up capturing an opponent piece in order to avoid being forked. Additionally, the system is also able to use the fork offensively as well.

1.7 A guide to reading the thesis

This thesis consists of four sections. The first, comprising chapters 1 and 2, motivates and lays the groundwork for the research presented. The second, consisting of chapters 3, 4, and 5, presents and discusses the technical issues that arise and how they are solved and implemented in the CASTLE system. The third, chapters 6 and 7, presents CASTLE's approach to modeling and reasoning about component function by describing the set of examples that the system can handle. In doing this, the section describes CASTLE's accomplishments in

```
(BRULE learned-strategy-method8574
(strategy learned-strategy-method8574 ?player (world-at-time ?time)
          (goal-capture ?target-piece (loc ?r2 ?c2) ?info)
    (plan (move ?player move ?en-piece (loc ?r ?c) (loc ?r3 ?c3))
          (next (move ?player (capture ?target-piece) ?en-piece
                      (loc ?r3 ?c3) (loc ?r2 ?c2)) done)))
<=
 (and (at-loc ?player ?en-piece (loc ?r ?c) ?time)
     (move-legal-in-world
        (move ?player move ?en-piece (loc ?r ?c) (loc ?r3 ?c3))
        (world-at-time ?time))
     (not (at-loc ?other-player ?other-piece (loc ?r3 ?c3) ?time))
     (at-loc ?opp ?target-piece (loc ?r2 ?c2) ?time)
     (move-legal-in-world (move ?player (capture ?target-piece) ?en-piece
                                (loc ?r3 ?c3) (loc ?r2 ?c2))
        (world-at-time ?time))
     (at-loc ?opp ?other-target (loc ?r4 ?c4) ?time)
     (move-legal-in-world (move ?player (capture ?other-target) ?en-piece
                                (loc ?r3 ?c3) (loc ?r4 ?c4))
        (world-at-time ?time))
     (no (and (counterplan ?cp-meth ?opp
                (goal-capture ?target-piece (loc ?r2 ?c2)
                   (move (move ?player (capture ?target-piece) ?en-piece
                                (loc ?r3 ?c3) (loc ?r2 ?c2))))
                ?time ?cp)
              (counterplan ?cp-meth2 ?opp
                (goal-capture ?other-target (loc ?r4 ?c4)
                   (move (move ?player (capture ?other-target) ?en-piece
                                (loc ?r3 ?c3) (loc ?r4 ?c4))))
                ?time ?cp))))
```

A forced-outcome strategy for capturing a piece using two moves

Find a piece that can move

to an empty square and a piece of the opponent's that can be attacked by the moved piece and another piece of the opponent's that can be taken by the same piece

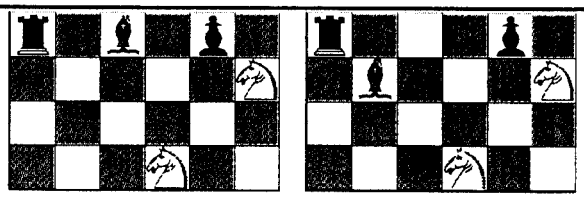
such that there is no counterplan for the opponent that handles both of the attacks

Figure 1.6: Learned strategy rule for the fork

implementing examples of flexible learning behavior, and discusses how the implementation could be extended within the framework presented to handle more examples. Finally, chapters 8 and 9 discuss the research against the background of previous research, and evaluate the degree to which CASTLE has accomplished its goals.

The thesis chapters are organized as follows:

- Chapter **2:** A detailed example
- Chapter **3:** CASTLE's decision-making architecture



(a) (b)

Figure 1.7: After learning the fork: Computer (black) to move

- Chapter 4: Diagnosing expectation failures
- Chapter 5: Repairing planner faults
- Chapter 6: CASTLE in operation
- Chapter 7: Enhancements to CASTLE's decision-making models
- Chapter 8: A look at related research
- Chapter 9: Evaluation and conclusions

The thesis presents a general approach to learning to plan. Within this framework, there are a number of other themes that are addressed, some of which may be of particular interest to the reader. One theme is how to construct an explicit model of a decision-making process. CASTLE's approach to learning employs introspection both for failure diagnosis and for rule construction. Thus, it is necessary that the planner and its assumptions be modeled in such a way as to support self-analysis.

A closely related theme in the thesis is diagnosis and model-based reasoning. Model-based reasoning is concerned with how an explicit model of a device can be used to diagnose faults in the device or to reason in other ways about the device's behavior [Stallman and Sussman, 1977; Davis, 1984]. Most of the previous research in this area has concerned devices such as combinatorial circuits, in which the desired behavior of any component of the device is a simple combination of its inputs, and in which a component's behavior is dependent only on its inputs and not on previous computations.[11] A number of issues arise in applying these techniques to diagnosing planner behavior.

Those readers who want to read about CASTLE's approach to particular issues can read one of the following portions of the thesis:

Contributions and accomplishments	Chapters 1, 2, 8, 9
The general approach (technical)	Chapters 3, 4, 5
Modeling a planning architecture	Chapters 3, 6, 7
Diagnosing a planner	Chapters 2, 4
Learning planning knowledge	Chapters 2, 5
CASTLE's implementation	Chapters 2, 6
CASTLE's behavior	Chapters 2, 6, 7

The two appendices contain the inference rules that make up CASTLE's decision-making model and explanatory theory.

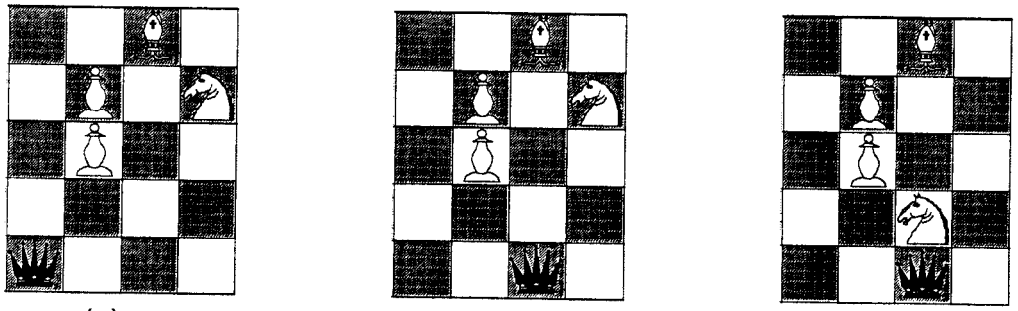
[11] A notable exception is [Hamscher and Davis, 1984], which discussed diagnosing circuits with state.

Chapter 2

A detailed example

In order to motivate and ground the issues discussed in this thesis, let's start with a detailed description of CASTLE in action, as it learns about blocking threats by *interposition.* In the chess situation depicted in figure 2.1(a), the computer has two options—it can use its queen to capture the opponent's knight, or it can construct a plan to take another opponent piece on a later turn. The only piece of the opponent's which is more valuable than the knight is the bishop, so this is the best piece to consider setting up for subsequent capture. One plan to capture the bishop is to move the queen two squares to the right and capture the bishop on the next turn. In deciding whether this is a viable plan, CASTLE must consider whether the opponent would have a response to this move that would disable the attack before it could be carried out.

It is clear in figure 2.1(b) that the opponent would in fact have a possible counterplan, namely moving its knight to block the attack. Suppose, however, that CASTLE didn't know that threats could be blocked by *interposing* a piece in this way. This might be the case, for example, if the system's previous game-playing experience involved other methods of counterplanning, such as running away or counterattacking, but not piece interposition. Under these circumstances, when considering the viability of its plan, the system would conclude that the opponent will not have a response to the threat. Given this assumption,



(a) (b) (c)

Figure 2.1: *Interposition* example: Computer (black) to move

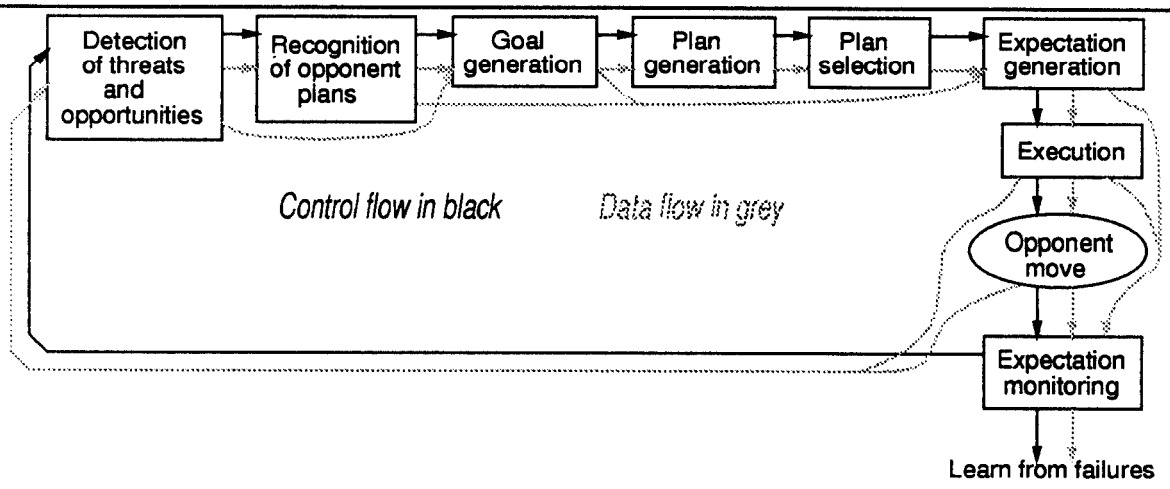


Figure 2.2: CASTLE's decision-making process

the system would consider the plan to take the bishop foolproof, and would proceed to carry it out.

Of course, this plan will fail, because the opponent will block the attack by interposing its knight, as can be seen in figure 2.1(c). CASTLE should learn from the failure by augmenting its counterplanner with a method for interposing a piece.

This chapter will discuss how this decision-making and subsequent learning take place in CASTLE. The steps in CASTLE's decision-making process are shown in figure 2.2, with arrows between the tasks showing the flow of control and data during decision-making. Each of these decision-making tasks corresponds to a *component* in CASTLE's architecture, whose job is to carry out the specified task. Each component has a set of rules for carrying out its task, and also has an explicit description of its purpose, the assumptions it makes, and its relationships with other components. The components and their implementations will be discussed in detail in chapter 3.

2.1 Interpreting the environment

The first task that CASTLE undertakes in making any decision is to examine its situation and determine the currently active threats and opportunities. Since any move can be a potential threat or opportunity in the context of a long-term plan, we begin by defining threats to be one-move captures. Given this definition, threats and opportunities are symmetric for the two players—a threat against the computer is an opportunity for the opponent, and vice-versa. CASTLE uses a single mechanism to detect threats against both players, and we will interchangeably refer to opportunities for one player and threats against the other.

Rather than reparse the board on each turn, the system maintains a set of currently active threats and opportunities, which it updates whenever the board changes.[1] In our

[1]This incremental threat detection mechanism is discussed in section 3.4, as well as in [Collins *et al.*,

example, the initial situation contains only one opportunity for a direct capture for the computer: moving the queen diagonally to take the opponent's knight. This opportunity may have been active previously, and carried over to the current set of active opportunities, or it may have been added this turn as a result of the most recent move (e.g., if the knight were just moved to its current location). The situation contains no direct threats against the computer.

Having detected the active threats and opportunities, CASTLE attempts to recognize plans that the opponent may be using or preparing. This task is carried out by CASTLE's *plan recognition* component, which uses several of the system's planning methods to see what plans CASTLE would consider if it were in the opponent's position, as well as by checking how the opponent's recent moves might be understood as part of a larger pattern of goal-related behavior. In our example CASTLE cannot determine any possible opponent plans, so it proceeds to the next step in the decision-making process. (We will see later how the plan recognition process is used in more complex examples.)

The next step is for CASTLE to formulate concrete goals for its move. This task is accomplished by the *goal generation* component, whose task is to recognize goals to consider in a given situation. In this case, the component recognizes three goals. First, the simple goal to take the opponent's knight is recognized from the opportunity for a direct capture. It should be clear that this particular "goal recognition" does not involve performing any new computation; rather it simply entails reformulating information that the system already knows about the availability of the capture into new terminology, that of active goals. The rule that accomplishes this says simply:[2]

TO COMPUTE: A goal to activate

DETERMINE: A possible capture of a piece
that was in the set of detected opportunities

The second and third goals are generated because there are two pieces that have very limited mobility and are thus recognized as likely targets for attack: the bishop and the top pawn. Having categorized them as likely targets, CASTLE will consider expending planning resources to try to capture them. Of course, a piece having limited mobility is also a sign that it may be well defended, so CASTLE has to be able to eliminate unattainable goals of this type without too much computation. At this point in processing CASTLE risks posting spurious goals, and will eliminate them later in the planning stage.

CASTLE thus has three active goals in the initial situation: capture the knight (through a direct attack), capture the bishop (using some as-of-yet-undevised plan), and capture the top pawn. There are, of course, other types of goals that CASTLE may recognize. These could include defending pieces from attack and putting pieces in safer positions, as well as such abstract notions as control of a region of the board and defensive structural arrangements of pieces. Additionally, there can be several reasons that any goal may be posted. For example, a piece may be a good target for capture if the opponent has a reason for not *wanting* to

1991a; Collins *et al.*, 1991b].

[2]These rules are presented more explicitly in chapter 3, and all of CASTLE's planning rules are given in appendix A. In this extended example the rules are only given in text format.

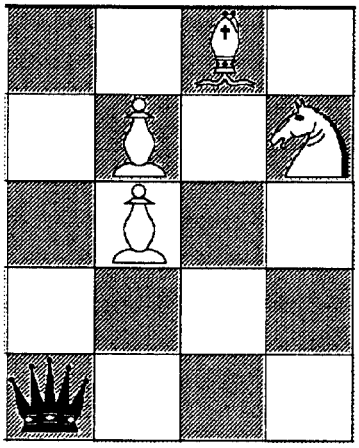


Figure 2.3: Initial situation: Computer (black) to move

move it, such as the role it plays in the opponent's current plan. We will discuss more complex goal-recognition of this sort in section 3.6, and assume for now that only the three goals mentioned above are present.

2.2 Plan generation

CASTLE now invokes its *planner* to generate a sequence of moves that will achieve the goal of capturing the opponent's bishop. The planner consists of components which embody different approaches to plan construction (described in more detail in section 3.7). The planning method invoked here is CASTLE's forward-directed search planner, which simply looks ahead two moves to find a viable attack. In other words, this method checks to see if it can move a piece to an empty square from which the piece will be able to make the capture successfully:

TO COMPUTE: A plan for the current situation

DETERMINE: A piece to make an attack
which can move to an intermediate location
and an opponent's piece
which can be captured by the attacker
from the intermediate location
such that the opponent has no defense

To determine whether the capture will be successful, the planner invokes CASTLE's *counterplanning* component to determine whether the opponent would be able to defend against the threat. If CASTLE cannot determine a reaction that the opponent would be able to make in response to the threat, the planner will assume that the attack will be successful. This assumption is consistent with the system's belief that its counterplanning component

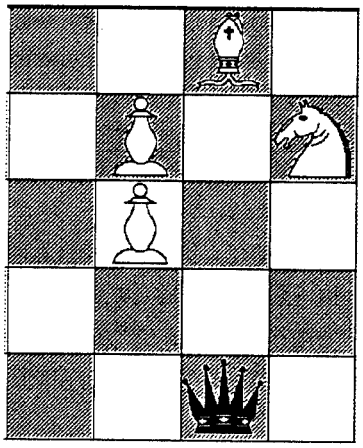


Figure 2.4: After the first plan step: Opponent (white) to move

is complete, that is, that it can determine a response to a threat whenever one exists. To the degree that this assumption is correct, the planning rule will always generate plans that will succeed. In addition, using the counterplanner both in actual counterplanning and in projecting the opponent's ability to respond to attacks benefits the system in two ways. First, the resulting planning process is more efficient, because the system uses specialized knowledge of reactions to threats to avoid having to examine all possible opponent actions as responses to the computer's move. More importantly, however, it enables the system to learn new counterplanning methods, as will be seen later in this chapter.

In our current example, this method will consider the plan of moving the queen two squares to the right, and capturing the bishop on the subsequent turn. The planner would then invoke the counterplanner to see if the opponent would have a possible response to the threat on its bishop. It is clear in figure 2.4 that the opponent would in fact have a possible response, namely interposing its knight to block the attack. If, however, the system lacks a counterplanning method for interposition, this response will not be generated. The system will therefore believe that the opponent will not have a response to the threat, and that its plan will succeed.

More specifically, suppose that CASTLE's counterplanner is equipped with the two counterplanning methods mentioned earlier, one for *running away* from a threat:

TO COMPUTE: A counterplan to a capture

DETERMINE: A move from the targeted square
to a square which cannot be attacked

and one for reacting to a threat by *counterattacking:*

TO COMPUTE: A counterplan to a capture

DETERMINE: A piece on the board that can capture the attacker

The opponent cannot respond by running away, because the bishop is blocked against the edge of the game-board by the pawn and the knight. The opponent is also unable to respond by counterattacking, because none of its pieces can attack the computer's queen in its new location. CASTLE concludes from this that the two-move plan to capture the opponent's bishop will succeed, and activates the plan to be considered for execution.

At this point (still before any move has been made) the system has two possible plans, one for capturing the opponent's knight and one for capturing the opponent's bishop, both of which it believes are sure to succeed. Since there are no active goals which are more valuable than the goals of the plans that have been generated, CASTLE will not attempt any more planning.

2.3 Plan selection

CASTLE's task is now to select which possible plan it will execute. CASTLE does this using its *plan preference* component, which (as its name suggests) applies knowledge of priorities between plans. The knowledge in the preference components forms a partial order of plans and goals, and the plan selection process is to pick a plan which is at least as good as all other possible plans.

This plan selection process is achieved by a rule that says roughly:

IF **?P** is the possible plan for a goal
which is preferable to the other possible plans for goals
and **?M** is the first step in **?P**

THEN Activate **?P** as the current plan
and activate **?M** as the move to make

The determination of preference between plans is carried out by CASTLE's *plan evaluation* component, which evaluates each plan and goal and returns a value to use in selection.[3] In our example, since both of CASTLE's plans to capture pieces are (it believes) guaranteed to succeed, they are evaluated simply as the values of the pieces that they capture. (Other plans would be evaluated differently if they were not guaranteed, or if they satisfied goals other than capturing pieces.)

The plan selection rule given above chooses the best plan and makes two assertions as a consequence. The first of them is to activate the plan as "current." The second of them is to activate the first move in the plan as the move to make. The second of these assertions, of the move to make, is used by the execution module; the assertion of the currently active plan is used for reasoning on subsequent turns.

[3]There are limitations to this approach, due to the well-known problem of assigning numerical values to various attributes of a plan and goal. In section 3.8 we discuss alternative approaches.

2.4 Expectation posting

Before executing the plan that it has selected, CASTLE invokes its *expectation generation component* to generate *expectations* that will monitor the plan's progress. These expectations are used not only to monitor the results of the plan steps that CASTLE executes, but also to monitor the correctness of the assumptions underlying its decision-making. These expectations must be as operational as possible, because they will be checked throughout plan execution. In other words, it is important to minimize the computation required to determine their truth or falsehood, because there are likely to be a large number of them being checked at any point during the execution of the program.

The most obvious expectation to post regarding any multi-step plan is that the plan itself will be executed properly. An expectation of this type is useful in the execution of complex plans [Doyle *et al.*, 1986] as well as providing an avenue to learning from their failure [Schank, 1982]. To accomplish this, CASTLE checks whether the currently active plan has more than one remaining step, and if so posts an expectation that the next step will be executed. In other words, whenever one plan step is executed, CASTLE generates a new expectation that the following step in the plan will be executed in turn. The expectation generation method which generates these expectations says roughly:

TO COMPUTE: An expectation to monitor

DETERMINE: The system's currently active plan
and the next move in that plan
and expect it to be executed next turn

In our example this rule will post an expectation that the computer will, on its next move, capture the opponent's bishop with its queen.

One major assumption that CASTLE made in constructing its plan is, as we discussed, that its counterplanner is complete. More specifically, CASTLE made the assumption that if there existed a move for the opponent which would disable the attack, then the counterplanning component would generate the move. From the perspective of learning, it is exactly assumptions such as this that should be monitored, because the failure of such an assumption will correspond to an opportunity to improve the performance of the relevant component. This assumption does not require an additional expectation to be monitored, however, because the failure of the assumption can be detected via failure of the expectation discussed above that the second move of the plan would be carried out. The reason for this should be clear: if there were a counterplan to the second move of the plan, and the opponent executed it, the consequence would be that the second move of the plan would not be carried out, and the expectation would fail. We will see in chapter 4 that when CASTLE's inference applies the expectation rule shown above, it records a link to the assumptions that underly each of its antecedents. Among these assumptions is the completeness of the counterplanning rule set. These links are later used when the expectation fails to diagnose which assumption is at fault. In chapter 6 we will see examples of different types of concepts being learned from the failure of the same expectation.

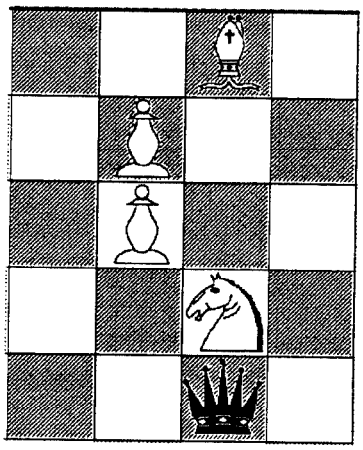


Figure 2.5: After the opponent's move: Computer (black) to move

In other situations, however, there will be underlying assumptions about the computer's components that must be monitored separately. These could include such assumptions as *I will have a counterplanning rule for every threat that may arise, I know all the ways that attacks can be enabled, No other plan will supercede the one I have activated,* and *No other plan would have been a better choice.* Posting expectations to monitor these types of assumptions will be discussed in chapter 3, and examples will be given in chapter 6.

2.5 Plan execution and expectation monitoring

At this point CASTLE is ready to execute its plan to capture the opponent's bishop. The plan it has selected, moving the queen over and then subsequently capturing the bishop, has been activated, and the first move in the plan has been designated as the move to make on this turn. It is now time to execute the move.

In chess, of course, there is little involved in executing a move. CASTLE updates its description of the board to reflect the situation after the move (shown in figure 2.4), increments its time counter, and allows the opponent to make its move. In other domains, such as robotic movement or device design, this execution of the computer's plan may be more complex, involving execution-time decision-making.

In the course of executing the steps in its plans, CASTLE must monitor the success or failure of its expectations. In our example, the only expectation that has been posted is that its next move will be to capture the opponent's bishop. Since CASTLE can't yet determine whether this is true or false, the expectation is not resolved at this time. The opponent makes its move, which of course is to block the attack with its knight. This interposition results in the board situation shown in figure 2.5. CASTLE must now recognize that its expectation has failed.

CASTLE determines the success and failure of its expectations using its *expectation*

monitoring component, which resolves the consequences of expectations. The expectation monitoring rule which recognizes the expectation failure in our example says roughly:

IF There was an expectation of a move to be made
and the expectation has not been resolved yet
and a different move was made

THEN Note that the expectation has failed
and handle the expectation failure

Noting that the expectation has failed will ensure that the expectation failure isn't handled more than once, and "handling the expectation failure" invokes CASTLE's learning mechanism.

In our example, where the expectation is a move to be made by the system, one possible objection to this expectation failure recognition rule is that it will recognize the expectation failure only after the system has selected another move in place of the expected one. In our example, this means that after the opponent makes its move, CASTLE will invoke its decision-making mechanism to choose a move, and it is only after a move has been selected and executed that CASTLE will notice that its expectation has failed. One solution to this somewhat counterintuitive behavior is to equip CASTLE with an expectation failure recognition rule which says:

IF There was an expectation of a move to be made
and the expectation has not been resolved yet
and the move is no longer able to be made

THEN Note that the expectation has failed
and handle the expectation failure

With this rule added, CASTLE can recognize the expectation failure at the time that the opponent interposes the knight. This approach has a problem, however, in that the number of rules required to recognize expectation failures increased dramatically when all rules of this sort are added, because rules are needed for each way in which the move might end up not being made. CASTLE sticks with the first approach, at the price of often recognizing expectation failures later than they might be recognized. Note that CASTLE is not concerned with expectation failure *recovery*, in the sense of replanning to compensate for erroneous expectations, but rather with learning from the failures, so delayed detection may be less costly than in other systems. Recovery in CASTLE is handled simply by replanning once the system notices that its plans cannot be continued.

2.6 Failure diagnosis

At this point in our example, CASTLE knows that its expectation has failed, and that the failure occurred because the move can no longer be made. Furthermore, it has a description of the situation leading up to the failure. Additionally, it has a *justification* for its belief

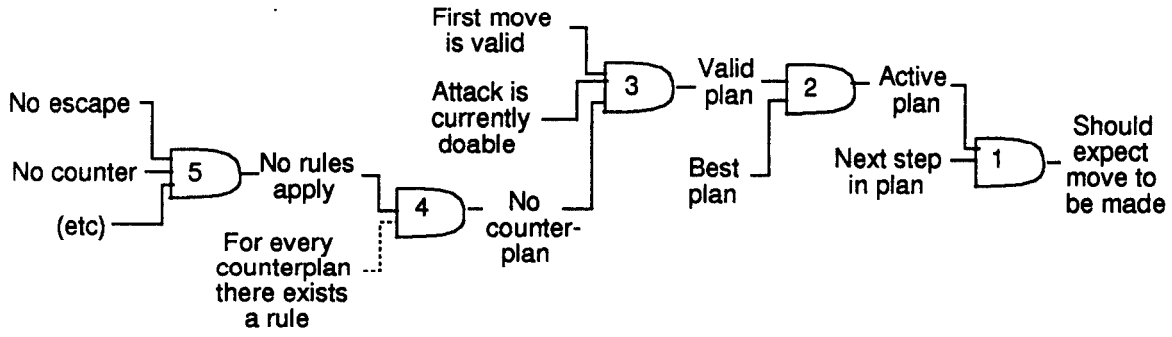


Figure 2.6: The justification structure for the expectation

that the expectation was correct, which is a record of the reasoning it used in generating the expectation, along with the assumptions that underly the correctness of the decision-making processes that were used [Collins *et al.*, 1989; Birnbaum *et al.*, 1990]. Armed with this information, CASTLE's *diagnosis engine* must now determine which of its decision-making components is at fault.

The basic process of failure diagnosis is a simple recursive procedure. If an expectation fails, CASTLE's diagnosis engine should check whether any of the antecedents of the rule which generated the expectation are contradicted by the facts of the case. If so, it should fault the antecedent (or antecedents) which have been contradicted, and recur. Otherwise, the problem lies in the assumptions associated with the rule itself [Birnbaum *et al.*, 1989].

This recursive procedure may of course be complicated by being unable to determine at every point which supporting belief to fault. If this determination cannot be made, CASTLE will nondeterministically choose one of the possible faults, and backtrack in the diagnosis search space when necessary. We discuss CASTLE's approach to avoiding this problem in chapter 4. In our example we will see that the diagnosis is in fact deterministic.

In examining the truth of each assumption, the diagnosis engine has more information at its disposal than the system had when the decision-making was originally performed. For example, if an assertion was made that a move would be legal in a projected future situation, the diagnosis engine can test the legality of the move against the situation which actually arose instead of the projected situation. If the two differed, this would signal CASTLE to diagnose the projection mechanism itself. We will discuss inference such as this in detail in chapter 4.

The justification structure underlying the system's belief in the expectation generated in our example is shown in figure 2.6. The conjunctions in the figure correspond to logical implications, in which the set of "incoming" assertions imply the truth of the "outgoing" assertion. Links shown as dotted lines represent implicit assumptions whose truth was not checked when the inference was made, but whose correctness underlies the implication's being correct. The `should-expect` expression corresponds to CASTLE's belief that it should expect the attack on the bishop to be made. The conjunction labeled **1** corresponds to the expectation generation method discussed on page 21, which said to expect the next step of

the currently active plan to be executed. Because the expectation has been faulted, and the attack is in fact the second move of the active plan, the fault must lie in the fact that the plan was active. Faulting this leads us to examine the conjunction labeled **2**, which represents the plan selection rule from page 20. Since the plan was in fact the best plan considered, the fault lies with the belief that the plan was valid, which brings us to the conjunction labeled **3**. This conjunction corresponds to the planning rule on page 18. Of the antecedents of this rule, the one which is incorrect is the belief that there was no move for the opponent which would disable the capture. Because this belief is now known to be false (since the opponent did in fact have a counterplan), and it is true that no counterplanning method applied to the situation, the diagnosis engine will conclude by faulting the assumption that CASTLE's counterplanner is complete.

2.7 Planner repair

Once CASTLE has determined that the plan failure is due to its set of counterplanning methods being incomplete, it knows that it needs to construct a new rule to perform counterplanning, i.e., a new rule which will be used whenever the counterplanning component is invoked. This rule should encode the type of counterplan that the opponent used in the situation that caused the failure. To construct this rule, CASTLE uses explanation-based learning [Mitchell *et al.*, 1986; DeJong and Mooney, 1986] to generalize a description of the move that the opponent made.

CASTLE carries out this process using *component performance specifications.* These specifications describe the correct behavior of each component, in a way that can be recognized after the fact in a situation in which the component should have produced that behavior. In other words, when the system has reason to believe that a component should have behaved differently than it did in a given situation, based on information from the diagnosis engine, this specification will enable the system to determine how the component should have behaved, and to construct an explanation of why it should have done so. An explanation of the correctness of the specification as applied to a particular situation can then be generalized using EBL, and the antecedent of a new rule can be constructed.

A performance specification for the *counterplanning* component says roughly:

TO COMPUTE: The reason that a move was a counterplan to an attack

DETERMINE: How the move disabled the attack on the previous turn

Since this specification is only used after-the-fact to explain why a particular move is a correct counterplan, the computation of *move disablement* need not be implemented in a way which can be evaluated before the fact. In other words, even without a specific rule for disabling an attack by interposing a piece, the system has the ability to observe the game board sequence after the fact and use its domain knowledge to realize that the attack was blocked. It does this by examining what conditions had to be true for the attack to succeed, and what effect of the disabling move caused one or more of the preconditions not to be met. Reasoning of this type is too complex to be used in planning, because it involves

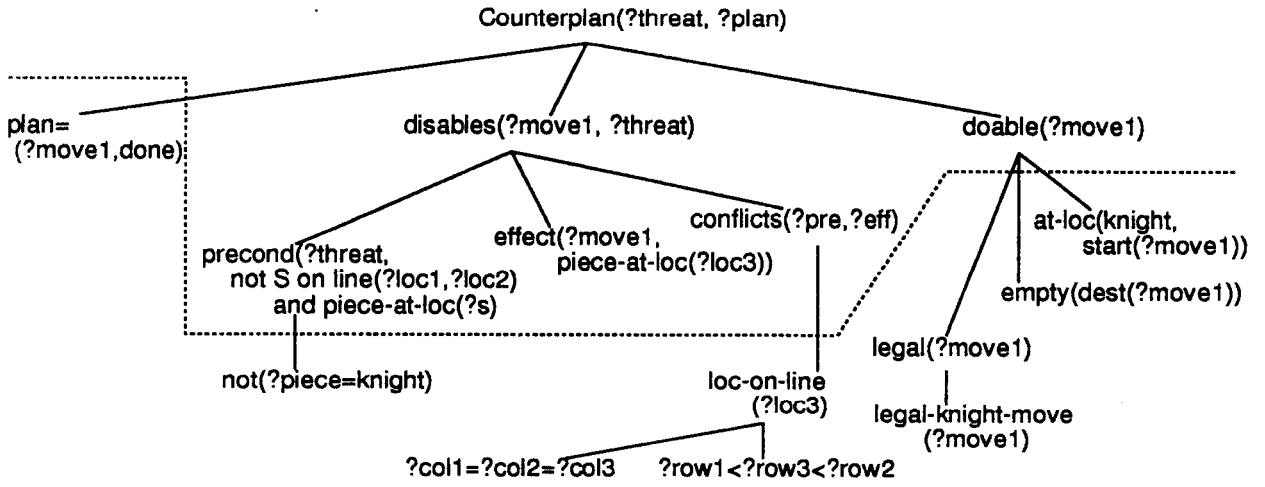


Figure 2.7: Explanation of desired counterplanner performance

enumerating all possible moves, as well as all effects of those moves, and comparing them to all the preconditions of the threat. This reasoning can, however, be used in after-the-fact analysis.

The explanation of why the opponent's move was a good counterplan is shown in figure 2.7. The chain of significant steps in the explanation is, roughly:

- The opponent counterplanned against my queen-bishop attack, *because*
- The attack by my queen on the opponent's bishop was disabled by the opponent's move, *because*
- A precondition of my attack was that the line of movement be clear, *and*
- An effect of the opponent's move was that a square that was vacant was filled, *and*
- The square filled by the opponent is on the line of attack.

Explanation-based learning is then invoked to generalize the details of the explanation that are not relevant to its being correct. For example, it is irrelevant what type of piece is interposed, because all links in the explanation tree which relate to the interposed piece will remain valid for any type of piece. On the other hand, it is important that the piece attempting the threat not be a knight. The system then collects the expressions in the proof tree which are as general as possible while being operational, meaning that they can be used effectively in planning. In our example, the operational expressions say that the counterplan was to make a move to a location on the line of attack, such that the move is legal, as long as the piece being blocked is not a knight.

These generalized proof-tree leaves will then form the antecedent of a new rule. Since the leaves imply the target concept, and the target concept implies the correct behavior of

```
(def-brule cp-learned-1
  (counterplan learned-cp-meth-1 ?player                    To counterplan against
     (goal-capture ?piece3 (rc->loc ?r2 ?c2)                an opponent threat
       (move ?other-player (capture ?piece3) ?piece         against a piece
             (loc ?r1 ?c1) (loc ?r2 ?c2)))
    ?time ?counterplan)
<=
 (and (= ?counterplan                                       plan to make a move
        (plan (move ?player move ?other-piece
                    ?other-loc (loc ?r3 ?c3)) done))
      (not (= ?piece knight))                               If the piece isn't a knight
      (loc-on-line ?r3 ?c3 ?r1 ?c1 ?r2 ?c2)                 to a square on the line
      (move-legal-in-world                                  of attack
          (move ?player move ?other-piece                   if there's a piece that can
                ?other-loc (loc ?r3 ?c3))                   move to that square
          (world-at-time ?time))
      (at-loc ?player ?other-piece ?other-loc)              and the square is empty
      (not (at-loc ?any-player ?any-piece (loc ?r3 ?c3))) ))
```

Figure 2.8: The new counterplanning rule: interposition

the counterplanner, the leaves are sure to specify a correct counterplanning method. The new counterplanning rule for our example is shown in figure 2.8,[4] which says roughly:

TO COMPUTE: A counterplan to an attack

DETERMINE: A location on the line of attack
that another piece can move to

When this rule is added to the set of counterplanning rules, CASTLE will predict the opponent's response correctly in a situation where interposition is viable, such as the situation of figure 2.4. More importantly, because the same set of rules is used for the system's own counterplanning against opponent threats, CASTLE will be able to respond to threats in a new way. Before learning this new rule, CASTLE was unable to counterplan in situations where *running away* and *counterattacking* weren't possible. Now CASTLE can counterplan by interposing a piece.

To test the generality of the learning process, as well as of the learned rule itself, we repeated the process described above in three new board configurations, all of which involved a possible opponent interposition. The three board sequences are shown in figure 2.9. In each case the diagnosis engine traced the failure back to the counterplanning component, and the learner constructed the appropriate rule. Additionally, rules learned in each of the examples was able to be used in the others. We will see examples of CASTLE learning altogether different rules of other types in chapter 6.

[4]The rule format is discussed in section 3.1.

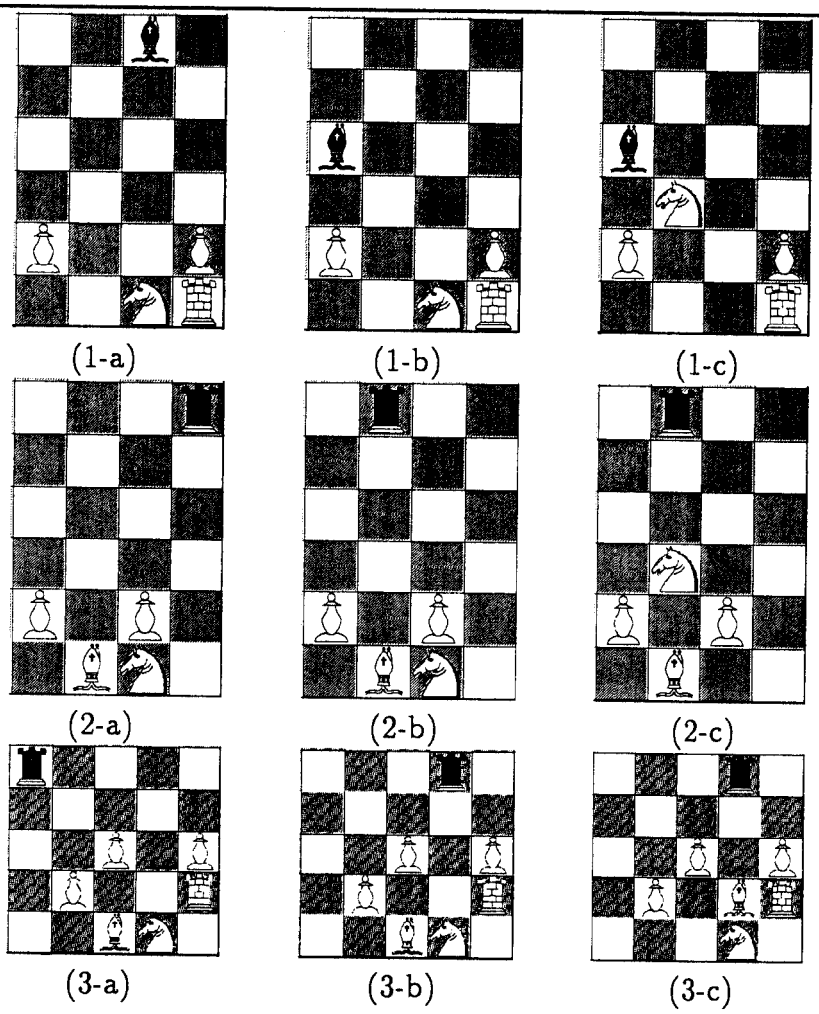


Figure 2.9: Variant examples of interposition

2.8 What we've seen

Several points have been illustrated in the *counterplanning* example described in this chapter. First, the sequence of steps that make up CASTLE's decision-making processes have been described. These steps include interpreting the environment, generating goals, constructing plans, selecting the best plan, and executing it. Plan execution includes several steps which aid execution and also facilitate learning, such as expectation generation and monitoring. Expectation failures are diagnosed, and new rules are constructed to repair the planner.

Several types of knowledge were used in the course of the example:

- CASTLE's decision-making procedures are explicitly encoded in rules that the system is able both to invoke and to decompose and reason about.
- A justification structure is maintained showing why the system believes and expects what it does. Similar justification structures exist for all the system's rules,

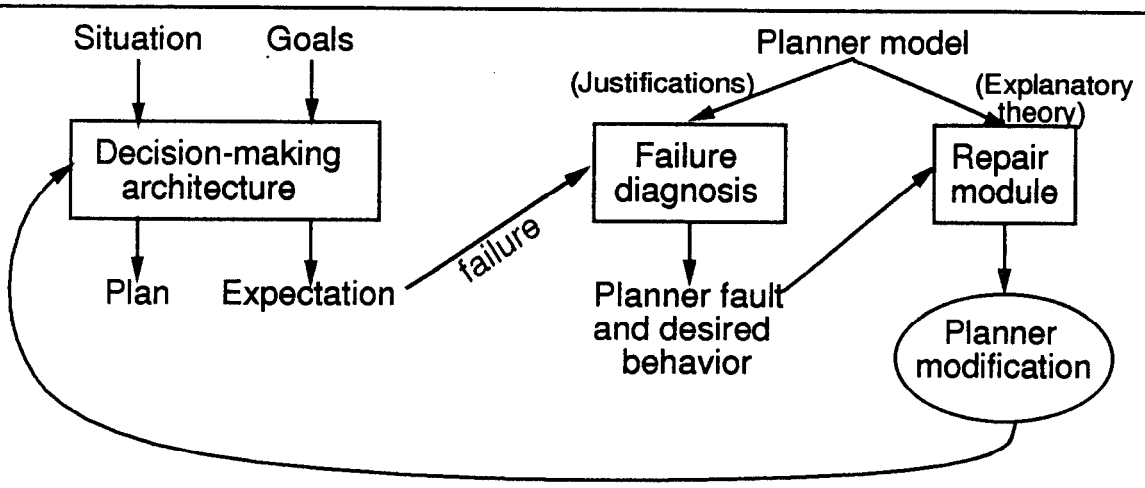


Figure 2.10: The learning process in CASTLE

enumerating the assumptions underlying the system's belief that the rule's antecedents imply its consequents.

- Specifications exist for each component in the system, describing the correct behavior of the component.
- An explanatory model is used to explain desired planner component behavior.

The next three chapters will present these processes and models more generally.

Chapter 3

A decision-making architecture

CASTLE's breakdown of the decision-making process into distinct tasks, as briefly described in chapter 2, is shown in figure 3.1. Each of these tasks is implemented in CASTLE's architecture by one or more *components*. Each component consists of a set of rules comprising the system's knowledge of how the task of that component can be carried out. We discussed each of CASTLE's components briefly in the previous chapter, and discussed the rule set and specification for the *counterplanning component* in somewhat greater detail. In this chapter we will examine each of CASTLE's other components in more depth.

3.1 Control flow in CASTLE

Decision-making in CASTLE is controlled by *meta-rules*. These meta-rules are executed in sequence by CASTLE's control program, and their job is to control the invocation of the inference rules that constitute CASTLE's components [Davis, 1980; Genesereth, 1983;

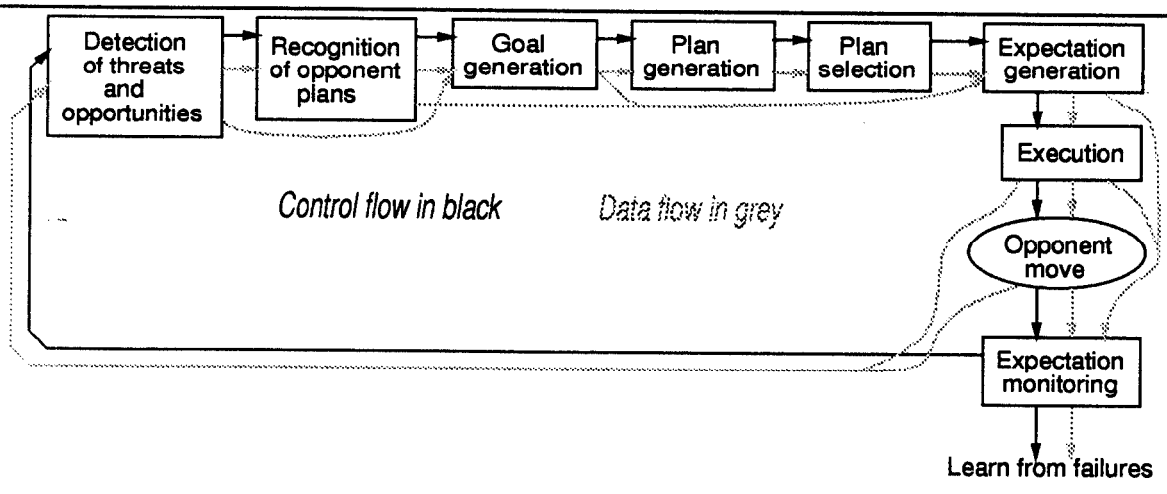


Figure 3.1: CASTLE's decision-making process

```
(def-rule comp-counterplan comp-plans
 (and (active-goal opponent ?opp-goal current-time)
      (counterplan computer ?opp-goal current-time
                   ?counterplan))
=>
 ((possible-plan computer current-time ?counterplan
                 (goal-counterplan ?opp-goal))))
```

RULE: COMP-COUNTERPLAN
ANTECEDENT: If there's an active opponent goal, and I can make a counterplan,

CONSEQUENT: The plan is possible to execute

Figure 3.2: Deciding to respond to a threat (counterplan)

Wilensky, 1981]. This involves deciding when the various components should be invoked, as well as what information should be passed to them. The meta-rules operate in a forward-chaining fashion: Upon execution, they query the system's inference engine to determine the status of their antecedents ("left-hand-sides"), and if these queries resolve as true, the rules then assert their consequents ("right-hand-sides") into CASTLE's propositional database.

Most of CASTLE's meta-rules are very straightforward, such as the rule shown in figure 3.2 (which we discussed in chapter 2) which says roughly:

IF The opponent has an active goal
and I can generate a counterplan against it

THEN Assert that the counterplan is a possible plan.

The rule definition in the figure can be read as follows: "Define a forward-chaining rule named `comp-counterplan`, which is in the set of rules called `comp-plans`. The rule's antecedent is a conjunction of two queries [shown in the second and third lines of the figure], an `active-goal` query and a `counterplan` query. The rule's consequent [given in the last two lines] is a `possible-plan` expression that should be added to CASTLE's memory if the rule fires successfully."

CASTLE's inference engine is organized around a fairly standard deductive database [Charniak and McDermott, 1985]. *Queries* are made in the form of expressions containing variables, and the inference engine either succeeds, returning variable bindings for which the expression is true, or fails. These queries can be resolved in three ways. The first is to use memory retrieval, which searches CASTLE's propositional database for a belief that matches the query. The second way that queries are resolved is by backward-chaining rules, called `b-rules`, which are invoked when a query is made that matches the rule's consequent [Hewitt, 1969]. When a `b-rule` is invoked, it is resolved by recursively forming a query to determine the truth of its antecedent. If the inference engine can determine a set of bindings for which the `b-rule`'s antecedent is true, it will conclude that the query is true for that set of bindings.

The third way that queries are resolved is to invoke special-purpose LISP code. This is primarily used to achieve faster execution than is possible with interpreted rules. It is additionally used to perform meta-level computation involving other queries. These uses will be described in more detail as they arise.

```
(def-brule counterplan-2
  (counterplan cp-counterattack ?player                          To counterplan
               (goal-capture ?target ?target-loc                 against an
                      (move (move ?opp-player (capture ?target)  opponent attack
                                  ?opp-piece ?opp-loc ?target-loc)))  against a
               ?time ?the-response)                              computer piece
<=
 (and (at-loc ?player ?piece ?loc ?time)                         Find a piece that
      (move-legal (move ?player (capture ?opp-piece)             threatens the
                  ?piece ?loc ?opp-loc))                         attacking piece
      (= ?the-response                                           and capture the
         (plan (move ?player (capture ?opp-piece) ?piece         attacker with it.
                     ?loc ?opp-loc)
               done)) ))
```

Figure 3.3: Reacting to a threat by counterattacking

As we said earlier, CASTLE's meta-rules control the invocation of the backward-chaining inference rules in CASTLE's decision-making components. Each component's inference rules are invoked using a specific query format which they all share. Meta-rules invoke components by making queries in the specified format of the component, which are then resolved using the component's rules. For example, when the meta-rule in figure 3.2 is applied, two queries are made to CASTLE's inference engine. The first, the `active-goal` query, is resolved by searching the system's memory for propositions previously asserted by the goal-generation meta-rules. The second query, the `counterplan` query, is resolved by invoking the backward-chaining rules in CASTLE's counterplanning component. One such rule, for counterplanning by counterattacking, is shown in figure 3.3. This rule says roughly:

TO COMPUTE: A response to a threat of capture

DETERMINE: A piece on the board that can capture the attacker

The rule definition in the figure can be read as follows: "Define a backward-chaining rule named `counterplan-2`. The rule's consequent should be the `counterplan` expression [the next five lines of the definition], which is matched against the queries made to the system's inference engine to determine if the rule should be invoked. The rule's antecedent [which is shown after the back-arrow] is the conjunction of three queries, the `at-loc` query referring to pieces at board locations, the `move-legal` query referring to the legality of moves, and the = query which constructs the response out of the information computed by the previous two queries."

These rules fit together in CASTLE as follows: When the `counterplan` query in the meta-rule of figure 3.2 is made, it is matched against the consequent of the rule in figure 3.3, which succeeds in matching with the bindings {`?cp-method=cp-counterattack, ?player=computer, ?time=current-time, ?the-response=?counterplan`} and with the variables relating to the goal matched to an active opponent goal. The system will then

`(at-loc ?player ?piece ?loc ?time)`	*A player has a piece at a location*
`(active-goal ?player ?goal ?time)`	*A player has an active goal*
`(possible-plan ?player ?time ?plan ?goal)`	*A player has a possible plan to satisfy a particular goal*
`(active-plan ?player ?time ?plan ?goal)`	*A player is executing a plan for a particular goal*
`(move-to-make ?move ?player ?goal ?time)`	*A player made a particular move in service of a goal*

Figure 3.4: Some belief types in CASTLE's memory

attempt to resolve the query by recursively querying the inference engine to determine whether the rule's antecedent is true, and if so, under what conditions. This will perform the computation necessary to generate a counterplan. Three queries are made in the course of this rule execution, corresponding to the three conjuncts of the rule antecedent in figure 3.3. The first, the `at-loc` query referring to the location of a potential counterattacking piece, is resolved using database lookup. The second, `move-legal`, is resolved using backward-chaining rules which encode the moves that are legal in the game. The third, `= (''equals'')`, is a primitive query which equates two expressions and binds all variables appropriately, if possible. If all of these queries resolve consistently, the `counterplan` query will succeed. This will cause the meta-rule to succeed, and CASTLE will assert that the counterplan discovered is a possible plan.

To summarize, most of the decision-making in CASTLE is performed by the backward-chaining rules which resolve the queries. The forward-chaining rules are almost exclusively meta-rules which are designed to control the flow of execution. While most of CASTLE's meta-rules are at about the same level of complexity, some of them are more complex, such as those that invoke the various planning techniques used to generate sequences of moves. The more complex meta-planning rules, which we will discuss further in section 3.7, are designed to invoke the appropriate decision-making components without wasting time computing plans that are inappropriate for the current situation.

3.2 CASTLE's basic representations

Figure 3.4 shows some of the types of beliefs that CASTLE maintains in its propositional database. Each of these represents information that is maintained over time and which is not local to the computation of a particular component. Most of these beliefs are used to store the results of a component's computation, and are then referenced by other components later. These beliefs are asserted by CASTLE's forward-chaining meta-rules.

The `at-loc` beliefs are asserted by CASTLE's board update rules, which maintain the

goal ← (goal-capture *opp-piece loc goal-info*)
← (goal-counterplan *opp-plan*)
← (goal-limit *piece loc goal-info*)
← (goal-none)
goal-info ← (move *move*)
← exposed
← restricted

Figure 3.5: Representing goals in CASTLE

system's knowledge of the state of the board at each time. The rest of the belief types shown represent the output of the goal activation, plan generation, and plan selection components. CASTLE represents elements such as plans and goals as structured objects that look syntactically like logical propositions.

The simplest of these is the representation of primitive chess moves. These moves are not necessarily moves that have been made, or moves that relate to goals or plans under consideration, but simply the actual moves that are physically made on the board. Moves are represented as:

(move *player move-type piece loc-start loc-end*)

where *move-type* is either the symbol move or the item (capture *opp-piece*). Locations are also structured items, of the form (loc *row col*). Putting these representations together, the computer's intended move in the interposition example from chapter 2, moving its queen from row 5 column 3 to capture the opponent's bishop at row 1 column 3, would be represented as:

```
(move computer (capture bishop) queen (loc 5 3) (loc 1 3))
```

Note that this is the representation for the move itself, not of the fact that he made the move at a particular time.

CASTLE represents goals using the taxonomy shown in figure 3.5. The most basic type of goal is a goal-capture goal, which represents a goal to capture a particular opponent piece. Goal-captures include information about what led to their activation in their goal-info slots, which is often useful in developing plans to satisfy them. In the simplest case, a goal-capture may have been activated opportunistically, just because there is a move that the system can make that will capture the targeted piece. In such cases, planning requires no inference and is merely a matter of transforming the information in the goal into the format of plans. We will see examples of other rules that make simple transformations when we discuss plan generation in section 3.7. The key point is that the goal-info slot provides a means of communicating information between the goal activation component and the plan generation component, which enables the system to maintain autonomy of function and avoid duplication of effort that would result if the planner had to infer information that was already available to the goal generator. Other forms of goal-info currently used

plan ← (plan *move rest*)
rest ← done
← (next *move rest*)
← (move-pred (var *var*) *exp rest-then rest-else*)

Figure 3.6: Representing plans in CASTLE

by the system are that the targeted piece is exposed or is restricted in its movement, both of which signal that the piece may be easily captured, and that particular planning methods may be apropos. Other goals that CASTLE represents are goal-counterplans, which represent the goal to counterplan against an opponent's plan, and goal-limit goals, which represent the desire to limit an opponent's options.

The final complex structures represented in CASTLE are plans, which are represented using the taxonomy shown in figure 3.6. Plans consist of two parts: the first move to make in the plan, and an expression that can determine the rest of the plan. The plan may consist of only one move, in which case the rest of the plan is given as the symbol done. If the system has already determined the next move to make, it is represented as a next item with slots for the next move to make and the subsequent rest of the plan.

A more complex plan is one in which the move to make next must be determined after the opponent has made the intervening move. In this case the rest of the plan is of the type move-pred, and consists of an expression to evaluate as a function of the move that the opponent made.[1] The expression could of course also reference any information in the system's memory, but the opponent's move is given as a parameter for ease of reference. The system will then choose between two alternate continuations of the plan based on whether the expression evaluates to true or false. The expression references the move that the opponent made via the variable var which is given in the move-pred construct.

This construct for specifying conditional plans could be used to represent a plan for the computer (playing white) in the situation shown in figure 3.7:

```
(plan (move computer pawn move (loc 1 2) (loc 2 2))
     (move-pred (var ?opp-move)
       (= ?opp-move
          (move opponent rook ?move-type (loc 3 1) ?loc2))
       (next (move computer pawn (capture knight) (loc 2 2) (loc 3 1))
             done)
       (next (move computer pawn (capture rook) (loc 2 2) (loc 3 3))
             done) ))
```

The first move of the plan is to advance the pawn by one square. The opponent would then

[1]Conditional plans of this sort are very difficult to construct and reason about (but see, e.g., [Warren, 1976; Pryor and Collins, 1993]). CASTLE uses them only in scripted, predetermined ways.

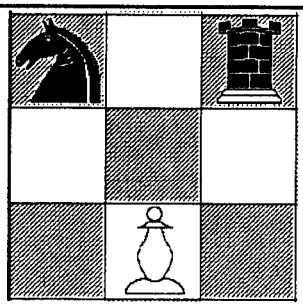


Figure 3.7: A conditional plan for white

be expected to move one of the two pieces out of threat. If the rook were unmoved, the computer should capture it, otherwise the knight should be captured. This is represented as a `move-pred` rest of the plan, in which the expression evaluates true if the opponent moved the rook. The rest of the plan for the true condition (if the rook was moved) is to capture the knight, and the rest of the plan for the false condition (if the rook was unmoved) is to capture the rook.

3.3 Component descriptions

CASTLE's backward-chaining rules are organized into *components* which perform the required tasks that constitute the system's intentional repertoires. To understand the format in which components will be presented in this chapter, consider the *counterplanning* component which we have been using as our example. Figure 3.8 gives a description of the most important aspects of the component. The *specification* is the system's description of the component's function, in this case that it generates plans which disable a plan of the opponent's. As we saw briefly in chapter 2, this specification is used in the construction of new rules for the component. The *methods* in figure 3.8 are samples of the rules currently in the component's rule set, each of which embodies a different approach to achieving its specified task. For example, the three counterplanning methods shown here (running away, counterattacking, and interposition) were discussed in chapter 2.

Component name	Counterplanning
Specification	The plan results in the opponent's goal not being satisfied
Methods	Run away, counterattack, interposition, ...
Invocation	`(counterplan ?method ?player ?goal ?time ?plan)`
Subcomponents	Threat and opportunity detection
Used by	Meta-planning, plan generation

Figure 3.8: The counterplanning component

The *invocation* of the component specifies the form of the query that must be made to CASTLE's inference engine in order to invoke the component. All the component methods are backward-chaining rules whose consequents ("right-hand sides") match the component invocation, so that querys to the inference engine of the specified form will invoke the component's methods. We will in one case discuss a set of forward-chaining rules as a component. When we do so the invocation slot will be replaced by an *assertion* slot which specifies the forms of the propositions that are asserted into CASTLE's propositional database when the rules are successfully applied.

The last two lines in the description of the counterplanning component, *subcomponents* and *used by,* show the other components with which the counterplanning component interacts. The subcomponents, such as threat and opportunity detection, are those components which are invoked in the course of executing the counterplanning component's rules; the components listed in the "used-by" slot are those components which invoke the counterplanner. We will see that this interleaving of CASTLE's components allows for complex decision-making behavior, and plays a key role in achieving cross-task transfer of learned concepts. For instance, in chapter 2, the failure that occurred when the counterplanning component was invoked by the plan generation component led to the acquisition of a new method that could also be used when counterplanning is invoked by the meta-planning component.

The rest of this chapter will discuss each of the components in CASTLE's architecture, which was shown in figure 3.1. For each component we will present a table of summary information for the component, and will examine meta-rules that invoke the components as well as sample component methods.

3.4 Interpreting the environment

The first decision-making task that CASTLE performs is threat detection. The *detection* component has the task of noticing threats and opportunities as they become available. Rather than recomputing these at each turn, CASTLE maintains a set of active threats and opportunities which is updated over time. To carry out this incremental approach to threat detection, the system uses a *detection focusing* component, which consists of *focus rules* that specify the areas in which new threats may have been enabled. The detection component itself, which consists of rules for noticing specific types of threats, is then invoked under the constraints given by the focusing component in order to determine the threats that have been newly enabled. The focusing component can be thought of as computing "where to look," while the detection component computes "what to look for." This interaction between components is shown pictorially in figure 3.9. The focusing rules examine the events that have occurred and generate bindings which describe the areas of the board in which new threats may exist. The detection rules then search the current situation for new threats, within the constraints imposed by the focusing bindings. This strategy is implemented by the forward-chaining rule shown in figure 3.10, which says roughly:

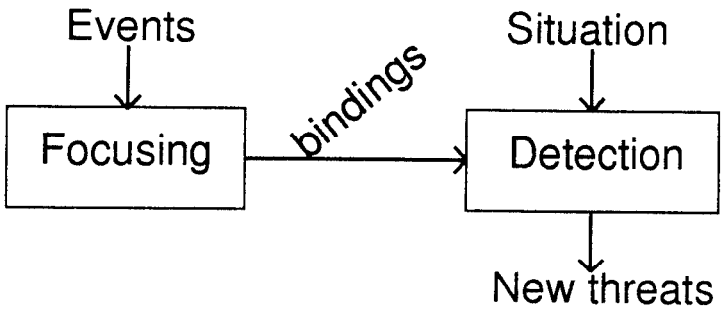


Figure 3.9: Incremental threat detection

IF There is a threat set from the previous turn
and all areas to focus on for new threats are determined
and all the new threats in those areas are generated
and all threats that can no longer be executed are filtered out

THEN Record the new set of possible threats as **old + new - disabled**

The two conjuncts that compute the newly enabled threats, namely the `focus` and `is-threat-in-world` expressions, invoke the two subcomponents of figure 3.9 that perform the incremental detection. The bindings are passed between the two components by CASTLE's unification mechanism.[2]

[2]These constraints could instead be passed explicitly as parameters of the two components [Collins *et al.*, 1993, Fig. 1]. This would allow some constraints to be processed more efficiently (namely those that do not correspond to single-variable constraints). This is discussed in more detail at the end of section 6.2.

```
(def-rule focused-detection comp-threats                          If possible threats
  (and (computed-possible-moves ?player                           were recorded last
          (world-at-time (1- current-time)) ?prev-moves)          time step,
       (is-set-of ?move-set (var ?move)                           Construct a new set
         (and (or (in-set ?move ?prev-moves)                      that includes the old
                  (and (focus ?f-method ?player ?move             set, and also apply
                              (world-at-time current-time))       the focus rules and
                       (is-threat-in-world ?t-detector ?t-type    then apply the threat
                          (player-opponent player)                detectors within the
                          (world-at-time current-time))))         focus constraints
              (move-doable ?move (world-at-time current-time))))) such that all moves
=>                                                                are feasible
  ((computed-possible-moves ?player current-time ?move-set)))     record the new set
```

Figure 3.10: Incrementally computing the possible threats

Component name	Threat detection
Specification	Captures which are legal in the game
Methods	Queen capture, rook capture, ...
Invocation	`(is-threat ?method ?type ?player ?move ?world)`
Subcomponents	(None)
Used by	Meta-rules

Figure 3.11: The threat detection component

The threat detection component is described in figure 3.11, and a sample threat rule is shown in figure 3.12.[3] This rule specifies the threats that can be made by a rook:

TO COMPUTE: Threats that can be made by a particular rook

DETERMINE: Opponent pieces which are on the same row or column such that no square between the two pieces is occupied

The focusing subcomponent is described in figure 3.13. This component is invoked by the system's meta-rules prior to invoking the threat detection rules, and the binding constraints generated by the focus rules are then passed to the detection rules. A sample focus rule is shown in figure 3.14, which says that a moved piece can make new attacks from its new location:

TO COMPUTE: The location of potential new threats

DETERMINE: The new location of the most recently moved piece **and** the class of threats from this location

[3]For efficiency reasons these rules are actually implemented in LISP code rather than CASTLE rules, but correspond directly to rules such as the one shown here.

```
(def-brule threat-detect-rook
  (is-threat threat-detect-rook move-capture ?target-player        There is a threat
             (move ?player (capture ?target-piece) rook            by a player's rook
                   (loc ?row1 ?col1) (loc ?row2 ?col2))            against the enemy
             (world-at-time ?time))                                at a given time
<=                                                                 if the rook and the
  (and (or (= ?row1 ?row2) (= ?col1 ?col2))                        target are on the
       (no (and (loc-on-line ?interm-row ?interm-col               same row or column
                             ?row1 ?col1 ?row2 ?col2)              and if no square
                (at-loc ?either-player ?other-piece                on the line of
                        (loc ?interm-row ?interm-col) ?time))))    attack is occupied
```

Figure 3.12: A sample threat detection rule

Component name	Detection focusing
Specification	Moves enabled by the most recent event
Methods	New attacker, new target, discovered attack, ...
Invocation	`(focus ?method ?player ?move ?world)`
Subcomponents	(None)
Used by	Meta-rules

Figure 3.13: The detection focusing component

A similar focus rule (see appendix A) says that a moved piece can also be the target of new threats:

TO COMPUTE: The location of potential new threats

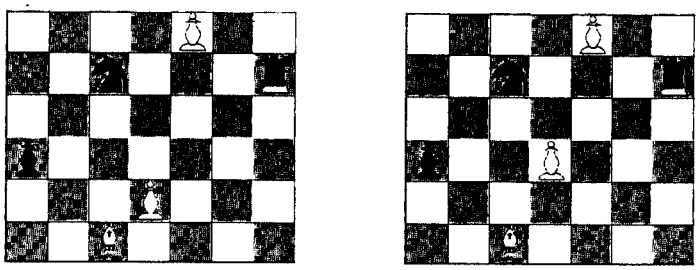
DETERMINE: The new location of the most recently moved piece
and the class of threats against that location

Figure 3.15 shows the opponent advancing a pawn from row 2 column 4 to row 3 column 4. When CASTLE uses its incremental detection mechanism to update its set of active threats and opportunities, the focus rules we have seen would generate two constraint specifications for the locations to be considered by the detection component: {?loc1=(3,4)} and {?loc2=(3,4)}. When these constraints are passed on to the threat detection component, CASTLE will look for threats subject to these constraints. The threat rule in figure 3.12, for instance, will respond to the constraint ?loc2=(3,4) by looking for a rook along row 3 or column 4 with an open line of attack to the newly moved pawn. Since the newly moved piece isn't a rook, the rule in figure 3.12 will not consider the case of ?loc1=(3,4), but a corresponding rule for detecting threats that can be made by a pawn will be invoked to check for new threats from this location.

It should be apparent that the focus rules given above are not themselves sufficient to enable CASTLE to detect all new threats. For example, in figure 3.15(b), the enabled threat by the opponent's bishop against the computer's rook will not be detected, because

```
(def-brule focus-new-source
    (focus focus-moved-piece ?player                                    Focus on
           (move ?player ?move-type ?piece ?loc1 ?loc2)                 threats
           (world-at-time ?time))                                       from the new
  <=                                                                    location of
    (move-to-make (move ?player ?prev-move-type ?piece ?old-loc ?loc1) the recently
                  ?player ?goal (1- ?time)) )                           moved piece
```

Figure 3.14: Focusing on threats enabled by the new location of a piece



(a) (b)

Figure 3.15: Enabling new attacks

neither the source or target of the threat are newly moved, so the detection rules will not be appropriately applied. We will see CASTLE learning a new focus rule for this type of *discovered attack* in section 6.2.

3.5 Plan recognition

The next task that CASTLE engages in is to attempt to recognize its opponent's plans. The *plan recognition* component, described in figure 3.16, has the task of guessing the plans that the opponent may be engaged in or considering. CASTLE's architecture currently jumps directly from the board situation to the plans that the opponent may be executing, without first predicting the opponent's goals. The reason for this is that the plan recognition methods that CASTLE employs are very situation-driven, and do not examine the system's situation to see what goals the opponent is likely to engage in. CASTLE generates its own plans differently, by first determining active goals and then generating plans for them.

The plan recognition component is invoked by the meta-rule shown in figure 3.17, which says roughly:

Component name	Plan recognition
Specification	The plan may be the one the opponent is engaged in
Methods	direct opportunities, script application, forward search
Invocation	`(recog-plan ?method ?player ?world ?plan ?goal)`
Subcomponents	threat detection, script application, ...
Used by	Meta-rules

Figure 3.16: The plan recognition component

```
[b]
(def-rule comp-plan-recog comp-opp-plans                          If I can
  (recog-plan ?method opponent (world-at-time current-time)       recognize a
              ?plan ?goal)                                        possible
 =>                                                               opponent plan,
  ((possible-opp-plan (world-at-time current-time) ?plan ?goal)))  record it
```

Figure 3.17: Invoking the plan recognition component

IF There is a plan recognition method
that can recognize a currently-possible opponent plan

THEN Record the possible opponent plan

The problem of plan recognition has been addressed in many areas of Artificial Intelligence, including story understanding [Charniak, 1972; Schank and Abelson, 1975; Schmidt and Sridharan, 1977; Wilensky, 1978; Kautz and Allen, 1986], computer-aided education [London and Clancey, 1982], and tactical decision-making [Azarewicz *et al.*, 1986]. In general most approaches to plan recognition fall into one of two categories: *top-down* or *bottom-up*. In top-down approaches, the system begins with a set of possible interpretations, and tries to determine which of them can account for the events with which it is presented. In a pure top-down approach, no analysis of the events is carried out except in service of confirming or denying the hypothesized interpretations. Put another way, a top-down approach to plan recognition is to first hypothesize plans which the opponent may be carrying out, and then check to see which (if any) of them match the situation. *Bottom-up* plan recognition is the opposite: Analysis begins with the events and the situation, and hypotheses are only suggested as a consequence of this analysis.

CASTLE's architecture allows both top-down and bottom-up plan recognition to take place. CASTLE's simplest method is a completely bottom-up one which represents any threats that were detected by the threat detection mechanism as plans that the opponent may execute. Other plan recognition methods are more top-down, using CASTLE's various planning mechanisms from the perspective of the opponent. All the methods that have been implemented operate by predicting plans from the current situation, but methods could be added that incorporate observed opponent moves into the plan recognition process [Charniak, 1972].

A very simple plan recognition method is shown in figure 3.18, which recognizes opponent plans to capture pieces through immediate attacks. As we discussed earlier, this rule is not performing any particularly significant computation; rather it is reformulating the knowledge that the system already has about possible threats into the vocabulary of possible plans. More specifically, this rule says:

```
(def-brule recog-plan-1
  (recog-plan recog-move ?opp-player (world-at-time ?time)
     (plan ?move done) (goal-capture ?piece ?loc (move ?move)))
 <=
  (and (computed-possible-moves ?opp-player current-time ?ms)
       (in-set (move ?opp-player (capture ?piece)
                     ?opp-piece ?opp-loc ?loc)
               ?ms)
       (= ?move
          (move ?opp-player (capture ?piece)
                ?opp-piece ?opp-loc ?loc)) ))
```

To determine an opponent plan to capture a piece, get the set of detected threats and find such a threat to capture the piece

Figure 3.18: Recognizing plans to capture pieces

TO COMPUTE: A plan the opponent can execute

DETERMINE: The set of threats that the opponent can make
and an immediate capture that is in the set

Another, more complicated, method of recognizing threats is for the system to use its set of *offensive strategy* rules. These strategies, which will be discussed in detail in section 3.7.2, are scripts (a.k.a. schemata or macro-operators) which encode general plans that are known to be useful in achieving goals. The rule for recognizing opponent plans using offensive strategy scripts, which is not shown here but is given in appendix A, says roughly:

TO COMPUTE: A plan the opponent can execute

DETERMINE: An offensive strategy script
which the opponent can currently use

Each of these plan recognition methods reflects a particular type of opponent threat. There are, of course, other plans that the opponent may undertake that CASTLE should recognize. We will see how CASTLE can learn new plan recognition methods in chapter 6.

3.6 Goal generation

The next component CASTLE invokes is the *goal generation* component, which is described in figure 3.19. This component has the task of determining the goals that CASTLE should consider in its planning. These goals then drive CASTLE's plan generation methods. This intermediate step of goal representations, which is not taken in plan recognition, allows CASTLE to engage in more top-down reasoning in its own planning than it uses for recognizing opponent plans.

The goal recognition methods are applied by the meta-rule shown in figure 3.20, which says roughly:

Component name	Goal generation
Specification	The goal is worth pursuing and likely to be satisfyable
Methods	Capture piece, gain position, ...
Invocation	`(is-goal ?method ?player ?world ?goal)`
Subcomponents	Threat detection
Used by	Meta-rules

Figure 3.19: The goal generation component

IF The goal recognition component can find a goal
that is relevant in the current situation

THEN Assert that the goal is a currently-active goal

The simplest goal generation method is shown in figure 3.21. As discussed earlier, this rule simply transforms the captures that the system can make into the vocabulary of goals:

TO COMPUTE: A goal for a player

DETERMINE: The set of threats available to the player
and a direct capture that is in the set

This is analogous to the plan recognition rule shown in figure 3.18, which transformed knowledge of possible opponent threats against the computer into the vocabulary of recognized opponent plans.

A more complex goal detection rule, which is not shown here but is given in appendix A, detects pieces that are likely to be exposed to a large number of lines of attack. It determines this by measuring how many of its adjoining squares are empty. The rule says roughly:

TO COMPUTE: A goal for a player

DETERMINE: A piece of the opponent's
which looks exposed

```
(def-rule comp-goals-comp comp-goals                                  If I can generate
  (is-goal ?method computer (world-at-time current-time) ?goal)       a goal to pursue,
=>                                                                    record it as an
  ((active-goal computer ?goal current-time)))                        active goal
```

Figure 3.20: Invoking the goal recognition component

```
(def-brule is-goal-1
  (is-goal ?player (world-at-time ?time)
      (goal-capture ?opp-piece ?loc2 (move ?move)))
<=
  (and (computed-possible-moves ?player ?time ?move-set)
       (in-set (move ?player (capture ?opp-piece)
                     ?comp-piece ?loc ?loc2)
               ?move-set)
       (= ?move (move ?player (capture ?opp-piece)
                      ?comp-piece ?loc ?loc2))  ))
```

To generate a goal to pursue, retrieve the set of active threats against an opponent piece, and consider the goal of capturing it

Figure 3.21: Transforming a possible capture into a goal

Similar goal detection rules are used to find pieces which appear to have their movement constrained, as well as pieces that for other reasons appear to be worth consideration in planning.

In addition to the activation of goals to capture pieces, CASTLE also sets goals to counterplan against opponent threats. These are represented by the goals (goal-counterplan *opponent-goal*). The threats that CASTLE should counterplan against have already been recognized by the *plan recognition* component, and again merely have to be transformed into the vocabulary of goals. This is accomplished by the rule in figure 3.22, which basically spawns a counterplan goal for every possible opponent plan that has been recognized.

3.7 Plan generation

After generating the goals that it will pursue, CASTLE proceeds to generate plans to satisfy the goals. To minimize the computation that it has to perform, CASTLE starts with simple and efficient planning methods and then uses increasingly complex methods as necessary. This control of planning computation is accomplished through the system's *meta-planning* rules, which invoke CASTLE's variety of plan generation methods. The meta-planning rules

```
(def-brule is-goal-cp
  (is-goal computer (world-at-time ?time)
           (goal-counterplan ?plan))
<=
  (possible-opp-plan (world-at-time ?time) ?plan ?goal))
```

To generate a counterplanning goal to pursue, retrieve an opponent plan

Figure 3.22: Generating a goal to counterplan

Component name	Meta-planning
Specification	(forward-chaining)
Methods	Counterplanning, scripts, forward-search, ...
Assertion	`(possible-plan ?player ?time ?plan ?goal)`
Subcomponents	Counterplanning, scripts, forward-search, ...
Used by	(top-level)

Figure 3.23: The meta-planning component

are a subset of the system's meta-rules, which selectively invoke various planning components and assert propositions about the results into CASTLE's memory. (We distinguish meta-planning rules from general meta-rules because of their common structure and purpose; it should be kept in mind that there is no implementational difference.)

As is shown in figure 3.23, meta-planning rules assert `possible-plan` beliefs into the system's memory when they are able to construct a plan. Other meta-planning rules may then refrain from planning if a `possible-plan` belief has already been asserted, in order to avoid unnecessary computation. Again, in many cases the system will readily possess knowledge of actions that it can take represented in a form that can be transformed into the vocabulary of plans without significant computation. This occurs, for example, when the system is presented with an opportunity for a direct capture of an opponent piece. The meta-planning rule for transforming such a perceived opportunity into a plan is shown in figure 3.24, which says:

IF There is an active goal
and there is a move which will achieve it immediately

THEN Assert a possible plan to seize the opportunity

```
(def-rule comp-plan-move comp-plans
  (and (sorted-eval-gen (vars ?goal ?goal-eval ?a-goal)          If the active
       (and (active-goal computer ?a-goal current-time)          goal with the
            (eval-goal ?a-goal ?goal-eval) ))                    highest value
       (not (did-goal-planning current-time))                    before planning
       (opportunity computer (world-at-time current-time)        has an opportunity
                    ?goal ?plan))
=>                                                               Assert the
  ((possible-plan computer current-time ?plan ?goal)             opportunity
   (did-goal-planning current-time) ))                           as a possible plan
```

Figure 3.24: Deciding to grab an opportunity

Component name	Counterplanning
Specification	The plan resulted in the opponent's goal not being satisfied
Methods	Run away, counterattack, interposition, ...
Invocation	`(counterplan ?method ?player ?goal ?time ?plan)`
Subcomponents	Threat and opportunity detection
Used by	Meta-planning, plan generation

Figure 3.25: The counterplanning component

Another meta-planning rule of this sort concerns continuing a previously-active plan. This rule, given in appendix A, says roughly:

IF There was a plan from the previous turn
that has some steps still to be executed
and the next step in the plan is applicable now

THEN Assert as a possible plan the rest of the previous plan

The rest of CASTLE's methods for constructing plans involve more significant amounts of computation, which in most cases is performed by sub-components dedicated to the particular planning method. The planning methods that are currently implemented in CASTLE include:

- **Counterplanning:** Responding to enemy threats
- **Strategy script application:** General plans that accomplish particular goals
- **Option-limiting plan application:** Plans to limit opponent options
- **Forward search:** Game-tree search for ways to satisfy any goal

3.7.1 Counterplanning

One of CASTLE's planning components is dedicated to the task of counterplanning [Carbonell, 1979; Carbonell, 1981], that is, responding to the opponent's adversarial plans. As we discussed in chapter 2, the `counterplanning` component is used to generate responses to immediate threats. At the system's top-level decision-making, counterplanning is invoked by the meta-planning rule shown in figure 3.26, which says roughly:

IF The opponent has an active goal
and I can generate a counterplan against it

THEN Assert the counterplan as a possible plan

```
(def-rule comp-counterplan comp-plans                                  If there's an active
 (and (active-goal opponent ?opp-goal current-time)                    opponent goal, and
      (counterplan computer ?opp-goal current-time ?counterplan))      I have a counterplan,
=>                                                                     Then assert the plan
 ((possible-plan computer current-time ?counterplan                    as possible to execute
                 (goal-counterplan ?opp-goal))))
```

Figure 3.26: Deciding to respond to a threat (counterplan)

The two simplest counterplanning rules, which we discussed in chapter 2, are to counterplan by running away and by counterattacking. The first of these is shown in figure 3.27, which says roughly:

TO COMPUTE: A counterplan to a capture

DETERMINE: A move from the targeted square
to a square which cannot be attacked

The second, for counterattacking against the attacking piece, was shown in figure 3.3. It says roughly:

TO COMPUTE: A counterplan to a capture

DETERMINE: A piece on the board that can capture the attacker

A third way that CASTLE can counterplan is to use the method learned in chapter 2 for counterplanning by *interposition*. This rule, which is shown in figure 2.8 and is given in appendix A, says roughly:

```
(def-brule counterplan-run
  (counterplan cp-run ?player                                          To counterplan
    (goal-capture ?piece ?loc                                          against an opponent
     (move ?opponent (capture ?piece) ?opp-piece ?opp-loc ?loc))       attack against a
     ?time ?the-response)                                              player's piece
<=                                                                     Find a place the
 (and (move-legal (move ?player ?move ?piece ?loc ?new-loc))           attacked piece can
      (no (and (at-loc ?opponent ?other-opp-piece                      move to that no
                       ?other-opp-loc ?time)                           opponent piece can
               (move-legal                                             attack
                 (move ?opponent (capture ?piece)
                       ?other-opp-piece ?other-opp-loc ?new-loc))))
      (= ?the-response                                                 and return the plan
         (plan (move ?player move ?piece ?loc ?new-loc) done))))       to make the move
```

Figure 3.27: Reacting to a threat by running away

```
(def-brule counterplan-buy-time
 (counterplan cp-buy-time ?player                                  To counterplan
    (goal-capture ?target-piece ?target-loc                        against an opponent
         (move (move ?opp-player (capture ?target-piece)           attack against a
                     ?opp-piece ?opp-loc ?target-loc)))            player's piece
    ?time ?cp)
<=                                                                 (Unless the attacked
 (and (not (= ?target-piece king))                                 piece is a king)
      (= ?cp (plan (move ?player move ?cp-piece ?cp-loc ?interm-loc)
                   (next (move ?player (capture ?opp-piece)        First move to an
                               ?cp-piece ?interm-loc ?opp-loc)     intermediate square
                         done)))
      (at-loc ?opp-player king ?king-loc ?time)                    which threatens the
      (move-legal-in-world (move ?player (capture king) ?cp-piece
                                 ?interm-loc ?king-loc)           opponent's king
                           (world-at-time ?time))
      (move-legal-in-world                                         which from there
        (move ?player (capture ?opp-piece) ?cp-piece               can capture the
              ?interm-loc ?opp-loc)                                opponent piece
        (world-at-time ?time))                                     that is making the
      (move-doable (move ?player ?move-type                        original attack
                         ?cp-piece ?cp-loc ?interm-loc)
                   (world-at-time ?time))
      (no (and (at-loc ?opp-player ?other-piece ?other-loc ?time)
               (move-legal-in-world                                As long as no
                  (move ?opp-player (capture ?cp-piece)            opponent piece
                        ?other-piece ?other-loc ?interm-loc)       threatens the
                  (world-at-time ?time)))) ))                      intermediate square
```

Figure 3.28: Reacting to a threat by buying time

TO COMPUTE: A counterplan to an attack

DETERMINE: A location on the line of attack
that another piece can move to

A more complex way to respond to a threat is to *buy time* by putting the opponent into check in such a way that you will later be able to counterplan against the threat. This is achieved by the rule in figure 3.28, which says roughly:[4]

TO COMPUTE: A counterplan to a capture

DETERMINE: A location from which a piece can attack the king
and a piece on the board that can move to that location
such that the moved piece can counter the threat
from its new location

[4]The rule as shown in the figure is specialized for counterplanning by counterattacking.

Note that this rule is quite specific, in that the opponent must be put in check by the same piece that will then disable the threat. Another possibility would be to open up a discovered attack putting the opponent in check, such that the moved piece (i.e., the piece that opened up the line of attack on the king) could then disable the attack. We will see in chapter 7 how CASTLE can learn this counterplanning strategy.

Invocation by other components

In addition to being invoked by the system's meta-rules, the `counterplanning` component is also invoked by other components which need to reason about counterplans. One example of this is the method of planning by two-ply search which we discussed in chapter 2 and which will be described in more detail later in this section. This method invokes the counterplanning component to determine whether the opponent will be able to respond to a threat being considered for execution by the system. The system's offensive strategy scripts similarly make use of the specialized knowledge in the counterplanning component. In both of these cases, using the counterplanning component makes the planning process more efficient, because it allows the planning methods to straightforwardly use rules that encode knowledge of responding to threats, instead of forcing them to consider all the moves that the opponent might make. It additionally provides new opportunities for CASTLE to learn new counterplanning methods, because a failure of the counterplanning component in any of these contexts may let CASTLE learn a new rule that can be used whenever the counterplanning component is invoked.

3.7.2 Forced-outcome strategy scripts

The next planning method which CASTLE employs is *script application*. This component, described in figure 3.29, has knowledge of a set of *offensive strategies* that are commonly able to satisfy goals, and the component attempts to apply these strategies to the active goals and current situation.

CASTLE has two meta-planning rules that invoke the strategy component. One of them, shown in figure 3.30, applies the system's scripts to its active goals:

Component name	Offensive strategies (scripts)
Specification	The plan is a sequence of steps which will achieve an offensive goal
Methods	Fork, pin, simultaneous attacks, ...
Invocation	`(strategy ?method ?player ?world ?goal ?plan)`
Subcomponents	(None yet)
Used by	Plan generation

Figure 3.29: The offensive strategy component

```
(def-rule comp-strategy comp-plans                    If no plan has been
  (and (not (possible-plan computer current-time ?p ?g))   suggested, and the
       (sorted-eval-gen (vars ?goal ?goal-eval ?a-goal)    highest-valued
           (and (active-goal computer ?a-goal current-time)  active goal
                (eval-goal ?a-goal ?goal-eval) ))          such that a
       (in-set ?strategy-meth strategy-meths)              strategy method
       (strategy strategy-meth computer                    returns a plan
                 (world-at-time current-time) ?goal ?plan)  and the plan's
       (seq-1st ?plan ?move)                               first move is
       (move-doable ?move (world-at-time current-time)) )  currently feasible
=>                                                         Assert the plan as
  ((possible-plan computer current-time ?plan ?goal) ))    a possible plan
```

Figure 3.30: Applying offensive strategies: goal-directed

IF There is no plan suggested yet
and the highest-valued active goal
has an offensive strategy that can satisfy it
which can be executed in the current situation

THEN Assert the plan as a possible plan

The second meta-rule, which is given in appendix A, applies the strategies in an undirected fashion, looking for any strategy which applies in the current situation:

IF There is no possible plan asserted yet
and an offensive strategy yields a plan

THEN Assert the possible plan

These meta-rules are part of CASTLE's general attempt to minimize the amount of computation that is performed in planning. Script application is in general the most efficient non-trivial approach to planning, because it avoids projection by compiling it into predictive tests that can be immediately checked. Script selection is most efficient given constraints from the system's active goals, so CASTLE checks its scripts for applicability in a goal-directed fashion before trying them in an undirected fashion. Additionally, CASTLE only attempts script application if previous planning methods have not succeeded in generating a plan.[5]

One offensive strategy rule is shown in figure 3.31, which encodes the strategy of *forking*, in which a piece is moved to a location from which it can capture either of two enemy pieces. This rule says basically:

[5]One extension of CASTLE's undirected use of strategy scripts would be to apply them in a time-bounded fashion, even if a possible plan had already been suggested. If the rules that make up CASTLE's set of scripts were able to bound their computation when the meta-rules so indicated, they could then be used to spot unexpected opportunities even when other less-valuable opportunities had already been found.

```
(BRULE learned-strategy-method8574
(strategy learned-strategy-method8574 ?player (world-at-time ?time)          A forced-outcome
          (goal-capture ?target-piece (loc ?r2 ?c2) ?info)                  strategy for
    (plan (move ?player move ?en-piece (loc ?r ?c) (loc ?r3 ?c3))           capturing a piece
          (next (move ?player (capture ?target-piece) ?en-piece             using two moves
                      (loc ?r3 ?c3) (loc ?r2 ?c2)) done)))
<=
 (and (= ?opp (player-opponent ?player)) (= (current-game) chess)
     (at-loc ?player ?en-piece (loc ?r ?c) ?time)                           Find a piece
     (move-legal-in-world                                                   that can move
        (move ?player move ?en-piece (loc ?r ?c) (loc ?r3 ?c3))
        (world-at-time ?time))
     (not (at-loc ?other-player ?other-piece (loc ?r3 ?c3) ?time))          to an empty square
     (at-loc ?opp ?target-piece (loc ?r2 ?c2) ?time)                        and a piece of the
     (move-legal-in-world (move ?player (capture ?target-piece) ?en-piece   opponent's that
                                (loc ?r3 ?c3) (loc ?r2 ?c2))                can be attacked
        (world-at-time ?time))                                              by the moved piece
     (at-loc ?opp ?other-target (loc ?r4 ?c4) ?time)                        and another piece
     (move-legal-in-world (move ?player (capture ?other-target) ?en-piece   of the opponent's
                                (loc ?r3 ?c3) (loc ?r4 ?c4))                that can be taken
        (world-at-time ?time))                                              by the same piece
     (<= (piece-value ?target-piece) (piece-value ?other-target))           where the first
     (not (= (loc ?r4 ?c4) (loc ?r2 ?c2)))                                  target is worth
     (no (and (counterplan ?cp-meth ?opp                                    less than the 2nd
              (goal-capture ?target-piece (loc ?r2 ?c2)                     such that there is
                 (move (move ?player (capture ?target-piece) ?en-piece      no counterplan
                             (loc ?r3 ?c3) (loc ?r2 ?c2))))                 for the opponent
              ?time ?cp)                                                    that handles both
            (counterplan ?cp-meth2 ?opp                                     of the attacks
              (goal-capture ?other-target (loc ?r4 ?c4)
                 (move (move ?player (capture ?other-target) ?en-piece
                             (loc ?r3 ?c3) (loc ?r4 ?c4))))
              ?time ?cp)))))
```

Figure 3.31: Learned strategy rule for the *fork*

TO COMPUTE: A strategy for capturing a piece

DETERMINE: An piece that can be moved to an empty intermediate location
and an opponent piece that can be attacked
by the moved piece at the intermediate location
and another opponent piece that can be attacked
by the moved piece at the intermediate location
such that there is no single counterplan to both attacks

Suppose this rule is invoked in the situation shown in figure 3.32. CASTLE will attempt to apply the rule for each of the pieces it has on the board. For the portion of the board which is shown, it will first iterate over the locations to which the knight is able to move. For each location it will then see if it can attack any pair of opponent pieces. From one of these locations, to which it is shown moving in the figure, it can in fact attack both the opponent's bishop and his rook. The rule next checks if any of the counterplans against the bishop attack (e.g., moving the bishop, interposing a piece) will also counterplan against the rook attack. This not being the case, the rule will succeed with the plan to move the knight,

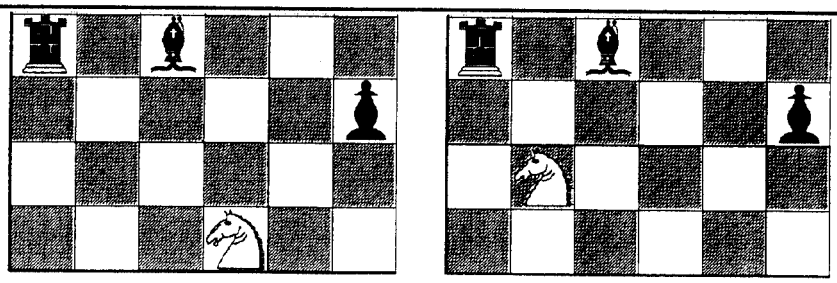


Figure 3.32: Using the *fork* strategy: White move

and CASTLE expects that the bishop attack will be carried out.

Another strategy script rules in CASTLE is for simultaneous attacks. We will discuss CASTLE's strategies more in chapter 6, when we see examples of CASTLE learning rules for the fork and other strategies.

In addition to being used for plan generation, CASTLE's offensive strategies are also used for *plan recognition,* as we touched on in section 3.5. (The meta-rule which uses the strategy rules for plan recognition is given in appendix A.) This use of the strategy rules is not fundamentally different from their use in planning, other than the opponent being the player in question rather than the computer. It does, however, provide an avenue for learning scripts from the opponent, as we will see in chapter 6.

3.7.3 Forward-directed search

If CASTLE cannot generate a plan using any of the methods we have discussed so far, it will attempt to use a straightforward but computationally expensive search technique. This approach, which we saw in chapter 2, is to search for a two-move plan that is guaranteed to capture a piece from the opponent. Basically CASTLE searches for a move that it can make that will put it in position to make a successful attack on the next turn.

This approach can be used in either a goal-directed or an undirected fashion. If goal-directed, the idea is to search for a piece to move and a location to move it to such that the piece will be able to carry out the goal (e.g., take the target piece) from the new location. If undirected, this must be done for all possible goals. These two possibilities can be handled by a single planning rule, shown in figure 3.33, which says roughly:

TO COMPUTE: A plan to capture an opponent's piece

DETERMINE: A piece which can move to a new location
and an opponent piece
which can be captured by the moved piece
from the new location
such that the opponent has no defense

If this rule is invoked with a specified piece to capture, the search will be constrained to find plans for capturing the specified piece. If it is not, CASTLE's inference engine will handle the

```
(def-brule plan-forward-2
  (plan-forward ?player ?time ?plan                                  To construct a plan
    (goal-capture ?target-piece ?target-loc (move ?move))            to capture a piece
    ?min-val)                                                        of a certain value
 <=
  (and (= ?plan (plan (move ?player move ?attack-piece
                        ?attack-loc (loc ?interm-r ?interm-c))
                      (next ?move done)))
       (move-doable (move ?player move ?attack-piece                 Find a move that
                      ?attack-loc (loc ?interm-r ?interm-c))         is currently
                    (world-at-time ?time))                           feasible
       (at-loc ?opponent ?target-piece ?target-loc ?time)            and a target piece
       (= ?move
          (move ?player (capture ?target-piece) ?attack-piece        that the moved
                (loc ?interm-r ?interm-c) ?target-loc))              piece is able to
       (move-legal-in-world ?move (world-at-time ?time))             attack
       (eval-goal (goal-capture ?target-piece ?target-loc            whose capture is
                                (move ?move)) ?val)                  more valuable than
       (> ?val ?min-val)                                             the minimum
       (no (counterplan ?cp-meth ?opponent
             (goal-capture ?target-piece ?target-loc                 such that the
                           (move ?move))                             opponent has no
             ?time ?counter-plan)) ))                                counterplan
```

Figure 3.33: Planning a two-move sequence

at-loc query by searching through the pieces that can be attacked. The first of these uses of the planning rule is handled by the meta-rule shown in figure 3.34, which says roughly:

```
(def-rule comp-forward-plan comp-plans                               If there is not
  (and (not (possible-plan ?computer current-time ?p ?g))            a plan suggested
       (sorted-eval-gen (vars ?goal ?goal-eval ?a-goal)              then retrieve the
           (and (active-goal computer ?a-goal current-time)          highest-valued
                (eval-goal ?a-goal ?goal-eval)))                     active goals
       (plan-forward computer current-time ?plan ?goal 0) )          and plan for them
=>                                                                   and assert results
  ((possible-plan computer current-time ?plan ?goal) ))              as possible plans
```

Figure 3.34: Brute-force goal-directed planning

IF There is not yet a possible plan
and the best active goal
has a plan found by brute-force search

THEN Activate the plan

The second, undirected application of the planning rule, is handled by a similar meta-rule, given in appendix A, which invokes the planner without a specified-goal whenever there is no plan for any of the active goals.

This is the last of CASTLE's planning methods. If none of these can produce a viable plan, CASTLE's meta-rules for move selection will generate a randomly-selected move to make.

3.8 Plan selection

Once CASTLE has generated plans for a number of active goals, the task at hand is to select the plan which should be executed. CASTLE performs this task using the *plan selection* meta-rule shown in figure 3.35. This rule says roughly:

IF The computer is not in check
and there is a possible plan that satisfies a goal
of higher value than the goals of all other possible plans
and whose first move is currently feasible

THEN Activate the plan for execution
and activate the first move in the plan as the current move

This rule chooses the plan which satisfies the highest-valued goal. The goals are evaluated using the *goal evaluation* component which is described in figure 3.36. Two simple goal evaluation rules are shown in figure 3.37. The first one says that the value of satisfying a

```
(def-rule comp-move comp-move                          ; If the computer
  (and (not (in-check computer current-time ?move))    ; is not in check
       (sorted-eval-gen (vars ?plan ?goal-val ?a-plan) ; choose the highest
         (and (possible-plan computer current-time     ; valued active goal
                             ?a-plan ?a-goal)
              (eval-goal ?a-goal ?goal-val) ))         ; for which there is
       (possible-plan computer current-time ?plan ?goal) ; a possible plan
       (= ?plan (plan ?move ?plan-rest))               ; such that the first
       (move-doable ?move (world-at-time current-time)) ) ; move is feasible,
=>                                                     ; Then activate the
  ((active-plan computer current-time ?plan ?goal)     ; plan and make the
   (move-to-make ?move computer ?goal current-time) ))  ; first move
```

Figure 3.35: Selecting the best move

Component name	Goal evaluation
Specification	The specified goal has the given value
Methods	Piece value, ...
Invocation	`(eval-goal ?method ?goal ?value)`
Subcomponents	(None)
Used by	Plan selection

Figure 3.36: The goal evaluation component

goal to capture a piece is the value of the piece captured. The second says that the value of satisfying a goal to counterplan against an enemy attack is the value of the piece saved.

Another approach to plan selection

It is clear that the scheme for plan/goal evaluation discussed here is inadequate for complex domains, due to the difficulty of assigning a numerical value to the benefits and detriments of a plan. A better approach, not yet implemented in CASTLE, would be to make use of comparisons resulting in a partial order in the space of plans and goals. This would reflect the inadequacy of assigning numerical values to plan/goal evaluations, and would allow some comparisons to be undefined. Additionally, it would provide a new realm of learning for CASTLE, by allowing CASTLE to learn new plan/goal comparison methods whenever an expectation failure reflected an incorrect evaluation of the system's or opponent's plans.

Examining such a scheme more specifically, the determination of preference between plans could be determined using a set of *plan preferences* which each encode a particular reason for a plan and the goal it serves being more valuable than another plan and goal. The plan selector would consider a plan to be more valuable than another plan if there are more reasons for preferring it over the other plan than vice versa. In other words, each plan

```
(def-brule eval-goal-1                                    ; To evaluate a goal
  (eval-goal (goal-capture ?piece ?loc ?info) ?value)     ; to capture a piece
<=                                                        ; Retrieve the value
  (= ?value (piece-value ?piece)) )                       ; of the piece

(def-brule eval-goal-2
  (eval-goal (goal-counterplan                            ; To evaluate a goal
               (goal-capture ?piece ?loc ?info)) ?value)  ; to defend a piece
<=                                                        ; Retrieve the value
  (= ?value (piece-value ?piece)) )                       ; of the piece
```

Figure 3.37: Two goal evaluation rules

Component name	Expectation generation
Specification	The expectation is worth monitoring
Methods	Next plan step, best opponent move, ...
Invocation	`(should-expect ?method ?time ?expectation)`
Subcomponents	(None)
Used by	Meta-planning

Figure 3.38: The expectation generation component

preference method would enter a vote for each pair of plans, and the plan that has more votes in its favor when compared to all other possible plans is the winner.

What would these plan comparison metrics look like? The most obvious one is to prefer plans for capturing higher-valued pieces. Another is to prefer plans that are guaranteed to succeed over those that are not. If one of the plans involved a large risk while the other had no risk involved, the low-risk plan might be preferred. Alternatively, if one of the plans achieved a large short-term gain but left the system in a poor position, while the other involved a low short-term gain but a better ending position, this too should be taken into account by a plan preference. These rules could also be modulated by the global situation, and could thus prefer high-risk plans if the system would lose without them, or could prefer short-term gain when the game is close to ending.

There are shortcomings to this scheme as well, since obviously some reasons for preference should carry more weight than others. One alternative might be to associate numerical weights with each comparison metric, which could be computed dynamically in terms of the situation of the game. Ultimately what is necessary is an explicitly encoded model of plan selection that includes both qualitative and quantitative factors.

3.9 Expectation generation

The *expectation generation* component is used to make assertions about events that CASTLE expects to occur in the future. These expectations for the most part stem from assumptions that CASTLE made during plan generation, because it is these predictions that need to be monitored for accuracy.

The expectation generation component is described in figure 3.38. It is invoked using a `should-expect` query, as is seen in the meta-rule shown in figure 3.39. This meta-rule performs the relatively simple computation:

IF An expectation generation method can determine an expectation

THEN Store it as an expectation to monitor

As in the other components we have seen, the actual computation involved in generating expectations is carried out by the component method rules. One such rule, shown in

```
(def-rule expect-expectations comp-expectations
  (should-expect ?exp-meth current-time ?exp)
=>
  ((expect ?exp)))
```

If an expectation method says to expect something Then expect it

Figure 3.39: Meta-rule for expectation generation

figure 3.40, says to expect the next move in the computer's plan to be executed on the next turn:

TO COMPUTE: An expectation to monitor

DETERMINE: The system's currently active plan
and the next move in that plan
and expect it to be executed next turn

CASTLE has a number of other expectation generation rules, which are given in appendix A. One of these rules says to expect the opponent to execute the best capture or defense on the next turn. Another, which we will discuss in section 6.4, says to expect that any threats against computer pieces will be countered, unless the opponent has an offensive strategy in place for achieving the threat. Yet another expectation generation rule says to expect pieces that are mobile to retain their mobility, unless the opponent has a method of limiting the computer's options. Figure 3.41 summaries the types of expectations that CASTLE maintains. These expectations are monitored over time to determine their success or failure. Their key use is in learning: A failure of one of them indicates a fault in CASTLE's inference procedures which must be repaired [Sussman, 1974; Schank, 1982; Hayes-Roth, 1982; Hammond, 1989]. These expectations can also be useful in other ways. For example, an agent could monitor expectations which indicate critical points in plan execution [Doyle *et al.*, 1986]. Another use is in perception, in which a system can increase efficiency by limiting

```
(def-brule exp-plan-exec
  (should-expect exp-plan-exec ?time ?exp)
<=
  (and (active-plan computer ?time ?plan ?goal)
       (= ?plan (plan ?cur-move ?plan-rest))
       (= ?plan-rest (next ?move ?rest-next))
       (= ?exp
          (move-to-make ?move computer ?goal (+ 2 ?time)))) )
```

I should expect a plan to be executed if I have an active plan which will be in the midst of execution next turn

Figure 3.40: Expecting the next plan step to be executed

Expectation	Meaning
(move-to-make *move player goal time*)	The player's move at the given time will be the specified move
(no-threats *player time*)	The player will not be able to make a capture at the given time
(no-opps-at-time *player time*)	The player will not have any opportunities for capture at the given time
(mobile-pieces *player time pieces*)	The player's given set pieces will all be mobile

Figure 3.41: Types of expectations

computations to those which will be useful in confirming or denying the agent's expectations [Birnbaum *et al.*, 1993].

Note that there is nothing inherent in CASTLE's architecture that forces these expectations to be taken into account by the planner, and in fact the expectations themselves are not necessarily generated as a result of planner assumptions. All that the architecture specifies is that rules will exist to generate and monitor the expectations. Several of the expectation generation rules, of course, are in fact triggered as a result of planner output. This is true of the `move-to-make` expectations about plan continuations, and is also true of `mobile-pieces` expectations that follow from recognized opponent plans to limit options. While most of CASTLE's expectations are not themselves used in future planning, this could certainly be implemented if more complex planning techniques were implemented in the system's decision-making architecture. If, for instance, the system incorporated a STRIPS or NONLIN-like planner, expectations could be used as initial conditions that are assumed to be true.

3.10 Execution and expectation monitoring

The last task which CASTLE performs is plan execution. In our domain of chess, of course, there is little involved in executing the plans which our planner has selected. The plan selection meta-rules shown in section 3.8 assert beliefs about the currently active plan, and additionally determine the first move to be made in the plan and assert it. CASTLE's control program then queries memory for a `move-to-make` expression, and proceeds to update the board and go on to the next turn.

One further task which CASTLE must undertake during plan execution is the monitoring of expectations. We saw in section 3.9 that CASTLE maintains a set of expectations that reflect the underlying assumptions that were made in planning. During execution, CASTLE's *expectation monitoring component,* described in figure 3.42, is invoked to determine the success and failure of its active expectations.

Component name	Expectation monitoring
Specification	The expectation has failed
Methods	Unexpected move, ...
Invocation	`(exp-failure ?exp ?time)`
Subcomponents	(None)
Used by	Meta-planning

Figure 3.42: The expectation monitoring component

One such expectation monitoring rule is shown in figure 3.43. This rule monitors expectations about moves that will be made at particular times, either by the computer or by the opponent. The rule says roughly:

TO COMPUTE: The failure of an expectation about a move to make

DETERMINE: that the expectation hasn't already failed
and whether the move that was made at the given time
was different from the expected move

The expectation monitoring rules are applied by the meta-rule shown in figure 3.44. When this rule determines that an expectation has failed, it asserts this fact in memory, in the form of an `expectation-failed` assertion, and invokes CASTLE's diagnosis and learning components by further making a `handle-expectation-failure` assertion. This latter assertion triggers CASTLE's diagnosis engine to determine the fault underlying the failed expectation, and the rule construction engine to repair the system so that failures of this type will not occur again. We will discuss these two processes in chapters 4 and 5.

A final point which is worth making at this time concerns the need for additional rules in order to notice expectation failures, rather than just directly monitoring the expected expressions themselves. Quite simply, this is necessary because the detection of an expectation failure often requires inference that is sensitive to the semantics of the particular

```
(def-brule exp-fail-move
  (exp-failure (expect (move-to-make ?move ?player                  To detect a failure
                                     ?goal ?time)                   of an expectation
                       ?exp) ?time)                                 of a player's move
<=                                                                  (if it hasn't
  (and (not (expectation-failed ?exp ?other-time))                  failed yet) compare
       (move-to-make ?other-move ?player ?other-goal ?time)         the move made to
       (not (= ?other-move ?move))))                                the expected one
```

Figure 3.43: Detecting an unexpected move

```
(def-rule monitor-exp-failure comp-expectations                    If I can detect
  (exp-failure ?exp current-time)                                  a failure
=>
  ((expectation-failed ?exp current-time)                          record it and
   (handle-expectation-failure ?exp current-time)))                learn from it
```

Figure 3.44: Detecting expectation failures

expectations. In our example, when there was an expectation that a particular move would be made at a particular time, a failure is noticed when a different move is made at the same time, but not when a different move is made at a different time. If the expectation was about the active goals, on the other hand, the activation of one goal might *not* in fact signal the failure of an expectation regarding another active goal. The monitoring rules thus constitute knowledge of the meaning of the expectations.

3.11 The lesson of CASTLE's architecture

This chapter has presented the decision-making model embodied in CASTLE. The flow of control is managed by meta-rules whose primary function is to selectively invoke various components and store their results in memory. The actual work of carrying out the various tasks is done by the components, which operate using sets of backward-chaining rules, each of which represents a particular way to carry out the task.

This chapter has attempted to serve two purposes. The first is to acquaint the reader with the decision-making model implemented in CASTLE, which will be the basis for discussion in the rest of the thesis. More important than the particular model itself, however, is CASTLE's

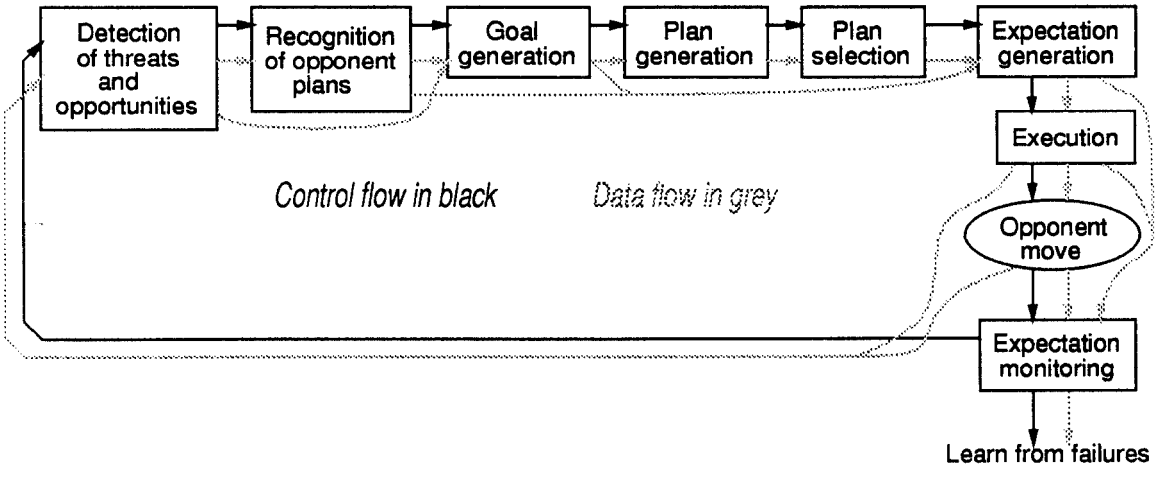


Figure 3.45: CASTLE's decision-making process

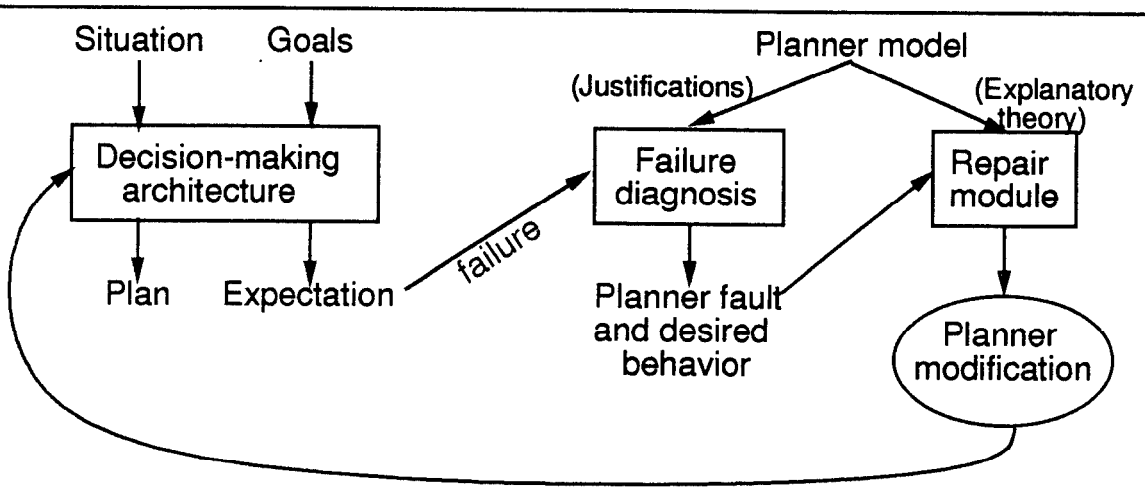


Figure 3.46: The learning process in CASTLE

approach to modeling decision-making processes. We will see in subsequent chapters that structuring a planner into components facilitates a powerful approach to learning from failures. This approach learns new methods for any of the tasks in the decision-making model, using the explicitly-represented breakdown of the tasks into components to isolate the component to be repaired and to formulate the new method. In particular, we will see that the specification of a purpose for each component facilitates learning of new methods by reasoning about the desired performance of the component that was not achieved.

It should be clear that many of the details of CASTLE's decision-making model are not themselves innovative. There are many other ideas that could be incorporated into CASTLE's model, such as complex planning methods (e.g., [Sacerdoti, 1975; Tate, 1977]), constructs for representing planning knowledge [Kuokka, 1990], and methods that incorporate execution-time inference [Firby, 1989; Simmons, 1991]. Nonetheless, the model will serve to demonstrate the utility of explicitly structuring a decision-model in components.

Chapter 4

Diagnosing expectation failures

In the course of execution, CASTLE uses the decision-making procedures described in chapter 3 to determine what actions it should take. The success of these actions, and the correctness of the assumptions on which they rely, are checked by the posting and monitoring of explicit *expectations*. The failure of one of these expectations indicates that CASTLE has made an incorrect decision, and thus signals an opportunity for the system to learn.

In principle, an expectation failure can be caused by a fault in any of the components or individual rules that were discussed in chapter 3, or by an incorrect belief in CASTLE's memory. The first step in CASTLE's learning process is to pinpoint the specific fault [Hammond, 1986b; Simmons, 1988a]. The process of searching for the fault underlying an expectation failure is *failure diagnosis,* and is handled by a module of CASTLE called the *diagnosis engine.* The diagnosis engine returns a representation of a fault in the planner, which is then passed to the *repair module* to be fixed.

CASTLE's diagnosis algorithm is fairly straightforward and does not in and of itself add to previous research in diagnosis. The bulk of this chapter, therefore, is devoted to discussing issues that arise in diagnosing failures in a complex decision-making architecture such as CASTLE's. Many of these issues only arise in diagnosing structures with the properties of a planner, such as state, quantification, and limited information. This chapter presents CASTLE's solutions to these issues, and closes with a discussion of alternate approaches.

Chapter outline

This chapter will discuss the following:

1. The knowledge structures that are used in diagnosis
2. The basic algorithm for diagnosis
3. How plan-time beliefs are retrospectively tested during diagnosis
4. Types of planner faults that can be returned by the diagnosis engine
5. Limitations of the diagnosis algorithm

The process of repairing faults will then be discussed in chapter 5.

4.1 Diagnosing the planner

As we have said, the goal of CASTLE's diagnosis process is to find a fault in the system's decision-making procedures. This is in contrast to much of the past research on learning in planning, most of which has had the goal of finding and repairing faults in the *plans* that the agent was executing (see, e.g., [Simmons, 1988a; Hammond, 1989; Chien, 1990]). Chapter 1 discussed the following benefits of debugging the planner rather than specific plans:

- Repairing the planner allows for a wider applicability of the repair, in situations other than that of the original plan.[1]
- Repairing the planner will allow the system to learn decision-making rules that affect plans indirectly (e.g., perception rules) or which manipulate the plans themselves (e.g., scheduling rules).
- Repairing the planner is well-suited to agents made up of collections of specialized execution-time components [Collins *et al.*, 1991b].

Diagnosing a failure as a fault in the planner requires that the agent have more information than is needed to diagnose a failure as a fault in its plan. Approaches to accomplishing the latter make use of knowledge of the causal relations between the steps in the plan and the desired outcomes of those steps [Simmons, 1988a; Chien, 1990]. This causal information overlaps with the information that is often used in constructing plans. In other words, most planners maintain information about the causal relations between steps in the plan being constructed and the desired states being achieved. This information is used to determine whether the plan being constructed will actually achieve its goals, and to detect interactions between steps [Sacerdoti, 1975; Wilkins, 1984]. When a plan failure occurs, this same information can be used to see what step in the plan resulted in the failure.

For CASTLE to diagnose failures in terms of faulty planner components, analogous information is needed concerning the relations between the expectations being monitored and the components of the planner that led to their being predicted. This causal information enables the system to relate faulty expectations back to the decision-making components that may be implicated in the failure.

CASTLE maintains this new type of causal information in the form of *justifications* [deKleer *et al.*, 1977; Doyle, 1979; Smith *et al.*, 1985]. Loosely speaking, justifications represent the reasons that CASTLE believes each of its beliefs to be true. Some of these reasons may be things that CASTLE explicitly considered when inferring the belief. For example, when a person makes a decision, such as whether to drive to work, he may explicitly consider certain factors that go into the decision, such as how much traffic he expects and how much gas he has in the car. In addition, he will make a great number of implicit assumptions, including assumptions about the world such as the belief that the car will move forward when he presses the gas pedal, or the belief that it will go in the direction he

[1] Reasoning by analogy from repaired plans is another way to widen the applicability of learned plan knowledge [Hammond, 1989].

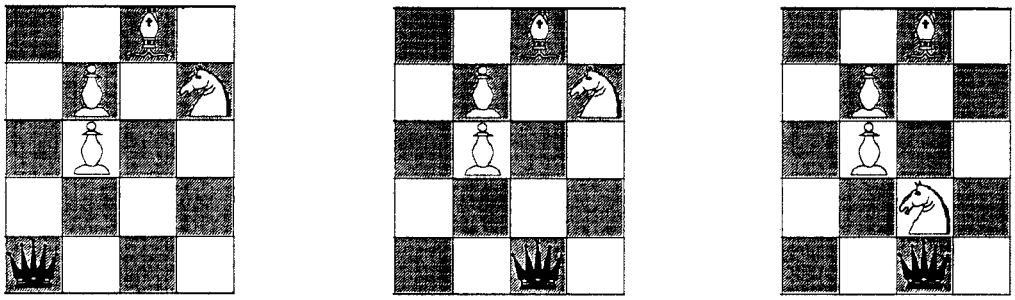


Figure 4.1: *Interposition* example: Computer (black) to move

points the steering wheel, and also beliefs about his own abilities such as the belief that he is capable of driving, or that his ability to predict traffic is reliable. CASTLE records the latter type of assumption along with the former, by making otherwise-implicit assumptions explicit and recording them with the decision rules that take them for granted.

4.2 Building justification structures

Let's now look at how CASTLE implements justifications. Every item in CASTLE's memory—beliefs, rules, and expectations—has an associated *justification structure* that encodes the system's reason for believing the item to be true. In the case of beliefs and expectations, these justifications will largely be made up of a record of the computation that led to the assertion of those beliefs or expectations into CASTLE's memory. In the case of rules, the justifications give a chain of inferences that led the system to believe that the antecedents of the rule in fact imply the consequent. These may be inferences that CASTLE used to learn the rule, or may have been supplied by the planner designer.

The most common situation in which justification structures are created is when CASTLE's inference engine concludes that a belief is true. This will occur whenever a rule is successfully applied. In the case of a forward-chaining rule, the assertion which is made into the system's memory is tagged with a justification which records the computation which led to its assertion. In the case of a backward-chaining rule, the belief that the query is true (given the variable bindings returned by the rule) is tagged with a justification structure. In both of these cases, the justification structure will primarily refer to the system's belief in the antecedents of the rule that made the inference.

Let's examine the justification structures that are built during the *interposition* example. In that example, we saw the sequence of events shown in figure 4.1, in which CASTLE expected its attack on the opponent's bishop to succeed. Instead, however, the opponent moved its knight in a manner that rendered the capture impossible. Given the failure of its expectation that the attack would succeed, CASTLE must determine why it made the mistake. In particular, it must realize that its counterplanning component is lacking a rule

```
(def-rule expect-expectations comp-expectations          If an expectation
  (should-expect ?exp-meth current-time ?exp)            method says to
=>                                                       expect something
  ((expect ?exp)))                                       Then expect it
```

Figure 4.2: Meta-rule for expectation generation

for responding to threats by interposing pieces.

To make this diagnosis, CASTLE examines its justification for belief in the truth of the expectation. This expectation was asserted by the meta-rule for expectation generation, shown in figure 4.2, which performs the simple computation: *If an expectation generation method says to expect something, activate it as an expectation.* This meta-rule in turn invoked the expectation generation method shown in figure 4.3, which says roughly: *retrieve the currently active plan and see if it has a second step, and if so, expect that I will execute it on the next turn.*

When these rules are invoked, CASTLE will first query its inference engine to determine the truth of the antecedents of the expectation generation method. This involves checking if it has an active plan in memory with more than one step, and constructing an expectation that the second move will be made. If such an expectation can be constructed, CASTLE will believe a should-expect proposition corresponding to the method's invocation (i.e., the rule's consequent). This will in turn cause the meta-rule to fire successfully, and the expect belief to be asserted into CASTLE's memory. For each of these steps in the inference chain, CASTLE will construct part of a justification structure for the asserted expectation. This justification is shown pictorially in figure 4.4 in the form of a digital circuit. The statements in the figure correspond to statements that CASTLE believes are true, and the "gates" that connect them correspond to the inferences that led the system to believe them to be true. **And** gates in justification structures correspond to inference rules whose antecedents are conjunctive. Since these rules apply only when all the conjuncts in their antecedents are true, the justification for the inferred belief is that it is considered true because all of its

```
(def-brule exp-plan-exec                                 I should expect
  (should-expect exp-plan-exec ?time ?exp)               a plan to be
<=                                                       executed
  (and (active-plan computer ?time ?plan ?goal)          if I have an active
       (= ?plan (plan ?cur-move ?plan-rest))             plan which will be
       (= ?plan-rest (next ?move ?rest-next))            in the midst of
       (= ?exp                                           execution next turn
          (move-to-make ?move computer ?goal (+ 2 ?time)))) )
```

Figure 4.3: Expecting the next plan step to be executed

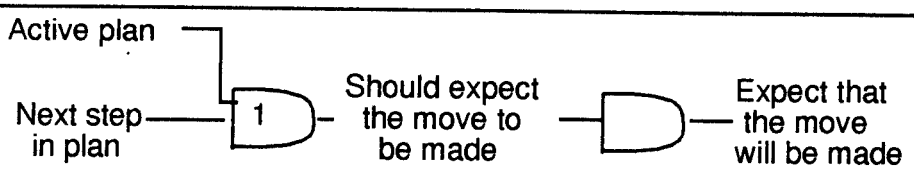


Figure 4.4: Part of the justification structure

antecedent beliefs are true. **Or** gates in justification structures correspondingly indicate rules whose antecedents were disjunctive, and signify that the belief is considered true because one of the antecedents was thought to be true.

In particular, the leftmost **and** gate in figure 4.4 corresponds to the meta-rule in figure 4.2, which inferred the **expect** belief, and the **and** gate labeled **1** corresponds to the rule in figure 4.3, which inferred the **should-expect** belief. The statement *active plan* represents the fact that the plan is currently active, which is the first antecedent of the expectation rule, and the statement *next step in plan* representing the belief that the move is the next step in the active plan, which is computed by the second conjunct in the rule. Collectively, figure 4.4 represents the two layers of CASTLE's justification for its belief in the expectation, corresponding to the final two inferences that resulted in the expectation being posted.

Of course, the system's belief that the given plan should be active is itself justified by the antecedent beliefs of CASTLE's plan selection rule (figure 3.35), namely that the plan was a valid one and that it was the best such plan available. Furthermore, the belief that the active plan is a valid one is itself justified by the plan generation rule which was used to generate the plan. In our example the plan was generated by the rule for two-move lookahead (shown in figure 3.33), which says roughly: *to generate a plan, determine a piece which can move to a new location, and an opponent's piece which can be captured by the attacker from its new location, such that the opponent has no counterplan to the attack.* Extending our justification to include these additional levels gives us the structure shown in figure 4.5. The plan selection rule is indicated by the gate labeled **2**, while the plan generation rule corresponds to the gate labeled **3**. (The gate corresponding to the expectation generation meta-rule has been omitted.)

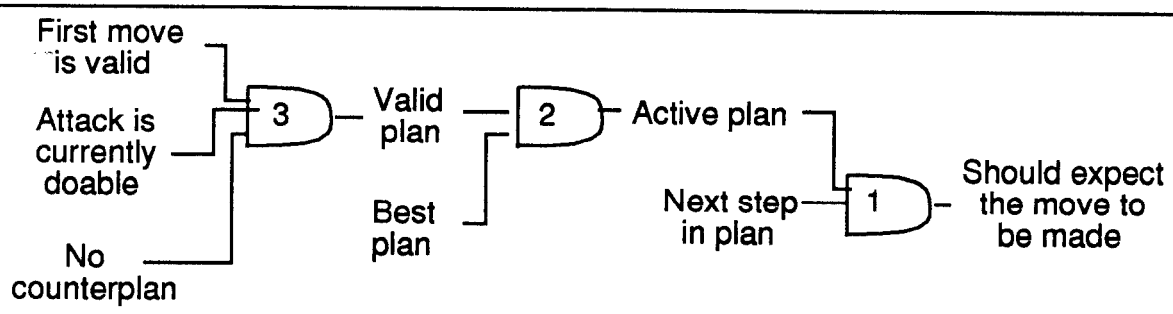


Figure 4.5: More of the justification structure

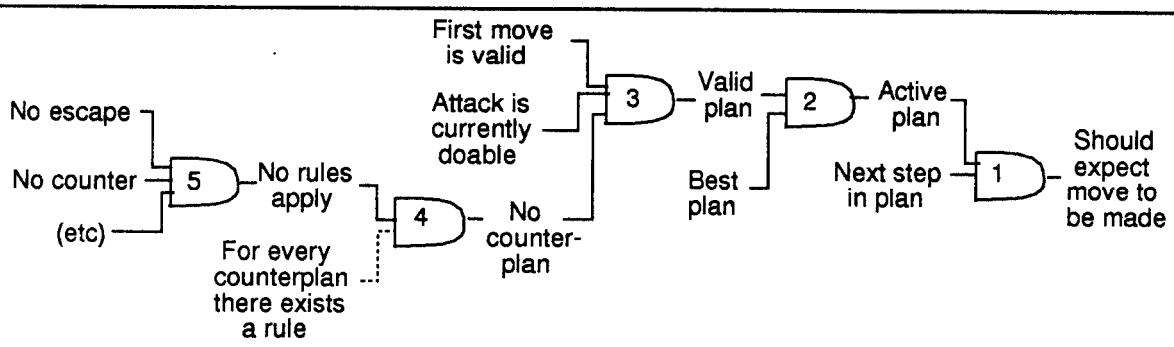


Figure 4.6: The complete justification structure

Several of the leaf nodes in this justification structure are themselves justified, as we see in the complete justification shown in figure 4.6. The belief that the opponent has no counterplan, for example, is justified by a record of the computation that CASTLE engaged in to determine that it was true. Since CASTLE uses its counterplanning component whenever it has to determine possible responses to an attack, the justification for the belief that the opponent will not have a counterplan is that no counterplanning rule was able to determine one (gate **4**). This will in turn be justified by the failure of each counterplanning method to determine a response (gate **5**).

Strictly speaking, however, the fact that no counterplanning rule applied is not truly a proof that there is no possible response to an attack, unless the system makes the implicit assumption that every possible counterplan will be generated by a counterplanning rule. CASTLE records such implicit assumptions, such as the assumption that its counterplanning rule set is complete, along with the rules that rely on them. This assumption is not checked during planning however, and is thus not an antecedent to the rule in question. Rather, the assumption is CASTLE's justification for believing that the rule itself is correct. In other words, CASTLE believes that the lack of a counterplanning rule generating a response implies that there will be no response, and the belief in this implication is justified by the belief that every possible counterplan will have a rule that will generate it. We show this *rule justification* with dotted lines in figure 4.6, indicating that it is the justification for the belief in the implication and is not truly a rule antecedent.

Rule justifications such as these are created when the rule is added to the system. Some of these are added by the programmer when the system is originally designed, while others are added by CASTLE when it learns new rules. We will see how CASTLE creates justifications for learned rules in chapter 5.

4.3 The basic diagnosis algorithm

The task of the diagnosis engine is to search the justification structure for an underlying faulty assumption which led to the failure. An underlying fault is a node in the justification structure that is found to be incorrect, and is not itself justified by other beliefs. In this

```
procedure diag( belief)
    if null(justification( belief))
        then signal-fault( belief)
    false-antecedents ← { ant ∈ antecedents( belief) | incorrect( ant)}
    if false-antecedents ≠ ∅
        then ∀ ant ∈ false-antecedents
            diag( ant)
        else diag(justification-rule( belief))
```

Figure 4.7: The basic diagnosis algorithm

sense the faults returned by the diagnosis engine are always the most basic faults that are implicated in the failure.

We can see from our discussion of justification structures in the previous section that for every failure there will exist a fault (or set of faults[2]) that is implicated in the failure, which is not itself justified. The proof of this hinges on the fact that whenever a belief is faulted, if it is justified, there will be a fault either with one of its antecedent beliefs or with the rule that states that the antecedents imply the belief.

This idea forms the basis of CASTLE's diagnosis algorithm: To diagnose a belief that has been found to be incorrect, check whether any of the belief's antecedents are contradicted by the system's current knowledge. If so, fault the antecedents which have been contradicted and recursively diagnose them. Otherwise, the rule which generated the belief is itself at fault, due to the fact that its antecedents are correct but its consequent is incorrect, and thus it does not represent a true logical implication. The rule itself should then be diagnosed. Whenever a belief or rule is found which is faulty and for which there is no justification, return it as an underlying fault.

This diagnosis procedure is shown in pseudocode form in figure 4.7 as a recursive procedure which is given an expectation (or in general a belief or a rule) that has been found to be faulty. It traverses the belief's justification structure and signals as faults all the terminal nodes of the justification structure which are faulty.

This diagnosis algorithm assumes that it is possible to test, at diagnosis time, whether each belief upon which the faulty expectation was based was, in retrospect, true or false. This determination is shown in figure 4.7 as an invocation of the `incorrect` procedure. In many cases this determination is easy to accomplish. For example, in the *interposition* example the system can know at the time of diagnosis that there was a move for the opponent to disable the attack on the bishop, because it is clear that the move was able to be carried out at the beginning of the sequence of events, and that after the opponent's move the attack was disabled. Some beliefs, however, will be more difficult to test. If CASTLE is unable

[2]We are assuming that all faults that are found are independent, and are ignoring the special issues that arise from multiple interacting faults. This simplifying assumption is discussed in section 4.8 and more generally in chapter 9. All statements about single faults apply also to multiple independent faults.

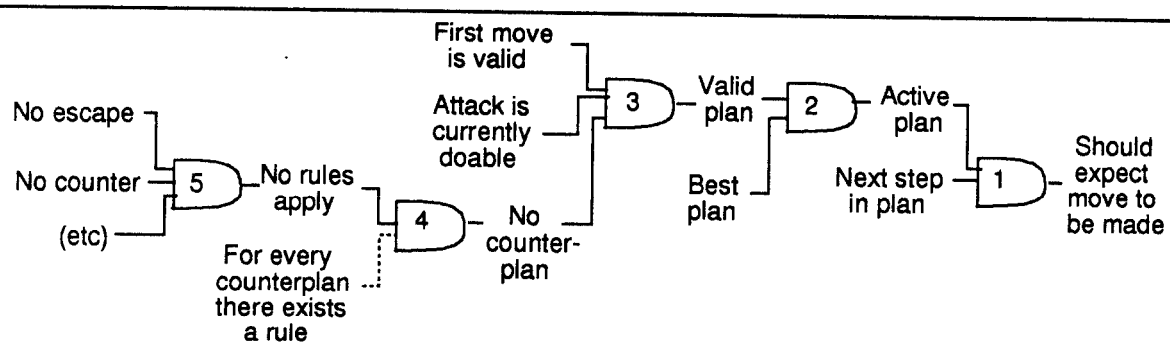


Figure 4.8: The justification structure for *interposition*

to determine the retrospective truth status of a belief, it non-deterministically searches all beliefs that may be faulty. We discuss other approaches to this problem in section 4.8.

4.4 Retrospective belief correctness

In traversing the justification structure, CASTLE's diagnosis engine tests the correctness of each belief implicated in the failure. This correctness check was discussed in the previous section as invocation of the `incorrect` predicate, which tests if a belief that was previously thought to be true can now be shown to be incorrect. Of course, the purpose of checking the beliefs in the course of diagnosis is that CASTLE will have more information available during diagnosis than it did when the belief was originally formulated during planning. This is particularly the case for CASTLE's predictions of future events or of the opponent's beliefs. These types of predictions are made using rules that make the best possible prediction at the time, but when CASTLE is diagnosing a failure the actual outcomes of these predictions may be known. It is therefore imperative that CASTLE evaluate the beliefs it is testing using all the knowledge available at the time of diagnosis, and not merely the knowledge that was available at the time of planning.

Let's look again at the justification structure in the *interposition* example, shown again in figure 4.8. The expectation itself, that the attack will be made, was seen to have failed when a different move was made at the time the attack was expected. To diagnose this belief, `diag` is invoked on the two supporting beliefs, that the plan was active and that the attack was the next step in the plan. The latter is seen to be true by checking that CASTLE had correctly determined the next step in the plan, which could have been at fault if the active plan were a conditional one and CASTLE had miscomputed the next step. For example, suppose CASTLE had a plan to:

> Make `move1`, then evaluate `exp` to decide among two possible next moves

that was represented as:

```
(plan move1 (move-pred (var var) exp next1 next2))
```

If CASTLE miscomputed the value of *exp*, it would then expect the wrong move to be made. In our example this is not the case, however, so diag turns to the *active plan* belief. Since the plan in question ended up being unable to be fully executed, CASTLE concludes that the plan should not have been active, and faults it.

The first supporting belief for the plan being active is that it was a valid plan. CASTLE believes this to be true because its planning rule (figure 3.33) generated it. Testing the plan's validity now, however, merely involves testing the legality of the steps in the plan, which makes it easy to see that the plan is not valid because the second move was not legal at the time it was to be carried out. Another supporting belief for the plan being active is that it was the best plan under consideration. CASTLE exonerates this belief by noting that had the plan been a valid one, it would indeed have been the best plan that was being considered.

The diagnosis engine now turns to diagnosing the decision made by the planning rule, corresponding to the **and** gate marked **3** in figure 4.8. The first antecedent belief, that the first move in the plan (moving the queen laterally) was valid, is checked simply, as is the belief that the attack was a valid move at the time the planning rule was invoked. This leaves CASTLE to diagnose the belief that there existed no counterplan for the opponent to respond to the attack on the bishop. This belief can be seen to be false fairly easily, because it is clear that before the opponent's move the attack was enabled, and that afterwards it was disabled, and thus the opponent's move had the effect of somehow disabling the attack. In this way CASTLE's diagnosis engine can fault its belief that the opponent would have no counterplan without knowing the details of how the opponent's move was in fact a counterplan.

At this point the diagnosis engine can fault either of the two beliefs that support the faulted assumption that no counterplan would exist. These two supporting beliefs are that no counterplanning rule applied and that the set of counterplanning rules is complete. The first can only be tested by actually retrying the counterplanning rules to see if one of them should have generated a counterplan. This is done by testing the antecedents of the rules using retrospective knowledge the same way we have been testing the supporting beliefs themselves. In our example, neither of the counterplanning rules (for escaping and counterattacking) evaluate differently using retrospective knowledge, so the diagnosis engine will conclude that it was correct in believing that neither applied. CASTLE will thus fault its belief that it has a counterplanning rule for all possible counterplans, because there clearly was a counterplan for which no rule existed. Since this completeness assumption is a leaf node of the justification, in that it is not itself justified, the diagnosis engine will stop here and conclude that the fault is that this assumption is false. The diagnosis conclusion is thus that there exists a counterplan for which there is not a counterplanning rule.

4.5 Implementation of belief testing

The methods used for belief testing in the example are summarized in figure 4.9. These methods derive from a general model of past, present, and future tense references in CASTLE's representations. This model encompasses the following:

- The vocabulary that CASTLE uses in planning

Belief	Incorrect in retrospect
Should expect	Did something else happen?
Active plan	Problem carrying out plan?
Next step in plan	Incorrectly compute next step?
Valid plan	Plan steps unable to be executed?
Best plan	Other considered plan now projected to have had better consequences?
Attack will be doable	Was the attack illegal at that time?
No counterplan	Did a move result in the plan's being disabled?
No rules applied	Should any rule have applied?

Figure 4.9: Testing justification structure beliefs

- Actions taken in chess
- Important preconditions and effects of actions

This model must include the relations between the semantics of different expressions in CASTLE's vocabulary. In some cases these relations will be simple "tense shifts," such as the statements *I will make this move next turn* and *I made this move two turns ago.* In other cases these relations will be between predictions and facts, such as the statements *I predict the opponent will make the following move* and *the opponent made this move last turn.* In still other cases the relations will be between theoretical knowledge and real, concrete instances, such as the beliefs *this move disables this other move* and *this move resulted in this other move being disabled.*

This model of relations between tense references is also used in constructing new rules. Since CASTLE's explanations of desired component performance will be in after-the-fact terms, in that they are being constructed at the time of repair, they must be adjusted to be applicable at the time the rule should have been used. This would come up, for example, if an explanation of the desired performance of some component included a reference to the fact that a given plan was able to be carried out. Such an expression would need to be translated into a form that could be used at plan-time, in this case an assertion that the plan in question is a legal one. We will discuss this in more detail in section 5.6.1.

A general theory of relations between tense references is, however, beyond the scope of what is implemented in CASTLE. To approximate such a theory, CASTLE's diagnosis engine uses a set of rules for translating expressions into after-the-fact tests. To test a belief's correctness, the system invokes translation rules that attempt to determine a corresponding post-facto test, and then queries CASTLE's inference engine to determine the truth status of this test.

Consider, for example, the belief that there does not exist a counterplan at a particular time. Retrospectively this belief could be tested by checking whether there was in fact a

```
(def-brule post-facto-cp
  (post-facto (not (counterplan ?cp-meth ?player          To retrospectively verify
                                ?goal ?time ?cp))         the belief that no
              ?current-time)                              counterplan would exist
 <=
  (and (> ?current-time ?time)                            If it's later,
       (move-to-make ?move ?player ?goal2 ?time)          and the move that was made
       (not (= ?goal ?goal2))                           was for a different goal
       (goal-possible ?goal (1- ?time))                   and the goal was unaffected
       (goal-possible ?goal (1+ ?time)) ))                (then the belief is true)
```

Figure 4.10: Post-facto checking for a counterplan

move made that disabled the goal. A rule for performing this retrospective test is shown in figure 4.10, which says roughly:

TO COMPUTE: The retrospective truth of a statement
that there did not exist a counterplan

DETERMINE: The move that was in fact made
and whether the goal was disabled by the move

This rule allows CASTLE to easily determine the truth of the proposition, without depending on the potentially at-fault planning knowledge in the counterplanning component.

In general, the rules used to perform retrospective belief testing must satisfy three conditions. First, they must look at what actually transpired, and not at predictions that can be made. Second, they must minimize their reliance on planning procedures, because it is these very procedures that are likely to be faulty when diagnosis is invoked. Lastly, the tests being performed must obviously correspond to the essence of the belief being tested, and must be a reliable indicator of the truth of the belief.

4.6 Testing quantified beliefs

One particularly difficult type of belief to test is one which is universally or existentially quantified, that is, a belief about a statement being true for all possible values of some variable, or about the existence of a value such that the statement is true. The difficulty in diagnosing these beliefs stems from the "propositional" nature of justification structures.

A universally quantified assumption, for example, is a shorthand for an arbitrarily long conjunction, with one conjunct for each of the possible values of the quantified variable. The number of conjuncts can in theory be infinite if the quantifier scopes over time, or plans, or the like. For this reason it is not feasible to represent and diagnose universally quantified assumptions in this simple manner. The same is true of existentially quantified assumptions, which are shorthands for arbitrarily large disjunctions.

Unfortunately the basic diagnosis algorithm procedure given earlier cannot diagnose a faulted quantified belief unless it is expanded in this way. To enable it to do so, CASTLE's diagnosis engine transforms quantified beliefs into almost-equivalent expressions that can sometimes be resolved with a single query to CASTLE's inference engine. A universally quantified assumption, for example, is determined to be faulty if there exists a value for which the embedded expression is false [Birnbaum *et al.*, 1989]. This corresponds to the mathematical transformation:

$$\forall_x f(x) \Longleftrightarrow \neg\exists_x \neg f(x)$$

which means that the assumption *"forall* `x`, `f(x)` *is true"* is equivalent to the assumption *"there doesn't exist a value of* `x` *such that* `f(x)` *is false.* The diagnosis algorithm then uses this transformed assumption by searching for a value of `x` such that `f(x)` is false, and concludes that the original assumption is faulty if such a value can be found. By making this transformation, an expression that seemed to require a large number of queries to CASTLE's inference engine has been handled using only one query.

Similarly, an existentially quantified assumption can be tested without having to reason about the arbitrarily long disjunction mentioned above, by simply passing it to CASTLE's inference engine and checking whether a value can be found for the variable such that the expression is true. This is because CASTLE's inference engine, like most deductive retrieval engines, implicitly treats all unbound variables as existentially quantified [Charniak and McDermott, 1985, sec. 6.3.3].

Negated quantified assumptions are handled similarly. The quantified assumption (with the negation stripped off) is handled as discussed above, but the query results are interpreted in an opposite fashion. If the original assumption was a negated universally quantified expression, the success of the transformed query (about the existence of a counterexample) indicates correctness of the negated assumption, and failure indicates fault. If the original assumption was a negated existentially quantified expression, the success of the query shows that the negated assumption is faulty.

Note that searching for counterexamples is much more efficient than an analysis of each conjunct or disjunct of an expanded quantified expression, but is not *complete.* Suppose that CASTLE doesn't know of a counterexample that would render the universally-quantified expression faulty, but instead is able to realize inferentially that the universal is false without isolating a particular counterexample. In such a situation CASTLE would not realize the expression's faulty status, and would assume that it's true. Similar situations arise in the other forms of quantified expressions.

The methods used to diagnose quantified assumptions are summarized in figure 4.11. We will now look at diagnoses of this type in the context of several examples.

4.6.1 Universally quantifying over plans

In the *interposition* example, the main assumption being made by the planner is that there will exist a counterplanning rule for each possible counterplan. This assumption is represented by a universally-quantified expression which ranges over possible counterplans.

Assumption	Query	Results interpretation
$\forall_x f(x)$	$\neg f(x)$	Success $\Rightarrow$ faulty
$\exists_x f(x)$	$f(x)$	Success $\Rightarrow$ not faulty
$\neg\forall_x f(x)$	$\neg f(x)$	Success $\Rightarrow$ not faulty
$\neg\exists_x f(x)$	$f(x)$	Success $\Rightarrow$ faulty

Figure 4.11: Diagnosing quantified assumptions

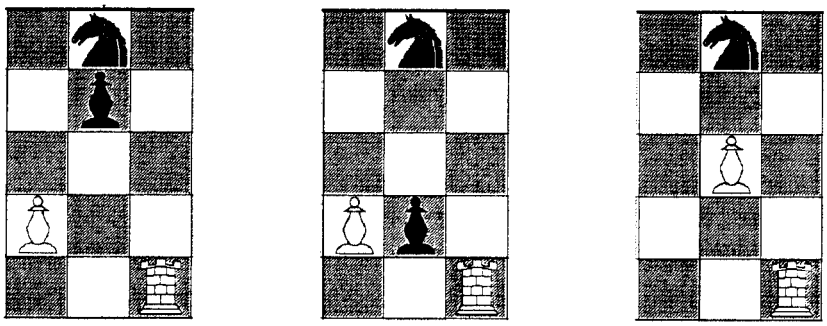
CASTLE's diagnosis engine must be able to examine this belief and transform it into something testable. In section 4.4 we glossed over CASTLE's testing this belief, saying that "there clearly was a counterplan for which no rule existed." While it may be clear to us, we must see how CASTLE can make this determination.

The simplest possibility, testing the subexpression (that there exists a rule) for each possible counterplan, is not feasible because it is impossible for CASTLE to enumerate all possible counterplans: The set of possible plans is arbitrarily large, spanning all the pieces on the board, all the squares they can move to, and also spanning an arbitrary amount of time.[3] Because of these difficulties, CASTLE uses the transformation discussed above to handle the diagnosis. The expression is transformed into the following: *The expression is faulty if there exists a counterplan for which no rule existed.* In other words, the universally-quantified expression is transformed into a search for a counterexample. Since CASTLE does know about the existence of the counterplan which was executed by the opponent, it can resolve this query easily and determine that the expression is faulty.

4.6.2 Existentially quantifying over rules

Once CASTLE has created the transformed expression for testing the universally quantified assumption in the *interposition* example, it has to test the quantified subexpression about the existence of a counterplanning rule. This is handled very simply by making the query to CASTLE's inference engine and seeing whether there exists a rule to resolve the query. In other words, CASTLE checks if any of its counterplanning methods should have generated the opponent's move as a counterplan. This "rechecking" is itself carried out using after-the-fact knowledge to see if any of the methods should have applied even if they didn't originally. In our example, this query will fail, due to CASTLE's not having a rule for counterplanning by interposition. CASTLE's diagnosis engine has thereby concluded that the assumption is false, and that the underlying fault is that it's not true that all counterplans had a rule to generate them. In particular, CASTLE knows that the opponent's counterplan of interposing the knight did not have a corresponding counterplanning rule.

[3]While most of the counterplans we have discussed are single-move, this is not always the case. *Buying time* (figure 3.28) is one example of a counterplanning strategy that creates multi-turn plans.



(a) (b) (c)

Figure 4.12: *En passant* example: Computer (black) to move

4.6.3 Universally quantifying over threats

Suppose the opponent captures a piece of CASTLE's in a way that CASTLE didn't know was possible. For example, if CASTLE's set of threat detection rules (section 3.4) did not contain a rule for detecting *en passant* pawn captures—the capture of a pawn that has made its initial move two squares forward by the opponent's moving a pawn into the intermediate square—it could be susceptible to such a threat without expecting it [Birnbaum *et al.*, 1989]. Unfortunately the expectation failure would result from the opponent's *not* making a different, predicted move, so CASTLE will have to diagnose the failure and determine that the fault is a lack of a threat detection rule.

While we will leave a complete discussion of this example to section 6.3, there is one aspect of the diagnosis that relates to our discussion of quantified assumptions. The underlying fault in the *en passant* example is that there did not exist a threat detection rule for the threat that the opponent carried out. This is represented in CASTLE as a failure of the assumption that *for every threat there will exist a threat detection rule to detect it.* The diagnosis process must be able to examine this assumption and see that it is faulty.

The key difficulty with making this diagnosis is that it is not feasible to enumerate all possible threats, due to the magnitude of the set of threats that can be made by any piece in any square to any other square. Perhaps even more importantly, it is clear from CASTLE's inability to recognize the threat that was indeed carried out that the system's knowledge of possible threats is incomplete. Because of this, CASTLE must make the diagnosis without iterating through the possible values of the quantified variable.

To do this, CASTLE once again transforms the assumption in the way we discussed above into the expression *there will not exist a threat for which there will not exist a threat detection rule.* To diagnose this form of the assumption, the diagnosis engine makes the query *does there exist a threat for which there did not exist a threat detection rule,* and the inference engine returns that there does exist such a threat, namely the *en passant* threat executed by the opponent.

4.7 Types of faults

When the diagnosis process is complete, CASTLE has identified a fault that underlies the expectation failure. This fault will be a knowledge structure that CASTLE thought to be true but that it now sees to be incorrect. These faults will fall into several broad categories: incorrect beliefs or negated beliefs, faulty rules, and incorrect underlying assumptions. As we will see, the most common form of the last of these is an incorrect assumption about the completeness of a component rule set.

4.7.1 Faulty beliefs

The simplest of these faults is an incorrect unjustified belief. A fault of this type will also be easy to repair: The belief can simply be retracted from the system's memory. However, faults of this type do not arise often in practice, because most of CASTLE's beliefs are justified by the rules that inferred them, and justified beliefs will not themselves be signaled as faults. Rather, in such a case a fault will be returned from the faulty belief's justification.

As an example of diagnosing a fault as a faulty belief, suppose CASTLE were designed to maintain memory of the skill levels of its various opponents. For each opponent that it plays, it could remember each strategy that the opponent knows. Whenever its plan recognition rules signaled a plan in progress, CASTLE could record that the opponent knows the strategy being employed. Additional information could be gained when strategies succeed or fail against the opponent. Then, in planning, CASTLE could use this information to predict what strategies will be successful and which the opponent is likely to employ.

Consider what could happen if CASTLE were in the midst of executing a plan when its plan recognition rules signaled that the opponent had the opportunity for a devastating attack. If the strategy involved in the attack is one that CASTLE believes the opponent is familiar with, it should suspend its own plan and defend against the attack. If in the end it turns out that the opponent does not attempt to execute the attack, CASTLE would have to diagnose its decision to suspend its own plan, and would conclude that it's likely that the opponent is in fact not familiar with the strategy that CASTLE recognized. Since the belief regarding the opponent's knowledge is not justified inferentially, but rather observationally, the underlying fault would simply be that the belief was incorrect.[4] As we said above, this fault would then be repaired by retracting the belief.

Even though in principle this type of fault seldom arises, it may arise more often if the system were designed to periodically forget justifications for long-known beliefs. Justification-forgetting of this type would reduce CASTLE's memory load, and may in fact be critical for a complex planner running for a long time. When a justification is forgotten, the only recourse in diagnosis is to consider the belief itself to be the fault. The fault could then be repaired simply by eliminating the belief. While this would not allow CASTLE to repair the flaw in its decision-making mechanism that inferred the faulty belief in the first

[4]There are actually other potential faults that could arise instead of the belief simply being false. For example, the opponent's powers of observation may be limited, or he may have been focused elsewhere, or he may have been executing a more valuable plan. If CASTLE could in some way exonerate these possibilities, it would be left with faulting the belief itself.

place, it would at least allow the system to learn *something* when the information needed to learn more is not available. This will be discussed more in section 4.8.

4.7.2 Faulty negations

Another relatively simple fault that the diagnosis engine can return is that a belief that was thought to be false is in fact true. Such a fault is called a *faulty negation.* In contrast to the *faulty belief* failures, faulty negations can often be the most basic underlying fault, because in CASTLE negations are not justified inferentially but rather are believed to be true due to the absence of belief to the contrary [Hewitt, 1969; Roussel, 1975; Reiter, 1978]. In other words, CASTLE usually believes a negation to be true because it does not know the statement being negated to be true, not because there are inferences made to support the belief in the negation itself. A consequence of this is that negations often are not justified by other beliefs.

As an example of diagnosing a faulty negation, consider the scenario presented above of CASTLE's maintaining a memory of its opponents' skill levels. Suppose that CASTLE is mounting an attack against an opponent using a particular strategy, and assumes in the course of planning that the opponent is unfamiliar with the strategy being employed. This assumption would take the form of the negation of the statement that the opponent is familiar with the strategy. If it turned out that the opponent was in fact familiar with the strategy, and defended against CASTLE's plan successfully, CASTLE's diagnosis engine should conclude that the fault underlying the plan failure was the incorrect belief that the opponent is not familiar with the strategy.

Unfortunately there is no new rule that CASTLE can learn in response to the failure, because there is no reason to believe that the system should have known about the opponent's knowledge of the strategy. Rather, the only possible repair is to add the belief to CASTLE's memory that the opponent is familiar with the strategy.

4.7.3 Faulty rule

Most of the failures that arise in CASTLE's execution are due to faulty inference procedures, not faulty beliefs. In particular, the third type of fault that can arise in CASTLE is an incorrect rule. A faulty rule will be the underlying fault whenever a rule is found in the justification whose antecedents (left-hand side) are all true and whose consequent (right hand side) is false, and which does not itself have an associated justification structure.

For example, consider the rule we saw in chapter 3, shown again in figure 4.13, for posting an expectation that the opponent will execute the best available attack:

TO COMPUTE: An expectation to monitor

DETERMINE: The opponent's highest-valued goal
that can be attained by a single move
that I haven't counterplanned against

```
(def-brule exp-best-opp-attack                                  To determine an
  (should-expect exp-opp-attack ?time ?exp)                     expectation
<=
 (and (max-var-val ?move (vars ?goal-val ?move)                 Find the move
       (and (active-goal opponent                               which the opponent
             (goal-capture ?piece ?loc (move ?move)) ?time)     has a goal to make
            (eval-goal (goal-capture ?piece ?loc (move ?move))  with the highest
                       ?goal-val)                               value
            (not (move-to-make ?my-move computer                which I haven't
                  (goal-counterplan                             counterplanned
                    (goal-capture ?piece ?loc (move ?move)))    against
                  ?time)) ))
       (active-goal opponent                                    retrieve (again)
            (goal-capture ?piece ?loc (move ?move))             the corresponding
            ?time)                                              goal
      (= ?exp (move-to-make ?move opponent                      and expect the
                (goal-capture ?piece ?loc (move ?move))         opponent to make
                (+ 1 ?time)))) )                                the move next turn
```

Figure 4.13: Expecting the best opponent capture

One problem with the rule as it is shown is that it does not take into account the fact that some opponent moves may be disabled by CASTLE serendipitously, if CASTLE was making its move for an entirely different reason but ended up counterplanning against the attack. The reason that this is a problem for the rule in figure 4.13 is that it only filters out opponent attacks which have been intentionally counterplanned against by CASTLE's chosen move. If the counterplanning were accidental, the expected opponent move would of course not be made, and the fault underlying the expectation failure would be this expectation generation rule.

The repair for the rule is fairly straightforward. The statement:

```
(not (move-to-make ?my-move computer (goal-counterplan ...)))
```

which says *my move to make wasn't intended to counterplan against his move,* should be replaced with:

```
(no (and (move-to-make ...)  (counterplan ?my-move ?move)))
```

which would say *my move to make didn't counterplan the opponent's move.* Unfortunately, this type of fault is very difficult to repair, because the rule involved in this case does not have a justification, and thus there is very little knowledge available about the reasoning involved in the rule. This lack of information makes it very difficult for CASTLE to decompose and reconstruct the rule appropriately. Thus it is a design goal of CASTLE that every rule have a justification indicating why it is believed to be true. This would be true automatically of any rules that CASTLE learned from experience, and it should also be true of rules given by the system designer.

4.7.4 Faulty completeness assumptions

The most common fault to arise in CASTLE is the failure of a *component rule-set completeness assumption.* This was the underlying fault in the *interposition* example of chapter 2, and is in fact the underlying fault in most of the examples that will be discussed in chapter 6.

A rule-set completeness assumption is CASTLE's assumption that the rule-set which implements a given component has all the rules that are needed to carry out the component's task correctly. Any system that uses rules for decision-making is constantly making assumptions of this type. CASTLE makes these assumptions explicit, which allows the diagnosis engine to detect when a failure is a result of the lack of a rule.

Consider the use of the counterplanning rule set in the *interposition* example. The planning rule (shown in figure 3.33) says roughly: *to generate a plan, determine a piece which can move to a new location, and an opponent's piece which can be captured by the attacker from the new location, such that the opponent has no counterplan.* As we discussed above, the planning rule tests whether the opponent will have a counterplan by testing whether any of CASTLE's own counterplanning rules apply, and assumes that if none apply then the opponent will not have a counterplan, or to put this another way, that if the opponent will have a counterplan then CASTLE has a rule to generate it. This is an example of a rule-set completeness assumption.

Because the planning rule relies on this assumption, it is represented explicitly in the justification for the plan's correctness (figure 4.6). This explicit representation allows the diagnosis engine to determine that the fault underlying the expectation failure is that the rule set was in fact not complete. A similar situation holds with respect to the *en passant* example of section 4.6, and indeed of most of the examples that we will see in chapter 6. The repair for this type of faulty assumption is to augment the rule set with a rule that covers the situation in which the rule set was found lacking. We will discuss this process in detail in chapter 5.

4.8 Algorithm limitations and extensions

The basic diagnosis algorithm presented here is fairly simple: Recursively examine the antecedent nodes for a faulty belief until finding the most primitive fault. The complexities that have arisen in CASTLE have for the most part involved the testing of individual beliefs: How can the most up-to-date knowledge be used in testing the correctness of a belief; and how can quantified beliefs be properly handled? These issues arise in any application of diagnosis to a planning system, and CASTLE's approach to addressing these issues has formed the bulk of this chapter.

There are, however, several assumptions underlying this approach to diagnosis that point to limitations in CASTLE's diagnosis algorithm. The first is the system's reliance on explicit justification structures. CASTLE's diagnosis engine relies on the fact that justification structures have been associated with every belief and rule in CASTLE's memory. While this is easy enough to accomplish in the domain of chess, it may be untenable in a more complex domain.

If we look at the two types of information in CASTLE's justification structures, however, we can see that we can obviate the need to remember much of it, at the expense of additional diagnosis-time computation to reconstruct it. The portions of CASTLE's justifications that refer to computation that took place can be eliminated entirely, and the diagnosis engine could attempt to reexecute the planning process as a means of reconstructing the justification structures as they're needed. Furthermore, this forgetting is not an all-or-nothing prospect. One approach would be to forget the justifications for beliefs that are no longer frequently referenced. Another would be to have a limited memory for justification structures, and to eliminate the least recently used nodes as memory is needed. A more homogeneous approach would be to eliminate certain types of information but to retain others, such as remembering the rules that led to facts being believed but to require the diagnosis engine to reconstruct which other beliefs were the antecedents to the rule application. In this way we can envision a whole range of possible approaches to limiting the amount of memory needed for records of planning.

The second type of information in justification structures, the record of otherwise-implicit assumptions made by CASTLE's decision-making architecture, is more difficult to eliminate, because by its nature it is not easily inferred. One approach to limiting the memory cost of such justification structures is to maintain only a set of axioms, and to require the diagnosis engine to combine them appropriately to discover the assumptions behind particular faulty beliefs.

A second assumption that CASTLE's diagnosis algorithm makes is that it will be able to retrospectively test all beliefs in the justification structures. This is not surprising, because diagnosing a justification without the ability to perform retrospective tests is analogous to diagnosing a circuit without any equipment to test the circuit components. If CASTLE were unable to determine which antecedent belief of a particular inference were faulty, it would have to either resort to non-deterministic search of all possible antecedents, or give up the diagnosis process and potentially miss the opportunity to learn.

While this may seem like a strong assumption, it is entirely consistent with the claim that CASTLE implements a powerful *knowledge intensive* approach to learning. The planner-diagnosis algorithm presented in this chapter has the ability to diagnose faults in a very complex structure, provided it is given enough knowledge to do so. Part of this knowledge is a model of retrospective belief analysis.

A third assumption, and an important limitation, of CASTLE's approach to diagnosis is that it relies on the existence of independent faults. Components are assumed to be either faulty, in which case there is a specific behavior that should have been generated but was not, or not faulty, in which case all desired behavior has been generated. CASTLE therefore does not support the idea of combined faults, when one component is only faulty in its relation to other components. Future research is necessary to determine how this can be accomplished within CASTLE's framework [Freed *et al.*, 1992]. It will also require modification to CASTLE's approach to learning discussed in the next chapter.

From the perspective of a designer of decision-making architectures, however, it is often possible to avoid multiple dependent faults by carefully constructing the architecture to maintain individual components for all likely faults. If two components are likely to interact

in a way that leads to a fault, a third component can be used to moderate between them. Dependent faults would then be treated as faults of the moderating component.

The motivation of CASTLE's approach to diagnosis is the assumption that walking the path from the expectation to its cause is more efficient than other methods of searching all the underlying assumptions. In CASTLE this appears to be the case, but in other situations it may not be, and there are numerous alternatives that should be considered. One would be to collect all leaves of the justification structure at the time the justifications are created, and search through them at diagnosis-time to find which are faulty. This would be preferable to CASTLE's approach if the system were unable to retrospectively test some of its nodes, because when nondeterminism is introduced the number of nodes examined in CASTLE's approach will approach the number of leaf nodes. It might also be a preferable approach if the inner nodes were more expensive to test than the leaves. Furthermore, if a system's repair module was only able to repair a small number of the possible faults, the approach to diagnosis by leaf-search could be made more efficient by only recording those leaves that represent possible repairs. In such a case the approach might be better than CASTLE's.

Hybrids of these two approaches are also plausible. One hybrid approach would be to start with CASTLE's approach and switch to leaf-search whenever the system is unable to retrospectively test a belief. In such a case the leaf-search diagnosis algorithm would only have to test those leaves that are descendants of the known faults. In general, which diagnosis algorithm to use depends on many aspects of the domain and the program's abilities. Most of the issues discussed in this chapter, such as retrospective testing and handling quantified beliefs, would be applicable in most other algorithms as well.

There are several enhancements of CASTLE's diagnosis algorithm that are possible. Some of these are heuristics in traversing the justification structure, such as *"when all but one antecedent have been eliminated, the last can be faulted without testing."* There may also be times that traversing the structure is more difficult than simply examining all possible leaf nodes, and heuristics can be used to see when this would be a better approach. Lastly, justification pruning techniques that would eliminate all substructures whose values cannot in any way have changed since plan-time would eliminate unnecessary computation in diagnosis.

Chapter 5

Repairing planner faults

The previous two chapters discussed how CASTLE's decision-making architecture makes decisions and how the diagnosis engine traces failures of these decisions back to underlying faults in the decision-making process. Representations of these faults are then passed to CASTLE's *repair module,* which has the task of effecting a repair. As we have discussed, the *faults* generated by the diagnosis engine are faults in CASTLE's decision-making architecture, not in the plans that were executed. Correspondingly, the repair module fixes the planner, not merely the plans that were being executed when the failure arose.

Because the faults that we are discussing are faults in CASTLE's planner, the repair module is always focused on a faulty element of CASTLE's architecture. In particular, each of the types of faults that was discussed in chapter 4 will focus the repair module on a specific part of CASTLE's decision-making model that needs to be repaired. If the fault is that a component's rule set is incomplete, for example, then the repair module can devote its attention to augmenting the rule set to make it complete. Alternatively, if the fault is that a particular rule is incorrect, then the repair module can focus on repairing that rule.

The repair module also knows more than just the identity of the faulty element of CASTLE's architecture—it also knows what performance was desired from that element in the situation in which it failed. For example, if the fault is that a rule set is incomplete, the diagnosis engine will also have determined the computation that should have been performed by the rule set that was not performed due to the incompleteness. Alternatively, if the fault is an incorrect rule, the diagnosis engine will have determined *how* it was incorrect, in terms of a computation that should have been performed differently. Given this information, the repair module has to determine what change can be made to the planner to fix the fault and thereby insure that similar failures do not occur again in the future. This chapter will discuss how this repair process takes place.

Chapter outline

This chapter will discuss the following:

1. How the repair module is invoked and what computation it performs
2. How new rules are constructed for component rule sets

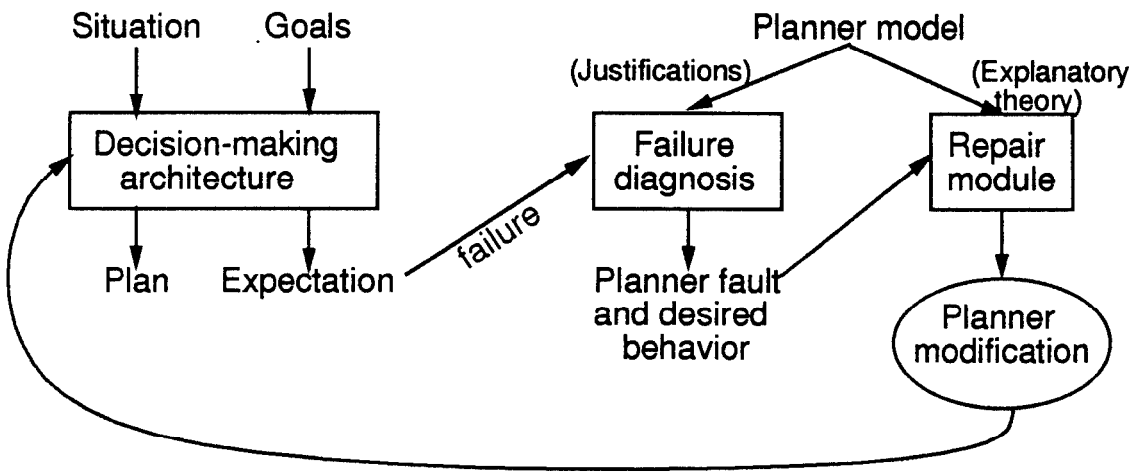


Figure 5.1: The learning process in CASTLE

- The use of explicit component specifications
- Explanation-based learning

3. The vocabulary used in constructing explanations
4. Integrating new rules into the planning architecture
5. Repairing other types of faults
6. Possible extensions of the repair module

Chapter 6 will then give a number of examples in detail.

5.1 Invoking the repair module

As we have said, CASTLE invokes its repair module for each underlying fault that is returned by the diagnosis engin. For each fault, the repair module is given a representation of the fault along with a description of the desired behavior that was not achieved. The module's task is to generate a modification to CASTLE's planner that will repair the fault and achieve the desired behavior in the future.

Consider the fault that must be repaired in the *interposition* example of chapter 2. In that example, we saw the sequence of events shown in figure 5.2, in which CASTLE expected its attack on the opponent's bishop to succeed. It had assumed that the opponent would have no way to counterplan against its attack in figure 5.2(b), and this incorrect assumption led to the attack's failure in figure 5.2(c). Underlying this failure was the fact that CASTLE had an inadequate set of counterplanning rules, and in particular was missing a rule for counterplanning by interposing pieces. For this reason the system did not foresee the opponent's counterplan.

In response to the failure, CASTLE invokes its diagnosis engine, which determines that the underlying fault was that the system's counterplanning rule set is missing a rule, and that the missing rule should have indicated that the opponent's move was a counterplan to the computer's attack. It is these two pieces of information—the fact that the rule set is incomplete along with the description of the desired behavior—that are passed to the repair module, along with a description of the situation in which the failure took place.

Given this information, the repair module generates a modification to CASTLE's planner that will fix the fault. In our example, this modification will consist of a new rule that should be added to the counterplanning component's rule set. More specifically, the rule should encode the idea of counterplanning by interposing a piece along the line of attack. It is the construction of this rule from the fault, the desired behavior, and the situation description that is the task of the repair module.

Of course, in theory the repair module could simply construct a new rule that will produce exactly the desired behavior, in the exact situation in which the failure occurred, without any generalization to other situations. While this will, strictly speaking, repair the fault, it will clearly not be an effective approach to learning, because CASTLE would have to learn methods separately for each situation in which they apply. In our example, this would entail learning separate rules for interposing pieces for every possible combination of attacking piece and location, target piece and location, interposing piece and location, and location on the line of attack. This is clearly impractical as a way to learn to plan, and clearly some amount of generalization is necessary to effectively learn. The rule construction method that we will discuss in this chapter will allow CASTLE to construct rules that are appropriately generalized.

5.2 Constructing new rules

Most of this chapter's discussion of fault repair will focus on repairing a single type of fault, namely incompleteness of a component rule set. We treat such a failure as paradigmatic primarily because this type of fault can be repaired in the most principled and deliberative

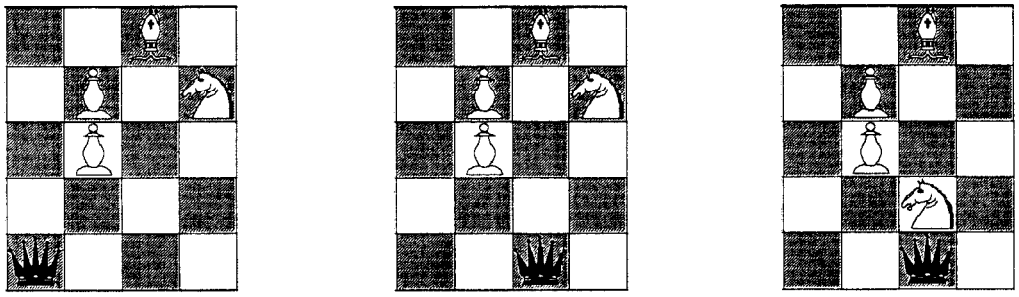


Figure 5.2: *Interposition* example: Computer (black) to move

fashion, as we touched on in section 4.7. The reason for this is that although the completeness assumption which is faulted is not justified *per se,* there are nevertheless other sources of information available that the learner can use in computing a repair. In particular, the *component specifications* discussed in chapter 3 describe the computation that should be performed by the component as a whole. As we discussed in chapter 1, this component-level information gives CASTLE the leverage it needs to learn a variety of types of concepts.

Underlying the idea of modeling the architecture in terms of components is that there is a high degree of regularity in the nature of the system's rules—e.g., the tasks they carry out and their methods of operation—to warrant reasoning about them in groups. To be effective in learning, the system's knowledge about its rules must be expressed at the appropriate level, which we claim is the level of cognitive tasks. In other words, to define an architecture that can learn effectively, it is necessary to specify what it needs to learn at the appropriate level of generality, in terms of the cognitive tasks being carried out. In chapters 6 and 7 we will present CASTLE's approach to modeling at the task level by discussing a set of examples in depth.

5.2.1 Augmenting an incomplete rule set

When the repair module is given a fault specifying that a component rule set is incomplete, the task is to construct a new rule which appropriately augments the component, using the available information. The first and most obvious constraint on the form of the new rule is that its consequent must match the format for invocations of the component in which it will reside. As was discussed in section 3.1, component rule sets contain backward-chaining rules that are used to resolve queries that are made to CASTLE's inference engine. Each component has an *invocation format* which is the form of queries which the component can be used to resolve. The rules in a component's rule set are all of the form *"the consequent is true if the antecedents are true,"* where the consequent is of the same form as the invocation of the component. In this way the rules in a component rule set can be used to resolve queries that match the invocation format of the components in which they reside.

Clearly, then, if the repair module is attempting to construct a new rule for a particular component, the new rule's consequent must match the invocation format of the component. More to the point, the invocation format will fully specify the consequent of the rule being learned. In the *interposition* example, therefore, the rule to be constructed must minimally be of the form:

```
(def-brule learned-counterplan-1
  (counterplan cp-learned-method ?player ?goal ?time ?plan)
<=
   rule antecedent here)
```

because this is how the counterplanning component, and therefore its rules, are invoked.

5.2.2 Explaining component behavior

The next, and more difficult step in learning a new rule is constructing the rule's antecedent. The antecedent must be an expression that describes the situation in which the desired component behavior should be achieved. Put another way, it must be an expression that is true in the situation of the failure, and whose truth should cause the component to behave in a particular way. In our example, therefore, the antecedent must be an expression which is true in the situation of figure 5.2(b), and which implies that the knight can be moved onto the line of attack to counterplan against the queen's attack.

To construct such an expression, the repair module must know *why* the move that the opponent made was in fact a valid counterplan. Unfortunately, while the diagnosis engine determined that the move was in fact a successful counterplan, it did so in a *non-constructive* way, namely by observing that the computer's attack was valid before the opponent's move and invalid afterwards. While this is certainly a correct indicator that the opponent's move was a valid counterplan, it is not sufficient to recognize similar moves in the future. CASTLE must instead generate a *constructive* explanation, in terms of the move itself and how it invalidated the attack, rather than merely in terms of the consequences in the example. Such a explanation will then be used to construct the new rule's antecedent.

Before this explanation can be generated, however, CASTLE must know what it is that needs to be explained. How can CASTLE determine why a component should have behaved in a given way? If all that CASTLE knew about each component was the rules that make up its rule set, then such an explanation could never be constructed, because the rule set is missing exactly the rule that would apply in the situation being analyzed. What is needed is a notion of the *intention* of the component's computation.

CASTLE represents this intention in the form of the *component specifications* that were discussed for each of the components in chapter 3. These specifications, which are part of CASTLE's self-model, describe the intended behavior of each component. CASTLE represents these component specifications as backward-chaining rules, and uses its inference engine to construct explanations of desired component performance as so specified. Performance specifications thereby serve as a bridge between CASTLE's representation of its component behavior and the knowledge that it has available for generating constructive explanations.

A performance specification for the *counterplanning* component is shown in figure 5.3, which says roughly:

TO COMPUTE: The reason that a move was a counterplan to an attack

DETERMINE: The move that made up the counterplan
that disabled the threat
and was a valid move at the time of the counterplanning

To construct an explanation of why a particular move is a valid counterplan, a query is made to CASTLE's inference engine of the form:

```
(method-spec counterplan
  (counterplan player (goal-capture piece loc threat))
               time counterplan)
```

```
(def-brule meth-spec-cp
  (method-spec counterplan
    (counterplan ?player (goal-capture ?piece ?loc ?threat)
                 ?time ?counterplan))
<=
  (and (= ?counterplan (plan ?cp-move done))
       (expl-disabled ?cp-move ?threat)
       (move-doable ?cp-move (world-at-time ?time)) ))
```

The specification of the CP component invoked on a goal to capture a piece: a CP is a move that disabled the threat, and was feasible

Figure 5.3: Specification for the counterplanning component

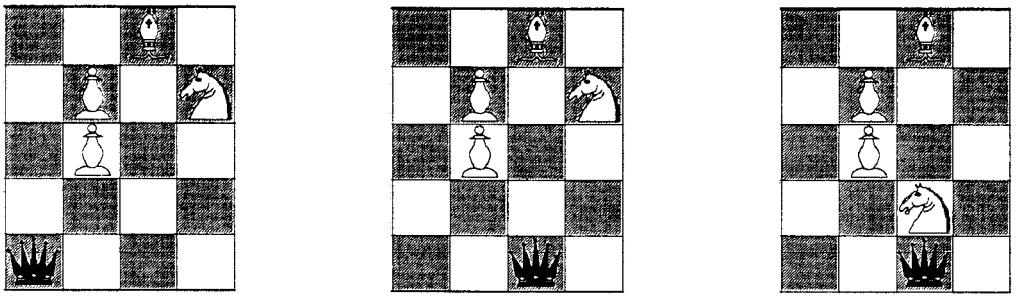


Figure 5.4: *Interposition* example: Computer (black) to move

with the appropriate *player, piece, location, threat, time,* and *counterplan* filled in. The rule that encodes the component specification will then perform the inference necessary to see why the move is indeed a counterplan. If the query is successful, the `method-spec` belief that is inferred will have an associated justification that will serve as an explanation for the specification. In other words, the record of the inferences that are performed in proving that the move was indeed a counterplan will make up the explanation.

Consider how this process would take place in the interposition example, shown again in figure 5.4. After the diagnosis process is completed, CASTLE knows that the fault was that the counterplanning component has an incomplete rule set, and that the missing rule should encode the knowledge that the opponent's knight moving was a counterplan against the computer's attack on his bishop. The first step in the repair process is now to explain why the knight's move was a counterplan. To construct this explanation, CASTLE makes a query of the form:

```
(def-brule expl1                          To explain how one move
  (expl-disabled ?move1 ?move2)           disabled another move
<=
  (and (expl-move-precond ?move2 ?pre)    find a precondition of the second
       (expl-move-effect ?move1 ?eff)     and an effect of the first
       (expl-conflicts ?pre ?eff) ))      such that they conflict
```

Figure 5.5: Explaining a move's disabling another move

```
(method-spec counterplan
  (counterplan opponent
    (goal-capture bishop (loc 1 5)
      (move computer (capture bishop) queen (loc 7 5) (loc 1 5)))
    (time 2)
    (plan (move opponent move knight (loc 2 6) (loc 4 5)) done)))
```

The rule in figure 5.3 will then be used to perform the inference necessary to determine the truth of this query. After extracting the opponent's move (`?cp-move`) from the representation of the one-move plan, the inference rule will cause a sub-query to be made to explain how the move disabled the computer's intended capture:

```
(expl-disabled (move opponent move knight (loc 2 6) (loc 4 5))
   (move computer (capture bishop) queen (loc 7 5) (loc 1 5)))
```

It is in resolving this query that most of the explanatory inference takes place. The inference engine will resolve the query using the rule shown in figure 5.5, which says roughly: *to determine whether one move disables another, see whether there is an effect of the first move and a precondition of the second move that conflict with each other.* Basically, this rule makes use of a domain theory in which the preconditions and effects of moves have been reified and can be manipulated directly. If there is an effect of the disabling move that conflicts with a precondition of the disabled move, the disablement has been proven.

The first step is to retrieve a precondition of the disabled move. A rule for one such precondition is shown in figure 5.6, which says roughly that *a precondition for a move's being made is that no location on the line of movement is occupied, unless the piece being moved is a knight.* A similar rule is used to retrieve an effect of the disabling move. This rule is shown in figure 5.7, and says roughly that *one effect of a move is that the piece being moved will be at the destination location.*

After retrieving a precondition of the disabled move and an effect of the disabling move, CASTLE must determine that the two conflict with each other. Viewing this problem, we can see that the two clearly conflict, because a square being occupied obviously conflicts with a square being empty, assuming that the two squares match. CASTLE performs exactly this reasoning using a set of rules for matches and conflicts, which reason about the vocabulary

```
(def-brule expl2
  (expl-move-precond (move ?player ?move-type ?piece      A move has a precondition
                           (loc ?r1 ?c1) (loc ?r2 ?c2))
    (not (and (loc-on-line ?r3 ?c3 ?r1 ?c1 ?r2 ?c2)       that there is no square
              (at-loc ?other-piece ?other-player           on the line of attack
                      (loc ?r3 ?c3) ?time))))              that is occupied
<=
  (and (not (= ?piece knight))) )                          if the piece is not a knight
```

Figure 5.6: A precondition to a move's execution

used to represent the preconditions and effects. One such rule is shown in figure 5.8, which says roughly that *a precondition that two conjuncts will not both be true conflicts with an effect that matches one of the conjuncts, if the other conjunct is true.*[1] In our example, this rule says that the precondition *no square on the line of attack is occupied* conflicts with the effect *the piece is on this square* whenever the newly occupied square is on the line of attack. Since in our example the knight's new location is in fact on the attempted line of attack, the precondition and effect do in fact conflict, and the disablement has thus been demonstrated.

The rules that we have seen make up a portion of CASTLE's vocabulary for explanation construction. This vocabulary explicitly represents knowledge that is otherwise wired into the rules of CASTLE's planner, such as the preconditions and effects of moves. It also represents how these conditions affect other conditions, which would not otherwise be represented in CASTLE. This knowledge permits CASTLE to determine that the crucial factor in the knight disabling the queen's attack is that the square that the knight moved to was on the line of attack. This vocabulary will be discussed in more detail in section 5.4.

CASTLE then has to infer the final conjunct of the rule in figure 5.3, to show that the move that the knight made was in fact possible to make. This is obviously trivial to show in our example where CASTLE is reasoning about a move that has already been made, but since the explanatory rules that we're discussing are part of a general theory, it's necessary to state

[1]The rule in figure 5.8 detects conflicts with the first of the two conjuncts in the precondition. A second rule, given in appendix B, covers the case of the second conjunct conflicting.

```
(def-brule expl3
  (expl-move-effect                                        an effect of a move
    (move ?player ?move-type ?piece ?loc1 ?loc2)           is that the piece is now
    (at-loc ?piece ?player ?loc2))                         at the new location
<=
  ())
```

Figure 5.7: An effect of a move's execution

```
(def-brule expl4                                    A condition that two things
  (expl-conflicts (not (and ?conj1 ?conj2)) ?conj2) aren't both true conflicts with
<=                                                  the second being true
  (true ?conj1))                                    if the first is already true
```

Figure 5.8: Conflict between two conditions

that a counterplan must be a move that can be made. More importantly, the inferences that CASTLE is performing here will eventually be part of a new counterplanning rule, and it is necessary for the new rule to check that it is only proposing counterplans that are in fact able to be carried out.

CASTLE infers that the move was doable because it is a legal move, the knight started out at the first location, and the destination square was empty. Had the opponent's move been a capture, it would have been seen to be possible because the target square is occupied by a piece of the computer's. With this inference being made, CASTLE has proven that the desired counterplanning component performance generated by the diagnosis engine was in fact correct performance for the counterplanner.

In some situations the diagnosis engine may be incorrect in assigning fault to a particular component. This may be due to there having been several candidates for repair which could not be distinguished using the limited inferencing capabilities of the diagnosis engine. Another reason for incorrect diagnosis engine output is that some of CASTLE's retrospective diagnosis rules are uncertain, and it is only in attempting to formulate an actual repair that it can be seen whether the component is in fact faulty. Each of these situations can easily arise with respect to components such as goal activation which are inherently difficult to match exactly with observable features of the environment. An incorrect diagnosis will result in the repair module being unable to explain the desired behavior. When this happens, control is passed back to the diagnosis engine for further diagnosis.[2] Passing control back to the diagnosis engine for further diagnosis may be desirable even after a successful repair, because it will sometimes enable CASTLE to learn more than one rule from a single example [Krulwich *et al.*, 1992a].

In the interposition example, however, determining that the desired counterplanner performance is in fact "correct" is not so critical, because it is clear from the example that the opponent's move in fact disabled the computer's plan, and thus was a counterplan. What is important about the proof that has been described is that the inferences it comprises constitute an explanation of *why* the desired counterplanner performance was in fact desirable. A record of these inferences, and thus of the explanation itself, is maintained as a justification for the truth of the query made from the component specification. In other words, the computation that has been described was in the service of inferring the `move-spec` expression that we saw earlier, and the inferences themselves are recorded in the justification of the inferred belief, which states that the opponent's move was a valid counterplan to the

[2] A more tightly coupled form of this process is discussed in [Freed, 1991].

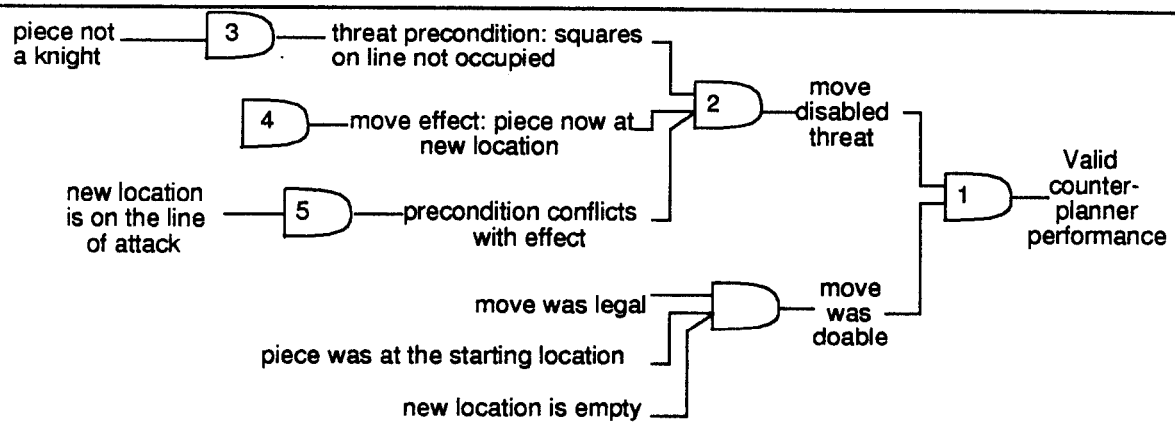


Figure 5.9: The justification for the desired counterplanner performance

computer's plan.

This justification is shown in figure 5.9. The **and** gates labeled **1** through **5** correspond to the inferences made by the rules in figures 5.3 through 5.8. Figure 5.10 shows the same structure in a form more conventionally used for explanations, as an inverted tree with the root being the explained belief and the leaves being the grounded supporting beliefs.

5.3 Explanation-based learning

CASTLE's next task is to use the explanation it has generated to construct a new rule for the incomplete component. This is accomplished by invoking *explanation-based learning* [DeJong and Mooney, 1986; Mitchell *et al.*, 1986] to determine the most general conditions under which the explanation holds. The representation of this condition will then be encapsulated into a new rule for the incomplete component.

5.3.1 EBL's functionality

The EBL algorithm is designed to take an explanation of why a set of facts (often called the *training example*) implies a conclusion (called the *target concept*). The algorithm generalizes the expressions in the training example and the values referred to in the explanation as much as possible. The generalization takes place in two ways:

- EBL uses the most general expressions possible
- EBL substitutes variables for constants wherever possible

The first of these, the use of more-general expressions, is accomplished straightforwardly by using the expressions that are the "highest" in the explanation tree,[3] subject to their

[3] "Higher" in the explanation tree corresponds to "rightwards" in a justification structure, meaning closer to the original query being inferred.

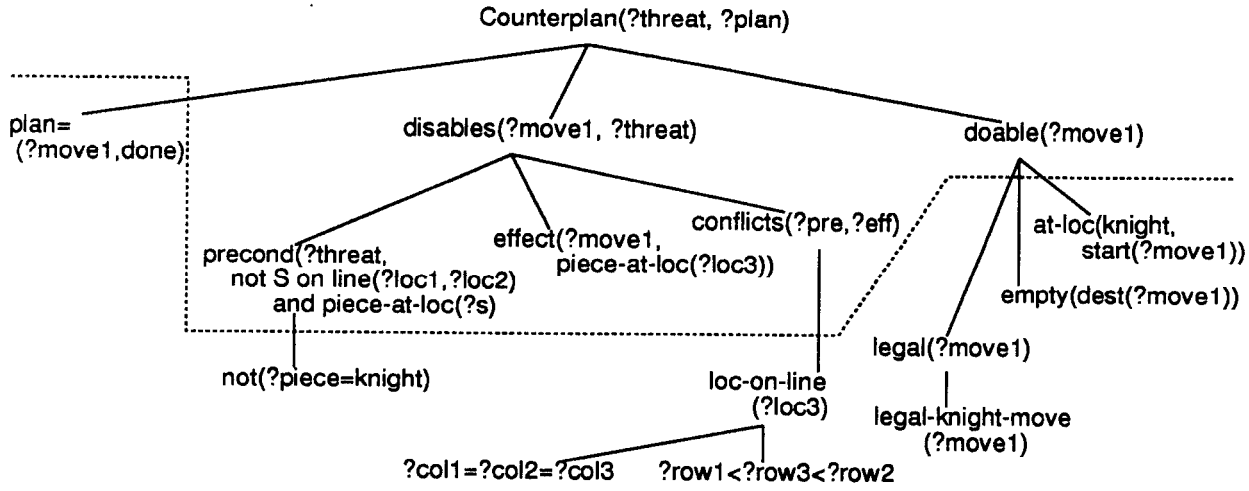


Figure 5.10: Explanation of desired counterplanner performance for interposition

satisfying an *operationality criterion* [DeJong and Mooney, 1986; Mitchell *et al.*, 1986; Segre, 1987]. The point in trying to use expressions higher in the explanation tree is that lower expressions reflect particular ways in which higher expressions can be true, and there are likely to be other ways that the higher expressions can be true as well. Using expressions that are higher in the tree will thus result in the expression being applicable in more situations, in that it will apply whenever the higher expression is true, not only when it is true in the way it was true in the example.

There is a limit to how general an expression will be useful, however, because in principle the most general expression capturing the explanation is a variablized form of the root node, in this case the `meth-spec` belief. The problem is that this expression, the component specification itself, is not effective to use in practice. This notion is captured by EBL's *operationality criterion*. The idea of operationality is that the goal of the EBL process is to construct representations of the situation that can be used more effectively than the root of the explanation. The notion of whether an expression can be computed easily enough to be useful in the rule resulting from learning is the *operationality* of the expression. We will discuss issues involved in determining operationality in section 5.5.

Figure 5.10 shows the explanation in the *interposition* example. The dotted line dividing the tree distinguishes operational expressions from non-operational ones [Braverman and Russell, 1988]. The expressions above the line are not operational, and cannot be used effectively in practice. The expressions below the line are considered operational, and can be used. In our example, for instance, the EBL algorithm should select the `legal` expression in place of the `legal-knight-move` expression. This will clearly broaden the applicability of the resulting expression, in that it will refer to all moves that are legal and not merely to all knight moves that are legal. Similarly, EBL will select the `loc-on-line` expression, and not the expressions stating that the columns of the three locations are equal and the rows line up, so the resulting expression will account for any square that is on the line of

attack between two others, not merely in the case where the squares are vertically aligned. On the other hand, it will not include the expressions that refer to preconditions and effects of actions, because these are not considered operational.

The second form of generalization, the substitution of variables for constants, is accomplished by looking at the inference rules used in constructing the explanation and determining which constants are necessary for the correctness of the explanation and which are merely the details of the example. In our example, EBL should realize that the particular squares on the board are irrelevant as long as the statements made about them (`at-loc`, `legal-move`, `loc-on-line`, etc.) hold. On the other hand, the statement that `not(?piece=knight)` is only appropriate for knights, because they are the only pieces whose lines of attack cannot be blocked, so this expression cannot be variablized. Similarly, because the `precond`, `effect`, and `conflicts` expressions are not operational, the explanation is only known to be correct for the particular preconditions and effects that were found, so the variables `?pre` and `?eff` cannot be generalized.

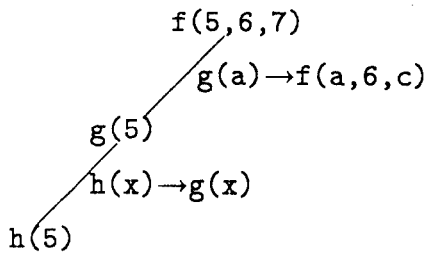
5.3.2 EBL as implemented in CASTLE

CASTLE's implementation of explanation-based learning is based on the "unification-based approach to EBL" [DeJong and Mooney, 1986]. The approach involves traversing the explanation structure in a depth-first fashion and collecting constraints imposed by the inference rules on the applicability of the explanation. CASTLE also augments the standard EBL algorithm in several ways, most notably in the handling of explanation leaves that are not operational.

CASTLE's EBL algorithm traverses the explanation in a depth-first fashion, collecting two types of information along the way. The first is a set of variable bindings that reflect the most general form the explanation can take and still be valid. These binding requirements come from the inference rules used in the explanation process, some of which are only valid for certain values of some variables. EBL collects these bindings in order to generalize the explanation as much as is supported by the inference rules employed. The second type of information collected in EBL's traversal of the explanation is a set of expressions that together constitute the most general expression whose truth entails the truth of the component specification which was explained. These two pieces of information will allow CASTLE to construct a new rule based on the explanation.

At each node in the explanation, an expression is shown to be true because it is inferred by an implication rule. Any constraints in the antecedent of the rule reflect constraints that must be present for the explanation to be true, and thus should be included in the generalization bindings. At any point in the traversal, any expression can be transformed into a more general expression that is supported by the explanation by specifying it with the generalization bindings. When EBL has traversed the entire explanation, the generalization bindings can be applied to the explanation's root node (i.e., the target concept) to derive the expression whose truth is implied by the general form of the explanation. The bindings are also used in the course of traversal to examine the general form of the current node's expression.

Whenever EBL comes to a node in the explanation that is considered *operational,* it



Target concept: f(a,b,c)
Training example: f(5,6,7)
Generalized antecedent: h(x)
Generalized consequent: f(x,6,c)
General rule: h(x)→f(x,6,c)

Figure 5.11: EBL example

constructs the most general form of the expression and adds it to the list of generalized expressions. More specifically, EBL returns the antecedent of the inference rule used at that node. This expression can then be specified with the generalization bindings to produce an expression that is exactly as general as is supported by the explanation as a whole.

Consider how EBL would handle the simplified example explanation fragment shown in figure 5.11.[4] The target concept of the explanation is f(a,b,c), and the example being used for learning is f(5,6,7). EBL first examines the inference that led to the belief in f(5,6,7), and sees from the supporting inference rule that f(a,b,c) is only supported by this explanation when b=6. The constraint b=6 is thus added to the set of generalization bindings. Next, EBL examines the g(5) node, and sees from its supporting inference that no constraints are placed on the value of x other than h(x) being true. Next, the EBL process hits the node h(5), which is a ground fact in the system's database. Assuming that the predicate h is operational, EBL will add h(x) to the set of general antecedent expressions. EBL will then conclude that the general expression that implies the target concept is h(x), and will use the generalization bindings to conclude that this is true for all target concepts of the form f(x,6,c).

A non-standard feature of CASTLE's EBL algorithm is its ability to generalize explanations with leaves that are not all operational.[5] In our simple example, suppose that the predicate h was *not* considered operational. In that case, h(x) could not be added to the set of generalized antecedents, so EBL would need some way to record that the proof was only valid when h(x) was true. CASTLE's implementation of EBL accomplishes this by adding the constraint x=5 to the set of generalization bindings. EBL then uses the match between g(a) and g(x) to propagate the constraint and concludes with a less general result, that f(5,6,c) is always true, with no necessary antecedents.

This ability to handle explanations with non-operational leaves is very useful when using a vocabulary as complex as CASTLE's representation of action preconditions and effects. The only additional assumption being made is that the truth value of an expression is determined fully by its parameters. In our simplified example, if CASTLE made the generalization

[4]This example has been made as simple as possible in order to explain details of the EBL algorithm.

[5]See, e.g, [Mitchell *et al.*, 1986, p. 52].

discussed above with h(x) not being considered operational, and h(5) was not always true, the generalization that f(5,6,c) is always true would be incorrect. This assumption is valid in CASTLE and in many other systems, because facts in the system's database (whose truth values may change over time) are considered operational. Expressions that are inferred by explicit rule application, on the other hand, most often have their truth values determined solely as a function of their parameters. The same is often true of the system's low-level primitives. However, this assumption would not hold, for example, for expressions that query the environment or in other ways access external information.

To summarize, the EBL algorithm performs the following with each explanation node on which it is invoked:

1. If the current node is a leaf that is operational, return its variablized form and no bindings requirements

2. If the current node is a leaf that is non-operational, return the bindings required for it to be true, and no new generalizations

3. If the current node is an operational non-leaf, return its variablized form with no bindings requirements

4. If the current node is a non-operational non-leaf, recursively generalize the supporting beliefs and collect the resulting generalizations and bindings requirements

5.3.3 Running EBL on the interposition explanation

Let's now see how EBL will generalize the explanation produced in the interposition example. The algorithm starts out with the root node of the explanation tree, which states the component specification for the counterplanner performance in the example, namely that it should have generated the knight move as a possible counterplan to the system's plan to capture the opponent's bishop:

```
(method-spec counterplan
  (counterplan opponent
    (goal-capture bishop (loc 1 5)
      (move computer (capture bishop) queen
            (loc 7 5) (loc 1 5)))
    (time 2)
    (plan (move opponent move knight (loc 2 6) (loc 4 5))
          done)))
```

The system then proceeds to traverse the explanation of this belief, shown again in figure 5.12 with labels on the expression nodes. The first subexpression (labeled **2**) is the equality statement used to extract the disabling move from the representation of the counterplan. CASTLE's EBL algorithm handles equality statements by recording the constraints between the variables on the two sides of the equality, which in our example state that:

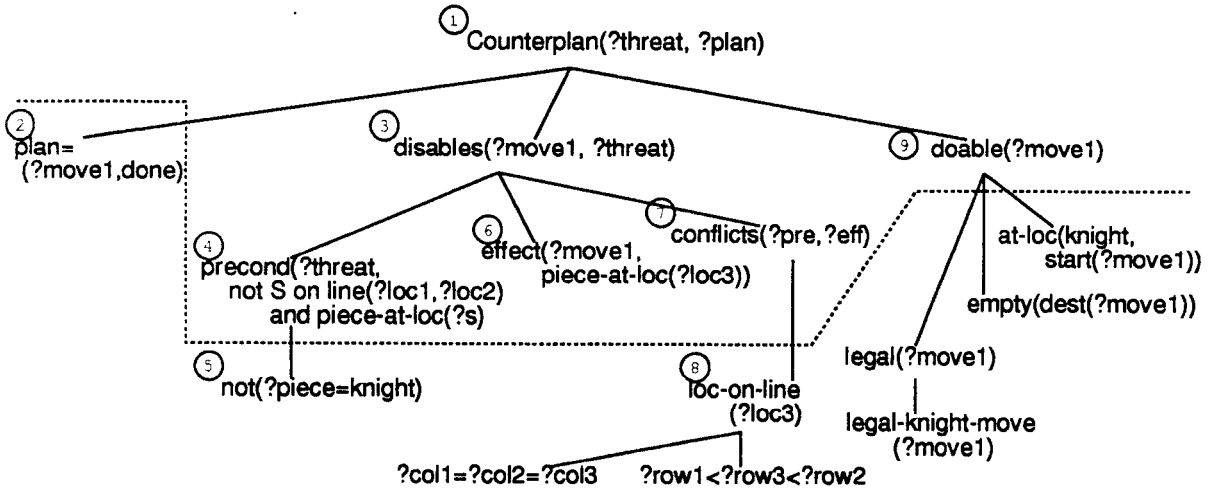


Figure 5.12: Explanation of desired counterplanner performance for interposition

```
(= ?counterplan (plan ?cp-move done))
```

Most equality statements of this sort are not added to the list of expressions, but are rather added to the constraints in the generalization bindings. In order to keep the explanations and learned rules more readable, however, CASTLE's EBL algorithm keeps explicit equality expressions in some situations. It should be clear to the reader that maintaining the equality expression and incorporating the constraints into the generalization bindings have equivalent effects on the resulting learned rule.

The EBL algorithm then turns its attention to the fact (in node **3**) that the opponent's move disabled the computer's plan:

```
(expl-disabled (move opponent move knight (loc 2 6) (loc 4 5))
   (move computer (capture bishop) queen (loc 7 5) (loc 1 5)))
```

The system matches this belief against the consequent of the rule used to make the inference (shown in figure 5.5), and determines that both of the moves can still be fully variablized, because the rule (as far as we know at this point) is true for all possible moves. The system then looks at the first of the three supporting beliefs (labeled **4**), that a precondition of the computer's move being carried out is that the line of attack is clear:

```
(expl-move-precond (move computer (capture bishop) queen
                                  (loc 7 5) (loc 1 5))
  (not (and (loc-on-line ?r3 ?c3 7 5 1 5)
            (at-loc ?other-piece ?other-player
                    (loc ?r3 ?c3)))))
```

When this is matched to the consequent of the rule that inferred it (figure 5.6), the EBL

system concludes that the belief would still hold for any type of move, so no constraint is placed on the move being considered.

The next supporting belief (labeled **5**), that the piece that the computer is trying to move is not a knight, is the next operational belief in the explanation that will be maintained in the generalizations:

```
(not (= queen knight))
```

As before, this belief imposes no constraints on the values of variables we have seen so far. The result of examining this expression is that its more general form, `(not (= ?piece knight))`, is added to the generalization being constructed by the EBL algorithm. The `precond` belief itself is *not* operational, and thus cannot be represented in the generalization. Instead, the variable `?pre` (referring to the move precondition in the rule for disablement, shown in figure 5.5) is bound to the precondition in the example, namely `(not (and (loc-on-line... )))`. The same occurs to the variable `?eff` when the EBL system handles the expression labeled **6**:

```
(expl-move-effect (move opponent move knight
                        (loc 2 6) (loc 4 5))
  (at-loc knight opponent (loc 4 5)))
```

This belief in and of itself doesn't impose any constraints on the generality of the variable values, since the rule in figure 5.7 holds true for any move. However, because this belief is in the explanation without any supporting beliefs, and it is not operational and thus cannot be included in the representation that is being constructed, EBL will fix the variables bound by the belief. As a consequence, the effect of the opponent's move being considered, `?eff`, will be bound to the `at-loc` effect.

EBL now turns its attention to the `conflicts` expression (labeled **7**), which will bring together the precondition and effect in a way that can be generalized usefully. The expression itself is:

```
(expl-conflicts
  (not (and (loc-on-line ?r3 ?c3 7 5 1 5)
            (at-loc ?other-piece ?other-player (loc ?r3 ?c3))))
  (at-loc knight opponent (loc 4 5)))
```

Since `conflicts` is not operational, this belief will not be included in the generalization being constructed by EBL, but the supporting belief, `loc-on-line` (labeled **8**), will be. More specifically, the forms of the precondition and effect have been fixed earlier in the EBL process, and a part of the precondition is used as a statement in the explanation which is itself operational. Because of this, the `conflicts` belief will always have the supporting belief `loc-on-line`. As a result, the `loc-on-line` belief:

```
(loc-on-line 4 5 7 5 1 5)
```

will be extracted from the precondition and generalized so that its component variables

match those in the general forms of the moves being considered, and will be added to the generalization being constructed by EBL. EBL will first examine the supporting beliefs to see if they impose any variable constraints, but in this case they do not.

So far, then, EBL has collected the following conjuncts for its generalization:

```
(= ?counterplan
   (plan (move ?other-player ?move-type ?other-piece
               ?other-loc (loc ?r3 ?c3)) done))

(not (= ?piece knight))
(loc-on-line ?r3 ?c3 ?r1 ?c1 ?r2 ?c2)
```

The last portion of the explanation, the `move-doable` belief labeled **9**, is handled in the same way. Since move-doable is itself not operational, the EBL algorithm collects the variable constraints imposed by the inference rule. Since the rule used to infer the `move-doable` belief was specialized for non-capturing moves, the variable `?move-type` is specialized to `move`. The following three sub-expressions are then added to EBL's generalization:

```
(move-legal-in-world (move ?other-player move ?other-piece
                           ?other-loc (loc ?r3 ?c3))
                     (world-at-time ?time))
(at-loc ?other-player ?other-piece ?other-loc)
(not (at-loc ?any-player ?any-piece (loc ?r3 ?c3)))
```

One problem with CASTLE's handling of this last portion of the explanation is that it should realize that the target location of the interposing move would be empty, because a necessary precondition for the opponent's attack to be a threat in the first place is that the line of attack be clear. In fact, it is precisely this requirement that is being blocked by the counterplan. Ideally CASTLE should realize that the destination square of the counterplanning move is necessarily empty because it must be for the attack to be valid, not merely because the counterplanning move itself was `doable` in the example, and that the final generalized expression listed above is therefore unnecessary. An approach to this problem would entail that EBL use knowledge about facts that are necessarily true at the time the rule is to be invoked. One such piece of knowledge is that the counterplanning component is only invoked on attacks that have all their preconditions satisfied, and thus on moves whose line of attack is clear. The application of knowledge of this sort is similar to the problem of transforming expressions in time, discussed in section 5.6.1.

CASTLE must now assemble the generalized explanation leaves into an antecedent for the new counterplanning rule. Since these expressions are the leaves of an explanation for the component specification, CASTLE knows that the expressions logically entail the truth of a generalized form of the specification. CASTLE can thus bundle up the generalized explanation leaves and the generalized component specification to form a new rule for the component. The first step is to specialize the component invocation, as given in the specification, into a rule antecedent by applying the variable bindings generated by EBL [DeJong and Mooney, 1986]:

```
(def-brule cp-learned-1
  (counterplan learned-cp-meth-1 ?player                    To counterplan against
     (goal-capture ?piece3 (rc->loc ?r2 ?c2)                an opponent threat
       (move ?other-player (capture ?piece3) ?piece         against a piece
             (loc ?r1 ?c1) (loc ?r2 ?c2)))
    ?time ?counterplan)
<=
 (and (= ?counterplan                                       plan to make a move
        (plan (move ?player move ?other-piece
                    ?other-loc (loc ?r3 ?c3)) done))
      (not (= ?piece knight))                               If the piece isn't a knight
      (loc-on-line ?r3 ?c3 ?r1 ?c1 ?r2 ?c2)                 to a square on the line
      (move-legal-in-world                                  of attack
          (move ?player move ?other-piece                   if there's a piece that can
                ?other-loc (loc ?r3 ?c3))                   move to that square
          (world-at-time ?time))
      (at-loc ?player ?other-piece ?other-loc)              and the square is empty
      (not (at-loc ?any-player ?any-piece (loc ?r3 ?c3))) ))
```

Figure 5.13: The learned rule for *interposition*

```
(method-spec counterplan
  (counterplan ?method ?player
               (goal-capture ?piece3 (rc->loc ?r2 ?c2)
                   (move ?other-player (capture ?piece3) ?piece
                         (loc ?r1 ?c1) (loc ?r2 ?c2)))
    ?time ?counterplan))
```

The next step is to extract from the specification the consequent for the new rule. All of CASTLE's component specifications are similar in form to the `method-spec` statement of the counterplanning component. The first parameter of the `method-spec` belief is the name of the component, and the second parameter is the form of invocation of the component. CASTLE thus extracts the second parameter of the expression, which in our example is `(counterplan ?method ?player...)`, and uses it to form the consequent for the new rule. Lastly, the variable corresponding to the method name must be replaced with a new and unique method name. The resulting learned rule is shown in figure 5.13. With this rule added, CASTLE recognizes interposition defenses that the opponent can use, and also interposes pieces to defend against opponent attacks.

5.4 CASTLE's explanatory theory

One of the crucial steps in CASTLE's learning process is the explanation process. As with other explanation-based learning systems, the explanations which CASTLE constructs form

the basis for the new rules that are learned, and the learning process relies on this ability to generate explanations whenever possible. This necessitates having an explanatory theory that is capable of explaining what the learner needs explained.

We saw elements of CASTLE's explanatory theory in section 5.2.2, which discussed how CASTLE explained the desired counterplanner behavior in the interposition example. The theory included inferences such as:

- *Counterplanning means generating moves that can be made that disable a threat* (figure 5.3).
- *One move disables another if an effect of the first conflicts with a precondition of the second* (figure 5.5).
- *An empty line of attack is a precondition for all moves except those by knights* (figure 5.6).
- *A non-capturing move can be made if it is a legal move, and the piece is at the starting location, and the ending location is empty.*

What principles govern the development of such an explanatory theory? The primary directive—which derives from our approach of diagnosing and repairing the planner itself and not just the failed plan—is that the theory be a meta-level model of the planning process that can reason about and explain the behavior of the system itself [Minton, 1988a; Birnbaum *et al.*, 1990; Collins *et al.*, 1993]. Such an explanatory theory meshes with CASTLE's explicit representation of its planning processes and their justifications (as discussed in chapter 4).

5.4.1 Modeling at the component level

What type of vocabulary should an explanatory theory use to explain planner behavior? Like any computer program, a planner can be discussed at many levels, from abstract behavior to program behavior to hardware behavior. What level of explanation is ideal for learning to plan?

CASTLE's approach is to model decision-making as a collection of *cognitive tasks,* each of which is implemented by a component and possibly a set of subcomponents. In this way CASTLE's *architectural structure,* in terms of components, mirrors the *semantics* of the decision-making process. In other words, while CASTLE's implementation in terms of rules and program code may bear little relation to any abstract description of the decision-making process—or, more precisely, no more relation to decision-making than to any other computational process—its organization into components directly reflects the conceptual tasks that make up decision-making. By organizing the system's explanatory theory around components and their cognitive tasks, we are able to generate explanations that relate to the semantics of decision-making. Put a different way, CASTLE constructs explanations not of desired rule firing performance, or desired search engine performance (see, e.g., [Minton, 1988a; Keller, 1987]), but rather of desired performance of various decision-making tasks.

The same quest for semantically meaningful explanations guides the representation of the constructs referred to in the explanatory theory as well. CASTLE reasons about notions such as move preconditions and effects not in their being preconditions of rules in the system, but as semantic entities in their own right. In particular, CASTLE's explanatory theory can reason about how preconditions and effects relate to each other, and can do this reasoning directly.

5.4.2 Elements of the theory

What elements make up CASTLE's model of decision-making? The following are the types of constructs that CASTLE represents:

- The cognitive tasks involved in decision-making
- Actions that can be taken in the domain
- Recognizing and reasoning about domain concepts

The first of these, the representation of cognitive tasks, consists largely of a representation of CASTLE's set of components. As we discussed in chapter 3, each component has associated with it a representation of its invocation and its specification. The invocation gives the form of the queries that invoke the component, and the specification describes the computation that it performs. These two elements provide a bridge between the vocabulary of tasks and the vocabulary of planning in the domain, in that the invocation gives the way that the planner invokes the component, and the specification describes the computation of the component in terms of the domain.

For example, the specification of the counterplanning component says that it outputs plans that disable a given threat. *Disablement* is a notion in the domain that is represented explicitly in CASTLE's model, as we have seen earlier in this chapter. Other domain notions that are represented include enablement, consequences of moves, and satisfaction of goals. Each of these are represented in terms of other ideas that are present in the domain, such as preconditions, effects, turns, moves, and captures, and many of these notions are relevant in a broader class of domains.

By representing all of these notions explicitly, and by explicitly representing "bridges" between the different levels of description, CASTLE can reason about abstract decision-making tasks and explain them in terms of details of the domain. This enables CASTLE to learn rules that are tailored for various cognitive tasks, but are expressed efficiently in terms of the domain.

CASTLE's explanatory theory, which is given in full in appendix B, is summarized in figure 5.14. The system's inference engine uses these notions to construct explanations of the desired component behavior. As we discussed above, this vocabulary allows CASTLE to explain facts about task performance in terms of planning in the domain.

Move preconditions	Precondition/effect conflicts
At starting location	Expression and its negation
Empty or enemy destination	Conjunction and negated conjunct
Clear line of attack	Precondition/effect matches
No resulting king threat	Matching expressions
Move effects	Conjunction and matching conjunct
Piece at destination	Move relations
Target not at destination	Enablement
Piece not at starting location	Disablement

Figure 5.14: Notions in CASTLE's explanatory model

5.5 Determining operationality

The output of the EBL process must be in a form that can be used effectively. In particular, the learned concept must be *operational,* in that it must be able to be used effectively by the system's decision-making processes to solve the given tasks [Mostow, 1981; Mostow, 1983]. The notion of effectiveness here has most often been defined in terms of computational cost, in that a learned rule must be expressed in a way that can be applied efficiently. The notion of *operationality* in explanation-based learning is an attempt to capture this need for the learned concept to be in a form that can be effectively used in practice.

To understand this notion, it helps to view the concepts being learned by EBL as reformulations of the target concept being learned, in light of the training example which was explained. The only reason that the reformulation (the output of the EBL process) is worth having is that it is better suited for use in practice than the target concept itself. An assumption which is made by EBL is that not all propositions can be effectively used, because otherwise the system could simply use the propositions of the target concept and avoid the need to formulate other representations of the concepts [Mitchell *et al.*, 1986].

Applying this idea to our example, CASTLE could in principle perform counterplanning using the definition of counterplanning shown in figure 5.3, which uses the `disabled` predicate to reason about moves that disable other moves. If this could be used directly, there would be no need for more specialized rules about counterplanning, such as the rule learned for counterplanning by interposition. The problem with this, however, is that the `disabled` predicate is extremely inefficient for use in practice, because it reasons explicitly about such complex concepts as action preconditions and effects and their interactions, and this reasoning involves a great deal of search inference. Invoking this type of reasoning in the course of planning would be very impractical, so CASTLE instead uses specialized rules for counterplanning that are expressed in a vocabulary that can be used effectively to recognize moves that respond to threats. This difference in vocabulary, classifying the system's

predicates as those that can and cannot be used effectively in practice, is the EBL notion of operationality. Expressions that can be used in practice are "operational," and those which cannot are "non-operational." The *operationality criterion* indicates which expressions are operational.

5.5.1 Prevous approaches

The most straightforward approach to determining operationality is simply to associate with each of the system's predicates whether the predicate is operational or not [Mitchell *et al.*, 1986]. One problem with this approach is that it does not allow the system's operationality criterion to evolve as the state of the system's knowledge and cognitive abilities change [DeJong and Mooney, 1986; Keller, 1988a]. More importantly, however, this approach is not a *theory* of operationality, in that it gives no basis for deciding which predicates are operational and which are not. Many systems, including CASTLE, use this fixed operationality definition to determine the operationality of many predicates in practice (chiefly those whose methods of computation do not change over time), but the theoretical basis for determining operationality must lie elsewhere.

One approach to determining operationality is to say that an expression is operational if it can be evaluated using a limited amount of inference, such as through schema application [DeJong and Mooney, 1986; Rosenbloom and Laird, 1986]. This approach assumes that the goal of learning new concepts is efficiency [Keller, 1988b], and, given this assumption, attempts to bound the amount of time taken for using any learned concept. Another advantage of this approach is that as the system learns new schemata, previously learned concepts will immediately make use of them, whereas learned concepts that consider schema application to be non-operational will be specialized for the particular instance of the schema that applied in the example, and will thus be unable to incorporate new schemata of the same type [Segre, 1987].

Another approach to determining operationality for a learner whose goal is efficiency is to perform an empirical analysis of the consequences of adding the concept [Keller, 1988a; Minton, 1988a]. An operationality measure determined in this way can be *continuous*, in that the empirical measurements give a *degree* of operationality rather than a binary yes-or-no determination [Keller, 1988a]. Unfortunately such empirical measures are heavily dependent on the context in which the measurements took place. Additionally, it is difficult to apply such techniques to learners with goals other than efficiency, e.g., flexibility derived from learning many new methods for carrying out tasks, because it is very difficult to empirically measure the effect that a learned rule has on a system's overall utility.

Other approaches are necessary to define operationality in situations where efficiency is not the overall goal of learning. In general, an operationality criterion depends on the learner's performance task and the type of performance improvement desired [Keller, 1988b]. In the context of story understanding, it is crucial that an explanation of an anomalous event address the aspects of the anomaly that were not understood before learning [Leake, 1988]. For a real-world planner, an operationality criterion should attempt to insure the robustness of the learned plans, in order that they be most effectively used in unpredictable and uncertain environments [Segre, 1988].

5.5.2 Operationality in CASTLE

CASTLE approaches operationality in much the same way as described above, but with some enhancements. Most of CASTLE's predicates have associated with them an *operationality value,* which is a boolean value indicating whether the predicate can be included in learned rules, or else have boolean-valued functions associated with them. The determination is made roughly as follows:

- Structure access, arithmetic, etc.: Operational
- Looping and collection of values: Depends on operationality of sub-expression
- Components which perform explicit search: Non-operational
- Search-free components: Operational

As we noted earlier, operationality decisions involve tradeoffs between efficiency on the one hand, and generality and the ability to incorporate new knowledge on the other. Including more specific expressions in learned rules will in general result in rules that are more efficient, while including more general expressions will allow the learned rules to apply in more situations [Segre, 1987]. There are two additional factors that influence the tradeoff between efficiency and generality in CASTLE. Each of them provide additional bias in the direction of more general learned rules.

One of these factors is that a more general learned rule may provide an additional avenue to learning in the future. As an example, suppose CASTLE were learning a new rule for planning that included making sure that the opponent did not have a counterplan to a particular attack. It may well be more efficient to include the specific reasons why no such counterplan existed, because this can be formulated more efficiently by making assumptions about the situation that are known to be true from other expressions in the rule. If, however, CASTLE has the learned rule invoke the counterplanning component instead, the system may in the future be able to learn new counterplanning rules from failures of the new planning rule. If the learned rule was formulated using specific information instead of an invocation of the counterplanner, the repair to such a failure would fix the planning rule fault but not the more general counterplanner bug. Because of CASTLE's ability to trace failures through rules to find learning opportunities, it is beneficial to include component invocations instead of specific expressions tailored for the situation, because such invocations may provide new learning opportunities in the future. This provides CASTLE with an additional bias towards considering component invocations to be operational, allowing the system to achieve a form of *incremental learning* that would not otherwise be available. We will see examples of this in chapter 6.

A second factor that arises in CASTLE is that including expressions that invoke components (instead of simply including the component method that applied in the example) is more valuable when the expression is in the scope of a negation than when it is not. This is because an expression that is too specific within a negation will result in an incorrect rule, while an expression that is too specific but not within a negation will result in a rule that is

incomplete but not incorrect as far as it goes. To see this, first consider a learned planning rule that uses a counterplan. If the rule includes an expression that is specialized for a specific counterplan, and the counterplanning component is later augmented, the planning rule will miss plans that it would otherwise generate (using the new counterplans), but will not produce incorrect plans. If, on the other hand, the learned planning rule refers to the lack of a counterplan, and the counterplanning component is later augmented, the planning rule without an invocation of the counterplanning component may believe that no counterplan exists when in fact one does, and will thus produce incorrect plans. Because of this, CASTLE considers some component invocations to be operational when negated, even then not otherwise.

5.6 Integrating new rules into components

Several issues arise in bundling up the output of the EBL process into a new rule for CASTLE's architecture. This section discusses several such issues that have not yet been discussed in this chapter. Some of these issues are handled in CASTLE, while some remain open for future research.

5.6.1 Transforming propositions in time

In principle, the generalized leaves of the explanation constructed by CASTLE will together form a proper antecedent for the learned rule, because they imply the truth of the component specification. One problem, however, is that the proof leaves may require information that was available when the explanation was constructed but not when the rule is to be applied. In other words, the expressions generated by EBL refer to events that have already happened, while we need the expressions in the new rule to refer to events that may happen in the future. This difficulty arises with regard to any statements referring to the time interval between the desired rule application and the expectation failure.

Transforming any expressions that cannot be evaluated when the rule is applied is a form of operationalization that corresponds to restating the preconditions into the "description languages" of other EBL systems (e.g., [Minton, 1984]). In particular, the problem of transforming expressions into a form that can be evaluated at rule-application time is the inverse of the *retrospection* problem in diagnosis, i.e., the need to evaluate expressions being diagnosed in a way that uses all information available at the time of diagnosis, not just that which was available at the time the expression was originally evaluated (see section 4.4). Such *retrospective analysis* transforms an expression forward in time, from plan-time to diagnosis-time, in order to take full advantage of knowledge gained in the interval; later in the learning process the learner must transform expressions backwards in time, from explanation-time to plan-time, to avoid dependence on this same knowledge.

This problem does not arise in the *interposition* example, but it tends to arise when learned rules refer to more than one prospective move. Consider, for example, if CASTLE were to learn an offensive strategy rule for the *fork* strategy [Birnbaum *et al.*, 1990; Collins *et al.*, 1993] shown in figure 5.15. As we discussed in section 3.7.2, CASTLE's offensive

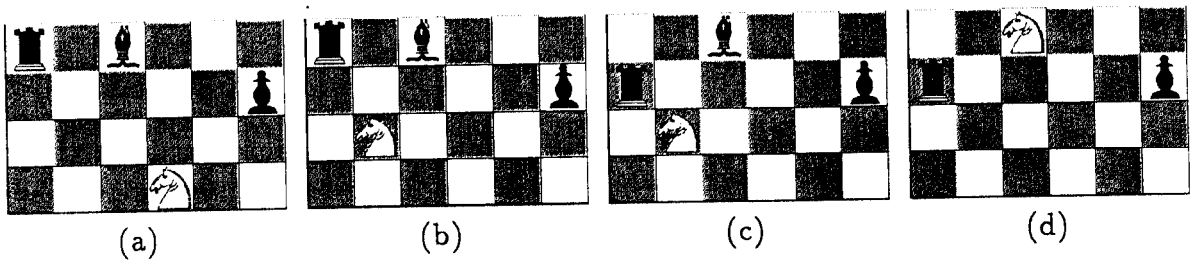


Figure 5.15: The *fork* strategy in use

strategy component consists of rules for scripts (schemata) of offensive strategies, which are general plans of attack that are applicable in many situations, and which when applicable are guaranteed to result in the satisfaction of a goal. In figure 5.15, the opponent moves its knight into a position that forks the computer's rook and bishop, so that one of them will certainly be captured in the subsequent turn. When the computer opts to save its rook, the bishop is lost. Following the expectation failure and diagnosis (which will be discussed in section 6.4), CASTLE constructs an explanation of why the move was a forced-outcome offensive strategy. This explanation, shown in figure 5.16, says roughly that the opponent's moves constituted such a strategy because *the opponent was able to move the knight to the intermediate location at time* T (first top-level branch), *was able to capture the bishop with the knight at time* T+1 (second branch), *was able to capture the rook with the knight at time* T+1 *as well* (fourth branch), *the first move enabled the other two* (third and fifth branches), *and the opponent had no move that would counterplan against both attacks* (final branch).

The generalized explanation leaves include the following:

1. The attacker's piece was in its initial position at time T.
2. The intermediate location was empty at time T.
3. The attacker could legally make the move at time T.
4. The attacker's piece was at the intermediate location at time T+1.
5. One of the defender's pieces was at its location at time T+1.
6. Another piece of the defender's was at its location at time T+1.
7. Both captures could legally be made at time T+1.
8. There was no double-counterplan at time T+1.

Several of these expressions suffer from the problem we discussed above, and refer to information that would not be available at the time the new rule is to be applied. In particular, since the rule is intended to be applied at time T, all the expressions that refer to events at time T+1 have this problem. These expressions are crucial in the explanation of

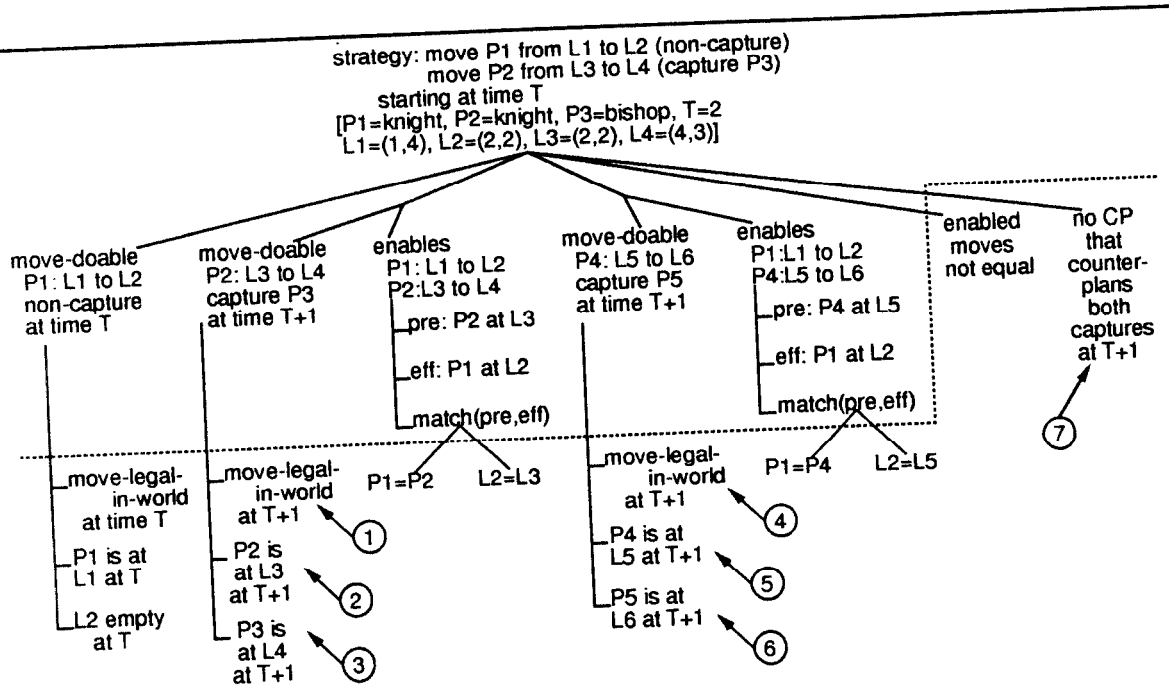


Figure 5.16: Explanation of desired strategy rule for the fork

the strategy, but must be modified to form rule conjuncts that can be evaluated when the rule is applied.

Our approach to handling these expressions is based on the observation that beliefs at time T+1 are true for one of two reasons: Either they were true at time T and have remained true, or they were established by the action at time T.[6] This gives us a straightforward procedure for transforming an expression back in time. If the statement was true at time T, it can be replaced with the same expression referring to time T, along with a statement (if necessary) of how its truth is known to be enforced between times T and T+1. If the expression was not true at time T, and was established by the action taken at time T, then it is replaced with an expression referring to those aspects of the action at time T that resulted in its being true at time T+1.

Returning to the expressions in the fork example, there are five that state facts about time T+1. Of these, the locations of the two attacked pieces, the legality of the attacks, and the lack of a double-counterplan are all true at time T+1 because they persisted from time T. The remaining expression, which states that the attacking piece was at the intermediate location, is true as a result of the action taken at time T, namely moving it from its starting location to establish the fork. Since the action at time T had no effect on the persistence of the beliefs from time T, the persisting expressions can simply be "moved back" to refer to time T. The other expression, that the attacking piece is at the intermediate location, is a

[6]This is the same insight used as the basis of non-linear approaches to planning, in which states are determined to be true either because of an immediately preceding action or because they were protected from an earlier time step. See, e.g., [Sacerdoti, 1975; Tate, 1977].

direct consequence of the action at time T, which is always going to be the move at time T whenever the strategy is executed. The upshot is that it can simply be removed from the antecedent.

In general, inference is required to show what additional assumptions, if any, are needed to insure the persistence or establishment of each expression. In our example, the persistence of the location of the attacked pieces from time T to T+1 is a consequence of the fact that it is the attacker's turn at time T, that there is no action taken on the board after time T until the defender's turn at time T+1, and that the locations of pieces on the board do not change unless actions are taken [Birnbaum *et al.*, 1990]. If CASTLE were operating in a domain in which these assumptions were not true (e.g., simultaneous-action games or games with non-deterministic effects, or in many non-game domains), the system would have to insert additional expressions in the learned rule to insure that it could depend on the persistence of location.

As another example, consider the inferences necessary for CASTLE to determine that it doesn't have to check that the attacking piece will be at the intermediate location at time T+1. First, it has to know that the piece being at that location is a result of the first move in the plan, which is considered in the `move-doable` branch of the explanation. Second, it has to know that if the plan is being carried out, then the move will definitely have been made at time T. Third, it has to know that the game dynamics insure that the piece can't change location between the end of time T and time T+1. If any of these assumptions were not true in the planner's domain,additional expressions would have to be added to the learned rule to insure the persistence.

Any inferentially-based approach to transforming learned expressions as we have been discussing is very complex, both in terms of the necessary computation and in terms of the knowledge needed for the inferences. Rather than carry out the full-blown approach, CASTLE employs a heuristic approximation intended to achieve the same results with much less knowledge-intensive inference. CASTLE's approximation is based on the fact that the system can examine the example situation itself to see if the fact persisted from a previous time or was recently enabled. For each expression that refers to the time interval after the rule is intended to be applied, CASTLE checks to see if the fact was true in the example at the earlier time. If it was true, CASTLE changes the expression to refer to the earlier time, and simply assumes that the fact will persist as it did in the example. If the expression was not true at the earlier time, CASTLE uses specialized rules to generate a replacement expression. These transformation rules are intended to compile knowledge about the conditions under which facts can be established.

In the fork example, all but one of the transformed expressions are seen to have been true at the time the strategy rules were applied, so they are simply changed to refer to the earlier time. The one expression which was not true earlier, the fact that the attacking piece should be at the intermediate location at the middle time step, is assumed to be the consequence of another step in the explanation, and is deleted. This assumption is embodied in one of CASTLE's transformation rules, which says that `at-loc` expressions that cannot be backed up should simply be removed. Other transformation rules often substitute expressions rather than simply deleting them. For example, one of CASTLE's transformation

rules specifies that a `move-to-make` expression, which refers to the move that a player will decide to make, should be replaced with an expression indicating that the move is a legal move and therefore *can* be made. Another transformation rule, which we will see in section 6.5, specifies that a `move-legal-in-world` expression, which checks if a move is legal and has a clear line of attack, should be replaced with a `move-legal` expression which only checks if the move is a legal one within the rules of chess. The latter transformation rule is in effect unpacking the `move-legal-in-world` expression into its component beliefs, and deleting the component beliefs (e.g., the clear line of attack) that cannot be backed up. These and other heuristic transformation rules achieve the same results as the transformation inference that we discussed earlier. It should be noted that these protection assumptions discussed above can in principle be included in the justification structures that CASTLE builds for learned rules, which would allow the system to refine its learned rules whenever a protection assumption is seen later to be incorrect.

This approach to transformation makes two assumptions. First, it assumes that facts that persisted in the example will *always* persist, and that no additional expressions are necessary to insure their persistence. Second, it assumes that there is a fixed mapping from expressions that need to be transformed to replacement expressions, and that this mapping holds in all examples and for all components and proof trees. The hope is that these assumptions will usually hold, and that any cases in which they do not hold can be handled by appropriately augmenting the component specifications to require the necessary additional explanatory inference. This has not been a problem in CASTLE's current implementation, however, and the transformation engine works sufficiently in all the examples that we will see in chapter 6. Implementation of the full-blown approach to expression transformation is a subject for future research.

5.6.2 Ordering sub-expressions in learned rules

The conjuncts in the antecedents of learned rules will by default be in the same order as they were generated in the explanation phase. While this will most often be a proper order, it will sometimes cause the learned rules to be less efficient than they could be. In the *interposition* example, for instance, the learned rule (shown again in figure 5.17) first determines the intermediate location on the line of attack, then determines a legal move to that line of attack, and then checks that the interposing piece is at the starting location for that legal move. When CASTLE is explaining the opponent's move as a counterplan, this ordering makes sense, because it first verifies that the counterplan moved a piece onto the line of attack, and only then verifies that the counterplan was a legal move from the previous location of the moved piece. This is sensible because the intermediate location is known in advance, as is the move being made, so no search is required to evaluate these expressions. When the rule is applied, however, this order will be extremely suboptimal, because it will generate all intermediate locations, and then all legal moves that move a piece to that location, and only then check if the move that was generated can be made by a piece currently on the board. Since the number of legal moves to the intermediate location (by any type of piece, from any location) is much larger than the number of pieces currently on the board, it would be more efficient to first evaluate the `at-loc` expression and then the `move-legal`.

```
(def-brule cp-learned-1
  (counterplan learned-cp-meth-1 ?player                    To counterplan against
     (goal-capture ?piece3 (rc->loc ?r2 ?c2)                an opponent threat
       (move ?other-player (capture ?piece3) ?piece         against a piece
             (loc ?r1 ?c1) (loc ?r2 ?c2)))
    ?time ?counterplan)
<=
 (and (= ?counterplan                                       plan to make a move
         (plan (move ?player move ?other-piece
                     ?other-loc (loc ?r3 ?c3)) done))
      (not (= ?piece knight))                               If the piece isn't a knight
      (loc-on-line ?r3 ?c3 ?r1 ?c1 ?r2 ?c2)                 to a square on the line
      (move-legal-in-world                                  of attack
          (move ?player move ?other-piece                   if there's a piece that can
                ?other-loc (loc ?r3 ?c3))                   move to that square
          (world-at-time ?time))
      (at-loc ?player ?other-piece ?other-loc)              and the square is empty
      (not (at-loc ?any-player ?any-piece (loc ?r3 ?c3))) ))
```

Figure 5.17: The learned rule for *interposition*

This particular problem can be handled by changing the order of the `at-loc` and `move-legal` expressions in the definition of `move-doable`, which was the source of the two expressions in the explanation. This is problematic, however, because it will make the `move-doable` rules less efficient at generating feasible moves in other situations. In general, an effective learner must be able to reason about the ordering of the expressions in rules that are learned [Mooney, 1988; Tambe and Rosenbloom, 1988b]. In the general case this requires reasoning about the expected number of ways each expression can be true, and how much each sub-expression constrains subsequent sub-expressions. This remains an open area of research in CASTLE.

5.6.3 Justifications for learned rules

Another step involved in integrating the new rule into CASTLE's planner is to tag the rule with a justification structure. CASTLE generates an appropriate justification for the new rule from the explanation which was used in learning [Mitchell *et al.*, 1986]. This new justification structure will thus show why CASTLE believes the rule to be true. An example of a justification for an application of the learned rule for interposition is shown in figure 5.18. This justification says that a move M is a counterplan because all of the antecedents of the interposition rule are true, and because of the unchecked assumption that the interposition rule is a correct counterplanning rule.[7] The interposition rule is believed to be valid because of the two antecedents in the explanation in figure 5.12, namely that the move disabled the threat and that the move was doable. This implication, that the belief in these two

[7] As discussed in section 2.6, dotted lines represent assumptions that justify the logical implication itself.

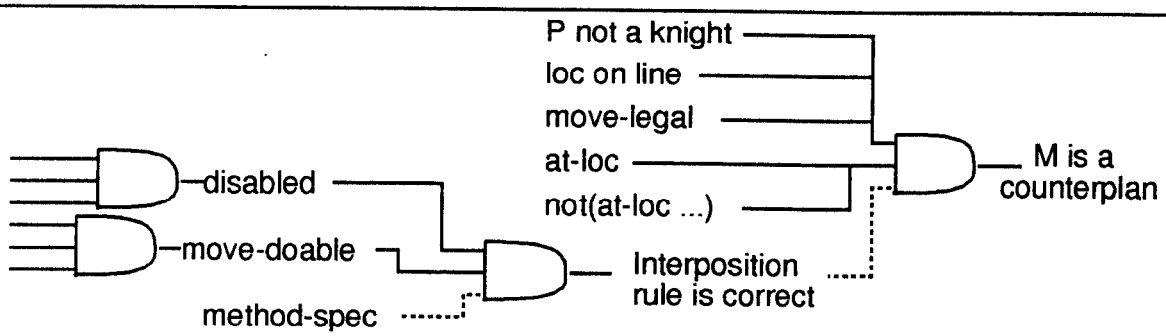


Figure 5.18: Justification for the *interposition* rule

antecedents implies the correctness of the rule, is itself justified by the specification for the counterplanning component.

This justification structure will facilitate learning in the same way the justifications for existing rules enabled CASTLE to perform the fault diagnosis discussed in chapter 4. In particular, the justification for the learned rule should enable the future learning that influenced the operationality criterion discussed in section 5.5. This justification structure can also be used for maintaining correctness when underlying assumptions change [deKleer *et al.*, 1977; Doyle, 1979; McDermott, 1989].

5.7 Other repair types

So far we have discussed only one type of fault repair, namely constructing a new rule to augment an incomplete component rule set. There are, however, times that other types of learning must take place. We now turn our attention briefly to these other types of learning. We will then discuss the *propagation of repairs* in section 5.7.3, in which one repair leads CASTLE to make other related repairs as well.

5.7.1 Learning or forgetting beliefs

The simplest type of repair to make is to add or remove beliefs from CASTLE's memory. If CASTLE discovered during diagnosis that a proposition that it believed to be true is in fact not true, and this faulty belief does not itself have a justification that will lead to more general repairs, one option is simply to remove the belief from memory. As we discussed in section 4.7, however, this is an extremely rare type of fault, because most of the beliefs in CASTLE's memory will have justification structures indicating a more basic fault, such as a faulty rule used to infer the belief. If, however, CASTLE were to periodically "forget" justifications for beliefs in its database in order to reduce memory requirements, it could be the case that the only repair available is to remove a faulty belief from memory.

A more common memory-related fault is for CASTLE to be *missing* a belief in memory, in which case a negation may be determined to be faulty. Since negations have no justification other than the lack of a matching belief in memory, it can easily be the case that a negation

is the most basic fault generated by the diagnosis engine. Whenever this happens CASTLE can repair the fault by asserting the missing belief into memory.

5.7.2 Masking or splitting rules

Another type of repair that can be made is to repair a rule that is seen to be faulty. The simplest form of this is to add a new conjunct to a rule's antecedent, in order to limit the scope of applicability of the rule and thereby stop it from being applied in an inappropriate situation [Krulwich *et al.*, 1989]. This can be done whenever CASTLE finds that a rule should not have been applied in a certain situation. Adding a new conjunct to the rule's antecedent ensures that the rule will not be apply in similar future situations.

A more complicated form of this type of repair is when a rule should have produced a *different* result in a given situation. In this case the rule must be split into *cases,* in which one will produce the same result as the old rule, and the other will produce the new result. Generating this repair requires that CASTLE construct an explanation of when the new result is wanted instead of the old result. The new rule can then be constructed in the same way that new rules were constructed earlier in this chapter, and the old rule can be masked so as not to apply in the given situation.

5.7.3 Propagating forward failed assumptions

Until now we have been discussing fault repair as a process of generating a single modification to CASTLE's decision-making architecture. Sometimes, however, a change that is made to one part of CASTLE's architecture necessitates making another change elsewhere as well. This is achieved by *propagating* a fault repair forward through the faulted justification structure to determine other structures that should be repaired.

Suppose, for instance, that CASTLE found a fault in a component specification, which is represented as a `method-spec` rule. After repairing the specification rule itself, CASTLE would have to propagate the repair forward to all other rules that relied on the truth of the specification, which may include all the rules in the component. As an example of this, suppose CASTLE realized that its specification for the counterplanning component was incomplete, and should also require that the counterplan not put the player in a worse situation that would exist if the threat were carried out. This repair would then have to be propagated to all the counterplanning rules to see if they must be repaired as a consequence of this change in the specification. The rule for counterplanning by interposition, for example, would have to be modified to check that the interposing piece cannot simply be captured by the opponent. This requires one of several conditions to be true: either the interposing piece is less valuable than the threatened piece, or it is less valuable than the attacking piece and it is guarded in its new location. This second condition was true in our example, as shown in figure 5.4, in which a pawn guards the knight from capture by the queen by ensuring that the queen could be captured if it were to capture the knight.

Repair propagation of this sort is an open area of research that has not been handled sufficiently in CASTLE. While CASTLE can handle propagating changes to a component specification, it has a harder time propagating changes to other rules or beliefs, because

it is harder to determine the way in which the possibly implicated construct relies on the repaired one. This is an open area for future research which is related to the general problem of handling faults that require multiple repairs [Freed, 1991].

5.8 What we've seen

This chapter has presented CASTLE's repair module, which formulates modifications to the system's decision-making processes in response to failures. The repair module constructs its repair by reasoning about *why* the faulty behavior took place and why the desired behavior wasn't achieved.

One of the primary goals of CASTLE's repair module is to approach learning as the application of self-knowledge. This self-knowledge includes the following:

- Specifications of the system's components and the tasks they perform
- An explanatory theory of planner behavior
- A model of the relationship between planner constructs and events in the world
- Information about correct formulation of new rules

The most common type of repair that CASTLE makes is to construct a new rule to augment an incomplete component rule set. There are, of course, drawbacks to CASTLE's approach to repair. One is that it is very difficult to learn knowledge that spans several components. This is because the system assumes that faults and repairs can be isolated to particular, individual components. While this buys a lot of leverage in terms of being able to apply knowledge about tasks and components, it does limit what can be learned, in particular about component interactions [Freed *et al.*, 1992].

5.8.1 The technical discussion of CASTLE

This chapter concludes the technical discussion of CASTLE's modules. The system's decision-making architecture (chapter 3), diagnosis engine (chapter 4), and repair module (chapter 5) together form CASTLE's learning model. This learning process, shown in figure 5.19, enables CASTLE to actively learn from failures.

The remainder of this thesis will demonstrate the utility and generality of this approach to learning planning knowledge. Chapter 6 shows examples of CASTLE learning a variety of concepts, and demonstrates the system's ability to repair any of its decision-making components. Chapter 7 then discusses how CASTLE's models can be extended to enable even more complex learning behavior than has been implemented. Chapter 8 discusses why CASTLE's approach to learning to plan is an improvement over previous approaches.

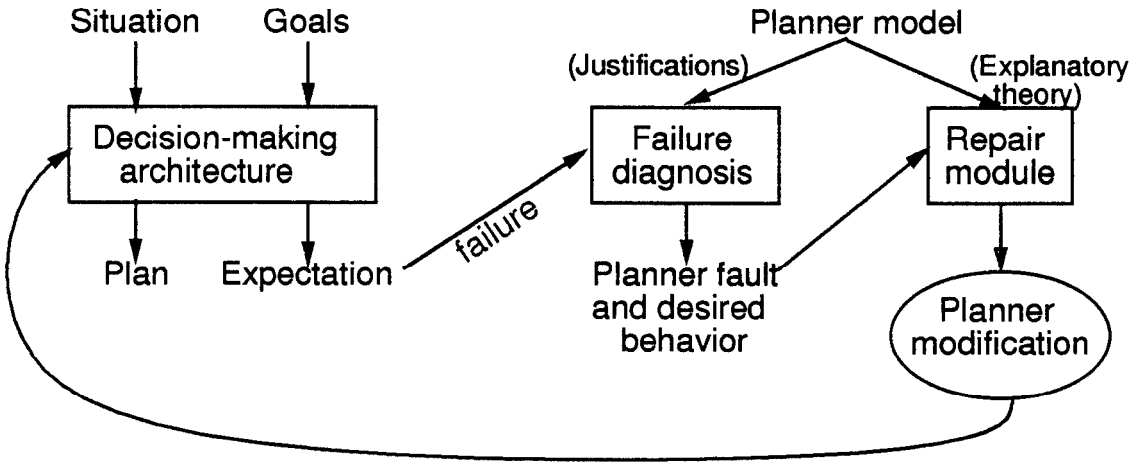


Figure 5.19: The learning process in CASTLE

Chapter 6

CASTLE in operation: Detailed examples

The previous three chapters have described the theory of learning to plan that is embodied in CASTLE. This theory enables CASTLE to learn planning knowledge from failures, and in principle should allow other systems, based on other decision-making architectures and operating in other domains, to learn in a similar manner.

Ideally, such a theory should be a recipe for designing a learning system: Take the approach to decision-making that you wish to use, add the ingredients of the learning theory, and you get a system that learns from the failures of your planner in your domain. It probably comes as no surprise that CASTLE's learning theory falls short of constraining system design enough to make this "recipe" approach a reality. Little formal guidance is provided as to the level at which to decompose an architecture into components, for example, or to the design of vocabulary for modeling decision-making.[1] Even without a complete set of such constraints from the theory, however, CASTLE does successfully embody useful approaches to these problems. This chapter will describe these approaches, and attempt to convey a feel for how such problems can be handled in general, through a description of the suite of examples that have been implemented in CASTLE. The examples illustrate an approach to decomposing a decision-making architecture and representing planning knowledge which is uniform enough to permit synergistic interactions among the representations used in each example. Chapter 7 will then present analyses of examples that have not been implemented, to convey how this approach can be extended.

Figure 6.1 describes the set of examples that we will be discussing. The examples listed as implemented can be run successfully in CASTLE as it currently stands, and are described in this chapter. The examples listed as extensions are more complex, and require augmenting CASTLE's models of decision-making or explanation. Although these examples go beyond what is currently implemented, they can be handled using the same approaches to diagnosis and learning that we have already discussed. These examples are discussed in chapter 7.

It should also be noted that in many cases, the details of the following examples—

[1] To be fair, of course, similar issues of representation and design of rules or operators arise in most AI theories.

	Example	Component
Implemented:	Interposition	Counterplanning
	Run away	Counterplanning
	Counterattack	Counterplanning
	Discovered attacks	Detection focusing
	En-passant	Threat detection
	Fork	Forced outcome
	Simultaneous attacks	Forced outcome
	Pin	Option limit
	Boxing in	Option limit
Extensions:	Buying time	Counterplanning
	Pawn line defense	Option limit
	Pin	Forced outcome
	Sacrifice	Forced outcome
	Midboard domination	Option limit
	Pawn line defense	Option limit

Figure 6.1: Learning examples: Implemented and extensions

including what is learned, the rules used in decision-making, the inferences involved in explaining the failures and desired behavior, and so on—reflect only one of the possible ways that that example could be implemented in CASTLE. More sophisticated analyses are feasible in many cases, and some of these are outlined briefly. The key point is that the analyses represent common approaches to the issues of design and representation in CASTLE's learning theory.

6.1 Simple counterplanning methods

In the *interposition* example in chapter 2, CASTLE started out knowing two methods of counterplanning, namely moving the attacked piece away from the attack and counterattacking the opponent's attacking piece. CASTLE can learn these two counterplanning rules fairly straightforwardly from situations similar to that of the interposition example. While it is of course implausible for a chess-playing agent not to have rules for these two counterplanning methods, these examples demonstrate CASTLE's ability to learn several methods for the same component.

6.1.1 Learning to run away

Consider CASTLE playing without a rule for *running away* from an attack, in the situation shown in figure 6.2(a). In its initial decision-making, the system invokes the planning rule

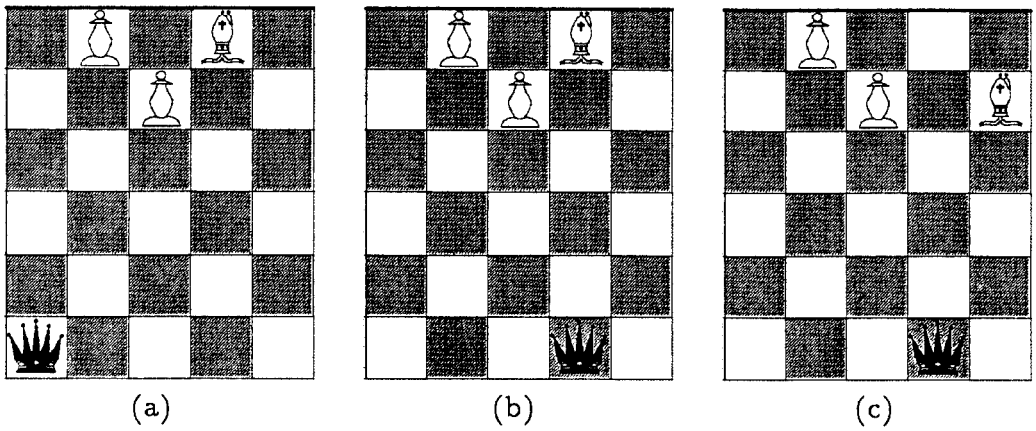


Figure 6.2: *Running away:* Computer (black) to move

from the interposition example (shown in figure 3.33 on page 55), which says roughly: *to generate a plan, determine a piece which can move to a new location, and an opponent's piece which can be captured by the attacker from the new location, such that the opponent has no counterplan, and plan to move the piece to the new location and then capture the target.* As we discussed previously, the planning rule tests whether the opponent will have a counterplan by testing whether any of its counterplanning rules apply, and assumes that if none apply then the opponent will not have a counterplan. If CASTLE lacks a rule for counterplanning by running away, it will not see that the opponent can counterplan against its attack by moving the bishop out of the way. CASTLE will therefore make the attack shown in figure 6.2(b). CASTLE's expectation of a successful subsequent attack then fails, and the diagnosis engine is invoked.

The diagnosis engine traverses the justification structure for the faulty expectation, which is shown in figure 6.3. This justification structure is the same as in the *interposition* example, except that the belief that no counterplanning rule applied will be justified by reference to the rules for counterattacking and interposition, with which CASTLE is equipped for this

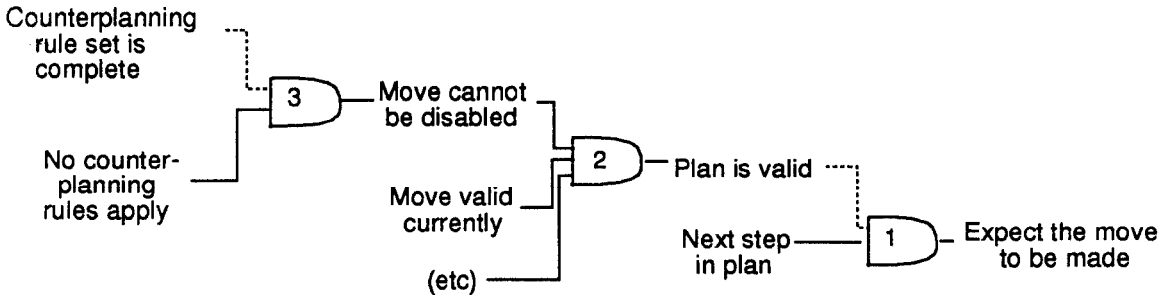


Figure 6.3: Justification for the failed expectation

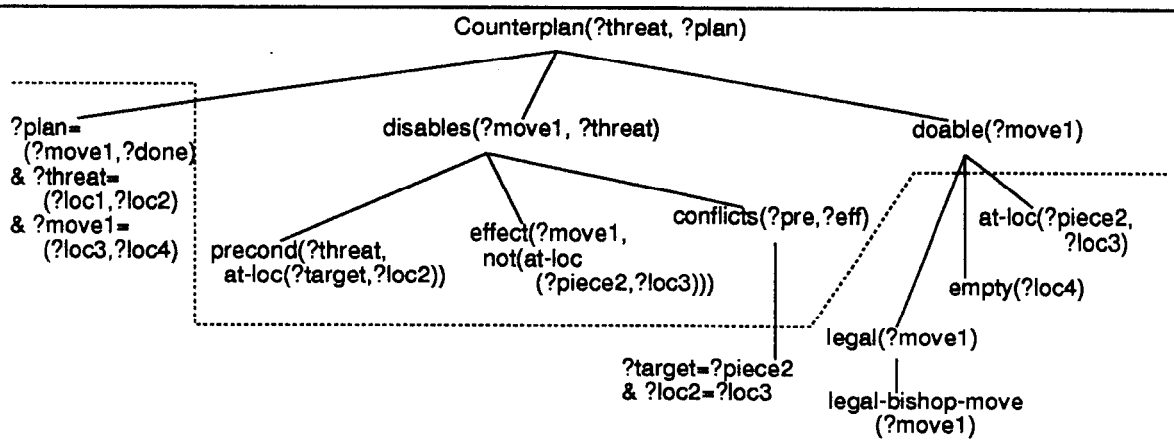


Figure 6.4: Explanation of desired *run away* counterplanner performance

example, and not for running away. The diagnosis process will be exactly the same as the diagnosis in the interposition example, as described in chapter 4. The diagnosis engine will conclude that the fault is a lack of a counterplanning rule that would have generated the opponent's move of his bishop in figure 6.2(c), and this fault will be passed to CASTLE's repair module.

The repair module responds to the fault by attempting to construct a new counterplanning rule. After retrieving the specification for the counterplanning component, CASTLE constructs an explanation of the opponent's counterplan. This explanation, shown in figure 6.4, is similar to the explanation in the interposition example (discussed in section 5.2.2), but with different action preconditions and effects involved in the disablement of the computer's plan. In the *run away* example, the disabled precondition of CASTLE's plan is that the target be at the destination location of the attack, and the effect of the opponent's move is that the bishop is no longer there. The explanatory inference rules used

```
(def-brule expl2c
  (expl-move-precond (move ?player (capture ?target)      A precondition of a
                           ?piece ?loc1 ?loc2)            capture is that the
     ?time (at-loc ?opponent ?target ?loc2 ?time))       target is at the location
 <= (= ?opponent (player-opponent ?player)))             being attacked

(def-brule expl3b
  (expl-move-effect (move ?player ?move-type              An effect of a move
                          ?piece ?loc1 ?loc2)             is that the piece is no
    ?time                                                 longer at the starting
    (not-true (at-loc ?player ?piece ?loc1 (1+ ?time)))) location
 <= ())
```

Figure 6.5: Explanation rules in the *run away* example

```
root: (expl-cp-spec (counterplan ?cp-meth ?player
                         (move ?player (capture ?target) ?piece
                               ?loc1 (loc ?r1 ?c1))
                         ?time ?cp))
leaf: (= ?cp (plan (move ?opponent move-move ?target
                          (loc ?r1 ?c1) (loc ?r2 ?c2))
                   done))
leaf: (= ?opponent (player-opponent ?player))
leaf: (at-loc ?opponent ?target (loc ?r1 ?c1) ?time)
leaf: (move-legal-in-world (move ?opponent move ?target
                                 (loc ?r1 ?c1) (loc ?r2 ?c2))
                           (world-at-time ?time))
leaf: (not (at-loc ?other-player ?other-piece (loc ?r2 ?c2) ?time))
```

Figure 6.6: Generalized leaves in the *run away* example

in constructing the explanation of the disablement are shown in figure 6.5.

The repair module then invokes explanation-based learning to form the new counterplanning rule. EBL returns the generalized root and leaves of the explanation shown in figure 6.6. These form a new counterplanning rule that says roughly:

TO COMPUTE: A counterplan to a threat of capture

DETERMINE: A plan to move the targeted piece
and a square to which it can legally move
and which is not occupied

If we compare these leaves with the hand-crafted rule for running away that we saw in figure 3.27, there are three differences between them reflecting three problems with the learned rule:

1. It unnecessarily checks the location of the targeted piece
2. It requires that the destination be empty, rather than allowing a capture
3. It might move the piece to an unsafe location

The first of these problems was discussed previously in section 5.3.3. CASTLE's explanation algorithm is unable to notice that the `at-loc` leaf, which was used to infer that the counterplan was a `doable` move, is unnecessary because the fact that the piece is at the particular location follows from the counterplanner's having been invoked with the given target.[2] As we discussed, it is an open research problem to extend EBL to detect such unnecessary explanation leaves.

[2]The `at-loc` expression furthermore hurts CASTLE's ability to use the learned rule in hypothetical situations, such as in the planning rule in figure 3.33, without adding additional vocabulary for "imagining" hypothetical situations.

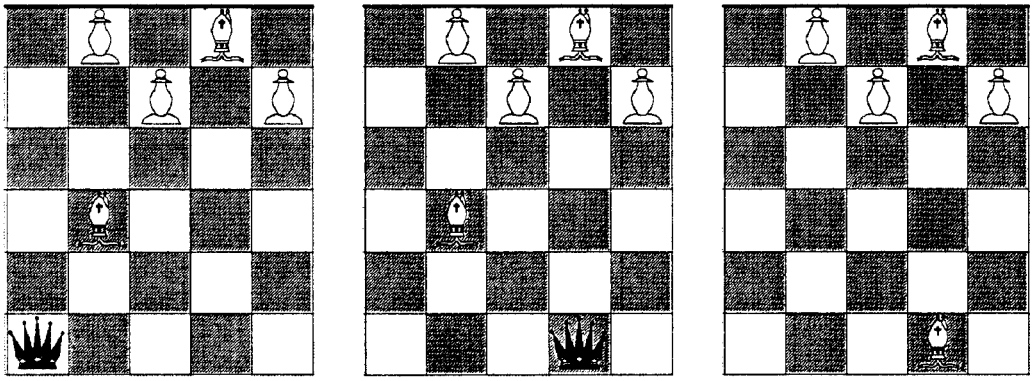


Figure 6.7: *Counterattack* example: Computer (black) to move

The second problem, that the `(not (at-loc ...))` leaf requires that the targeted piece be moved to an empty square and not capture a piece in the process, is a consequence of the fact that the `move-doable` predicate is not considered operational. We can see in figure 6.4 that if `move-doable` were operational, it would obviate the need to specify the type of move being made in the course of running away. This illustrates the tradeoffs involved in establishing operationality criteria that we discussed in section 5.5. In general, the more predicates that are operational, the more general the resulting rules will be, but the less efficient the computation will be. Simply listing `move-doable` as operational solves this problem in our example and in others.

The third problem, that the learned rule might move the targeted piece to an unsafe location, is a more serious one. This is a consequence of the fact that the counterplanning component has been specified as generating moves that disable threats, without any reference to the subsequent consequences of the moves being proposed. A similar fault arose in the interposition example, in that the learned rule doesn't prevent interposing a piece that is more valuable than the threatened piece, and in such a way that it can be taken. The solution to this problem involves extending CASTLE's explanatory vocabulary to include the notion of "improved outcomes," as we will discuss in chapter 7.

6.1.2 Learning to counterattack

CASTLE's other simple counterplanning rule is to capture the attacking opponent piece. Consider how this could be learned from the sequence of moves in figure 6.7. If CASTLE lacked a counterplanning rule for *counterattacking,* its planner would generate the same plan that we've seen in previous examples, moving the queen over and subsequently capturing one of the bishops. Because it lacked a counterplanning rule, it would not realize that the opponent could respond by capturing the queen with its other bishop.[3]

[3]As with the *running away* example, this appears to be a possibility that even beginning chess players

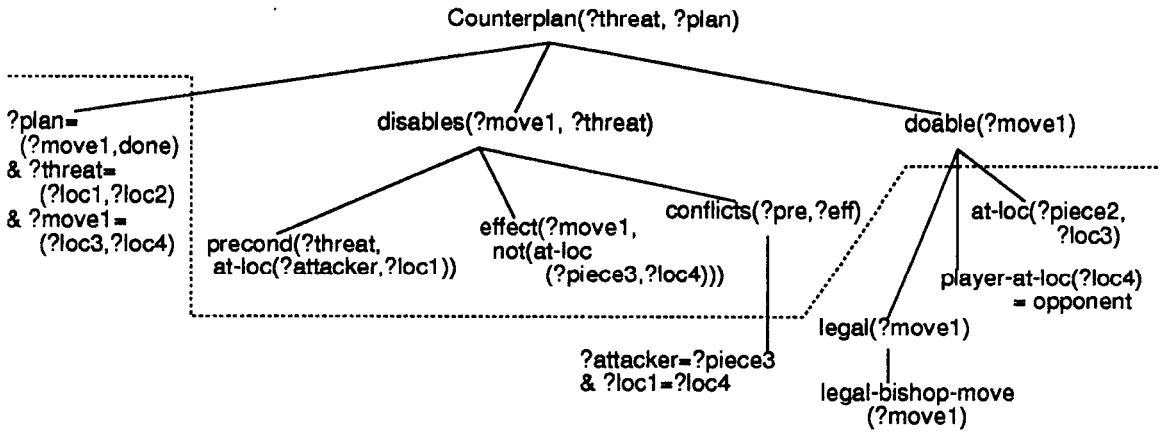


Figure 6.8: Explanation of desired *counterattack* counterplanner performance

After the opponent captures the computer's queen in figure 6.7(c), CASTLE must diagnose the failed expectation that its attack would succeed, and repair its planner. The diagnosis process proceeds in the same way as in the *interposition* and *running away* examples, using the same justification structure. The diagnosis engine traverses the justification structure shown in figure 6.3, and determines that the fault is that the counterplanning component lacked a rule for the opponent's response. The repair module then retrieves the specification for the counterplanning component and generates the explanation shown in figure 6.8. This explanation differs from the ones we've seen previously in the way in which the attack is disabled. The disabled precondition in the *counterattack* example is that the queen should be at its starting location, and this precondition is disabled by the queen being captured by the opponent's bishop. The explanation rules for the precondition and effect in this example are shown in figure 6.9. The leaves of the explanation are generalized to form a rule for counterplanning by counterattacking. Except that the learned rule once again checks unnecessarily that the attacker is at its starting location, the result is the counterattacking rule given in appendix A.

6.2 Learning to focus on discovered attacks

In section 3.4 we discussed CASTLE's *threat detection* component and the focus-of-attention mechanism implemented by the *detection focusing component*. The interaction between these two components is shown in figure 6.10. The focusing rules examine the events that have occurred and generate bindings that describe the areas of the board in which new threats may exist. The detection rules then search the current situation for new threats, within the constraints imposed by the focusing bindings. The new threats that are found are added to the set of previously-existing threats, and in this way the system incrementally maintains a

would generally see, but very young beginners might not.

```
(def-brule expl2b
  (expl-move-precond
      (move ?player ?move-type ?piece ?loc1 ?loc2)
      ?time (at-loc ?player ?piece ?loc1 ?time))
 <= ())

(def-brule expl3c
  (expl-move-effect (move ?player (capture ?target)
                          ?piece ?loc1 ?loc2) ?time
     (not-true (at-loc ?opponent ?target
                       ?loc2 (1+ ?time))))
 <= (= ?opponent (player-opponent ?player)))
```

A precondition of a move is that the piece be at the starting location

An effect of a capture is that the target is no longer at its old location

Figure 6.9: Explanation rules in the *counterattack* example

set of active threats.

Suppose that CASTLE had an incomplete set of detection focusing rules, in particular that it knew about threats being enabled either to or from the new location of a piece that has just been moved (as we saw in figure 3.14 on page 41), but did not know that threats could be enabled through *discovered attacks*, that is, by moving a third piece out of the line of attack [Collins *et al.*, 1991a; Collins *et al.*, 1991b]. Consider now the situation shown in figure 6.11(a), in which the opponent (playing white) advances its pawn and thereby enables an attack by its bishop on the computer's rook. When the system updates its set of active threats and opportunities, its threat focusing rules will enable it to detect its own ability to attack the opponent's pawn, but it will not detect the threat to its rook. As a result, in figure 6.11(b), the computer will capture the opponent's pawn instead of rescuing its own rook, and it will expect that the opponent's response will be to execute the attack which it believes to the only one available, namely to capture the computer's pawn. When the opponent captures the computer's rook in figure 6.11(c), the system has the task of diagnosing and learning from its failure to detect the threat which the opponent executed.

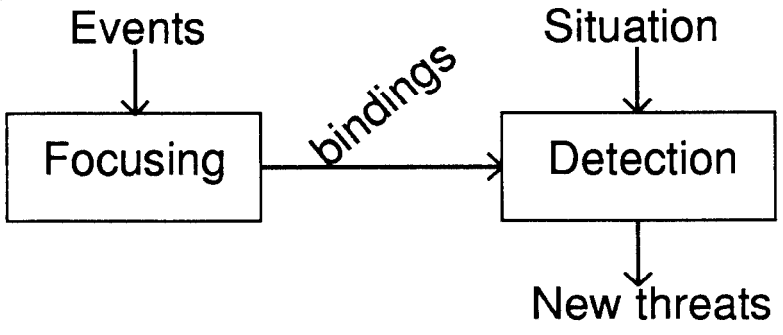


Figure 6.10: Incremental threat detection

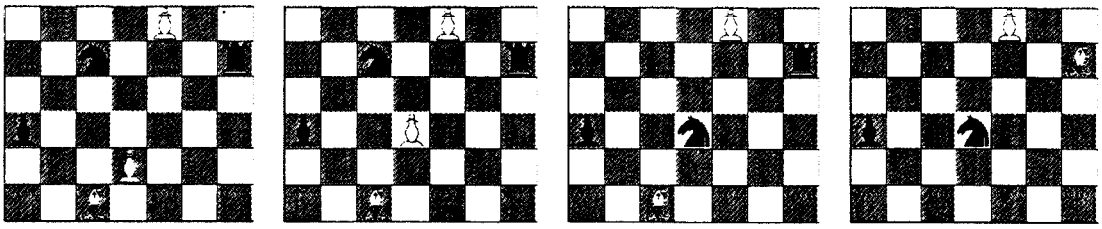


Figure 6.11: *Discovered attacks* example: Opponent (white) to move

The system then invokes its diagnosis engine to determine the component which is at fault, by examining the justification structure shown in figure 6.12. This leads it to conclude that the failure resulted from having an incomplete set of detection focusing rules. It will then try to learn from the failure by constructing a new one.

Two complications arise in the course of this diagnosis process. The first is that the enablement of the discovered attack did not take place in the previous move, which was the computer's move, but rather in the move before that. This means that the failure does not lie in the most recent application of the system's threat detection focusing component, but in the previous one. CASTLE handles this as follows:[4] In figure 6.12, the **active threats** node corresponds to a separate belief for each specific time step, with the threats that the system thought to be active at that time. The diagnosis engine initially faults the belief in the set of threats active at time 2, which corresponds to the board in figure 6.11(c). The opponent's threat against the computer's rook was *not* newly enabled at that point, since it was available at time 1 (figure 6.11(b)). Therefore, in traversing the justification

[4]The approach presented is similar to the "layers of temporal granularities" in [Hamscher and Davis, 1984].

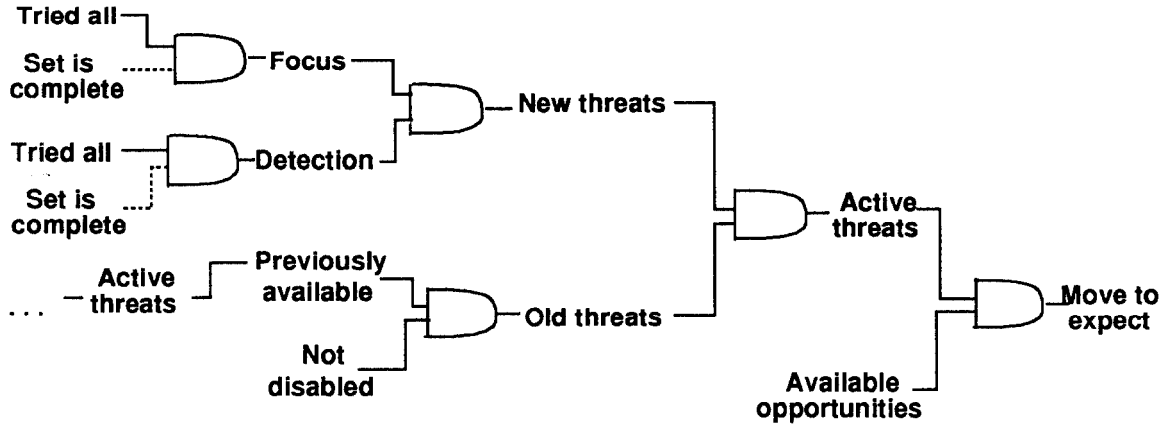


Figure 6.12: Justification for the failed prediction

structure, the diagnosis engine will see that its faulty **active threats** belief was due to a faulty belief about **old threats** that are still available, which was due to a mistaken belief about what threats were **previously available**. This belief in turn depends on the **active threats** belief from the previous time step, namely time 1 corresponding to figure 6.11(b). In other words, the **active threats** belief on the left side of figure 6.12 is the same as the **active threats** node on the right side, but for the previous time step, and it has the same justification supporting its belief. It is at this point that the opponent's attack was newly enabled, so the diagnosis engine diagnoses the faulty belief in the **new threats** enabled at that time step.

Another complication arises at this point. The diagnosis engine has the task of determining which of two sub-components are responsible for the failure: the *threat detection* component or the *detection focusing* component. In general, the task of distinguishing between them is one of *experimentation:* Would the detection component have been able to detect the threat had it been supplied with different bindings? Reasoning of this sort is made relatively easy by CASTLE's approach to inference and component invocation. Each of the two potentially-faulty components corresponds to a conjunct in the threat detection meta-rule, and each of these conjuncts can be queried independently as well as in tandem. Thus, to test the detection component's behavior in the absence of focus bindings, the diagnosis engine simply queries the threat detection component with the undetected capture. If the query succeeds, the detection component is in principle able to detect the threat, and is thus acquitted of the failure. If the query fails, the component is known to be faulty, and the supporting beliefs must be diagnosed. In the same way, the diagnosis engine tests the detection focusing component by making a query which invokes the focus component on the capture that was enabled. If the query succeeds, the focus component is known to be able to generate the capture as a possible threat, so it is free from blame. If the query fails, the focus component must be faulted. Using this method, alternative sub-components are tested and either faulted or acquitted in a reasonably straightforward fashion.

In our example, CASTLE's diagnosis engine tests the detection focusing component with the query `(focus ?f-method computer ?capture (world-at-time 1))`, with `?capture` bound to the capture which the opponent executed. Since neither of CASTLE's focusing rules can be used to focus on the opponent's capture, this query will fail, indicating that the focusing component is to blame. The diagnosis engine then examines the supporting beliefs for the detection focusing, and determines that the fault underlying the failure is that the detection focusing rule set is incomplete.

To repair the fault, the system retrieves a specification of the detection focusing component, which says that focus rules will indicate any moves which have been newly enabled. To learn from the failure, CASTLE uses the specification as a target concept to construct an explanation of what the focus rules should have computed. This explanation, which is shown in figure 6.13, says roughly that *the focus rules should have focussed on the bishop-rook attack,* because *the pawn move that was made enabled the bishop-rook capture,* because *a precondition to the rook capture (by a piece other than a knight) is that the line of attack be clear,* and *an effect of the pawn move was that the square was vacated,* and *the vacated square was on the line of attack.* CASTLE's EBL mechanism then constructs a rule

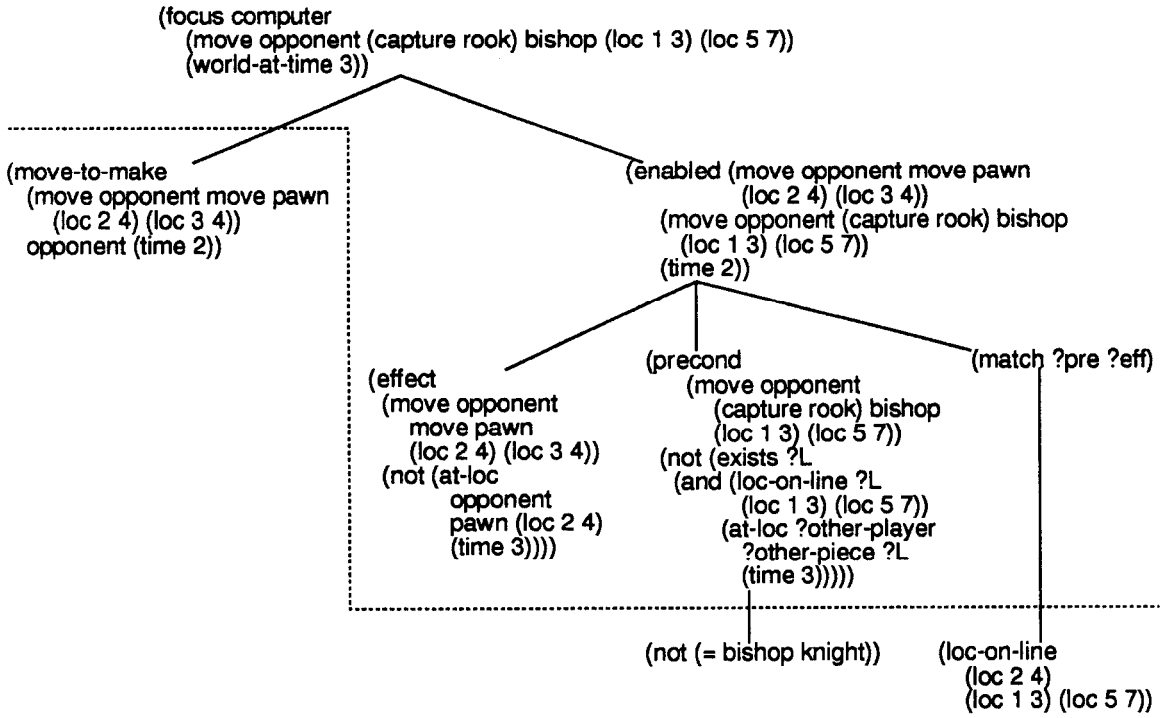


Figure 6.13: Explanation of desired detection focusing

from the proof tree leaves which will correctly focus on discovered attacks. This rule, which is shown in figure 6.14, says roughly:

TO COMPUTE: Areas in which moves may have been enabled

DETERMINE: Lines of attack through the most-recently vacated square for new moves by pieces other than knights

We tested this rule on a variety of examples of discovered attacks, shown in figure 6.15, and CASTLE consistantly learned the same new rule for detection focusing, and was able to use the rule learned in each example to correctly detect the enabled threats in the other examples.

One implementational note deserves mention at this point. In the other focus rules we have seen (figure 3.14 on page 41), the consequences of invoking the rules is to add constraints to the moves being considered. One rule, for example, says to focus on new attacks by the newly moved piece from its new location. When this rule is successfully invoked, the variables for the piece and starting location of the considered move are constrained to be the same values as the newly moved piece and its new location. Once this constraint is added, the focus rule has done its job for an entire class of possible new threats. The same is true for the focus rule that says to look for new threats against the newly moved piece in its new location. In both cases, the constraints are passed on to the detection mechanism through CASTLE's variable binding mechanism.

```
(def-brule learned-focus-method25
  (focus learned-focus-method25 ?player                    To find possibly
    (move ?player (capture ?taken-piece) ?taking-piece     enabled new threats
          (loc ?row1 ?col1) (loc ?row2 ?col2))             to focus on
    (world-at-time ?time2))
<=
  (and (move-to-make (move ?other-player move-move ?interm-piece
                           (loc ?r-interm ?c-interm)       See where the most
                           (loc ?r-other ?c-other))        recently moved
           ?player ?goal ?time1)                           piece used to be
       (not (= ?taking-piece knight))                      and, for non-knights,
       (loc-on-line ?r-interm ?c-interm                    find moves through
                    ?row1 ?col1 ?row2 ?col2) ))            the vacated square
```

Figure 6.14: Learned focus rule for *discovered attacks*

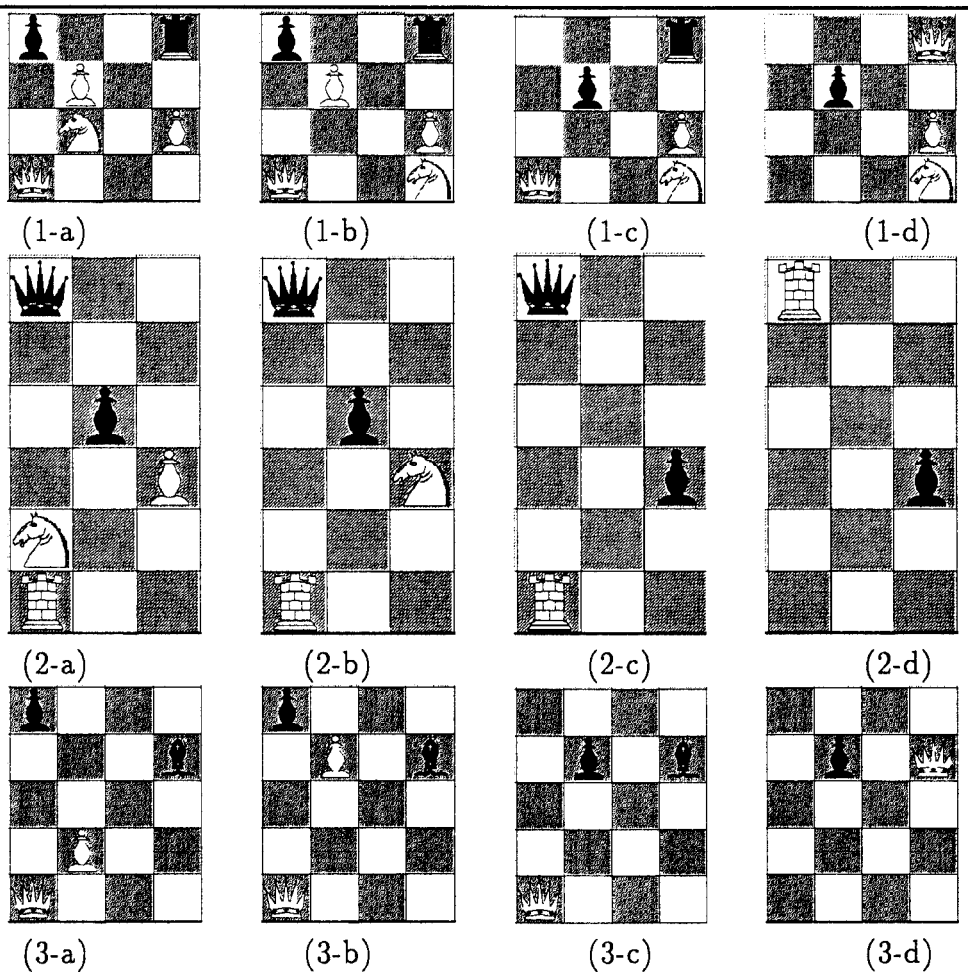


Figure 6.15: Variant examples of discovered attacks

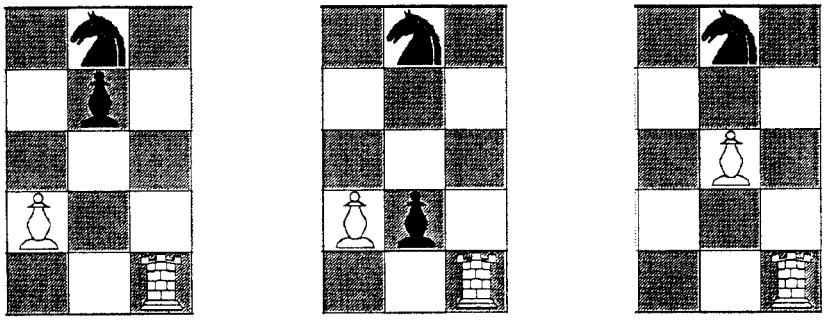
In the case of discovered attacks, however, the situation is not so simple, because there is no single variable constraint corresponding to the statement "the source and destination of the new attacks form a line through the vacated square." Since this doesn't correspond to a simple variable binding constraint, the discovered attacks rule must generate individual lines of attack one by one. This means that the threat detection rules must reenter the discovered attacks rule as many times as needed to enumerate the possibly opened lines of attack. This source of inefficiency could be obviated if CASTLE were able to maintain complex variable constraints, such as the colinearity of the three board locations in the discovered attacks rule. One approach is to represent these constraints explicitly in CASTLE's rule sets (as was mentioned briefly in footnote 2 on page 39). If CASTLE had an explicit representation of focus constraints, they could be generated in a single step by the focus rules and then applied by the detection rules later.

Another inefficiency in the rule CASTLE learns for discovered attacks arises with the conjunct in the antecedent that specifies that this focus rule should not focus on moves by knights. The reason for this is that knights do not require clear lines of attack, so that emptying a square will not enable a knight move in this fashion. We saw this same distinction in the interposition example, in which the learned counterplanning rule cannot be used to respond to knight threats. In both cases the basis for the distinction is the precondition rule (shown in figure 5.6 on page 92) which says that a clear line of attack is a precondition for all moves except those by knights.

This conjunct suffers from the same inefficiency that we discussed above, because CASTLE's inference engine cannot represent inequality constraints directly. The discovered attacks rule shown in figure 6.14 will therefore generate five times as many constraints as it should, one for each type of piece other than knights. This could in principle lead to improved performance, if the specification of the type of piece would constrain the lines of attack that are considered, but reasoning about which pieces can move along various lines of attack is by definition not carried out by the focusing component (and is rather left to the detection component). The upshot is that since the number of unnecessary constraints that would result from eliminating the conjunct is small (moves by two knights), and the savings in terms of inference engine time is fairly large, CASTLE removes this constraint in an *ad hoc* fashion when the system learns detection focusing rules. In general this problem is similar to the problem of ordering conjuncts, discussed in section 5.6.2, and remains an open problem.

6.3 Learning to detect en passant threats

Suppose that CASTLE has a set of threat detection rules that cannot detect *en passant* pawn captures [Birnbaum *et al.*, 1989]. *En passant* captures, which are unknown to many novice chess players, involve a pawn that has just moved two squares forward (as its first move) being captured by a second pawn moving diagonally into the square that the first pawn skipped. If CASTLE lacks a rule for such a threat, it may have a pawn which it expects to be safe from attack even though it is in fact susceptible to an *en passant* capture. An example of this is shown in figure 6.16, in which the system advances its pawn to threaten



(a) (b) (c)

Figure 6.16: *En passant* example: Computer (black) to move

the opponent's rook, and the pawn is subsequently captured *en passant.*

CASTLE now has a failure of its expectation that its pawn was safe, or, more precisely, that it had correctly computed a set of threats that the opponent could mount. To learn from the failure, CASTLE first diagnoses the failure of its expectation that the pawn was safe. The relevant justification structure is the same as the justification we saw in the *discovered attacks* example, shown in figure 6.12, which relates the system's beliefs about the currently available threats to its threat detection focusing and detection components. Here, however, in deciding which of the subcomponents to fault, CASTLE determines that the fault is not in the focusing rules, because the captured piece was the most recently moved piece, which was focused on by one of its focusing rules. Rather, the fault is in the detection rules, because even without any constraints from the focusing component, the threat would still not have been detected. CASTLE's diagnosis engine concludes that its set of threat detection rules must be augmented.

At this point, however, CASTLE is unable to construct an explanation of the new rule's behavior, because the fault relates to a rule of the game of which CASTLE is unaware. Learning from a failure of this type requires asking another agent (namely the human user) for the mechanics of *en passant* captures. This is implemented by a specification for the threat detection component that says roughly: *The detection component computes the ways that pieces can make captures that are within the rules of the game, as told to the computer by the user.* When CASTLE's deductive engine is invoked to explain why the move that the opponent made is an instance of this specification, it will conclude that it has to ask the user why the move was legal. The user's input is then transformed into the proper rule format.

6.4 Learning about the fork

We now turn our attention to more complex examples of CASTLE in action. Suppose we start CASTLE without any knowledge of the classic chess strategy, the *fork.* In the situation shown in figure 6.17, CASTLE is unaware in board (a) that the opponent can use one of his knights to fork the computer's bishop and rook, and so the system captures the opponent's

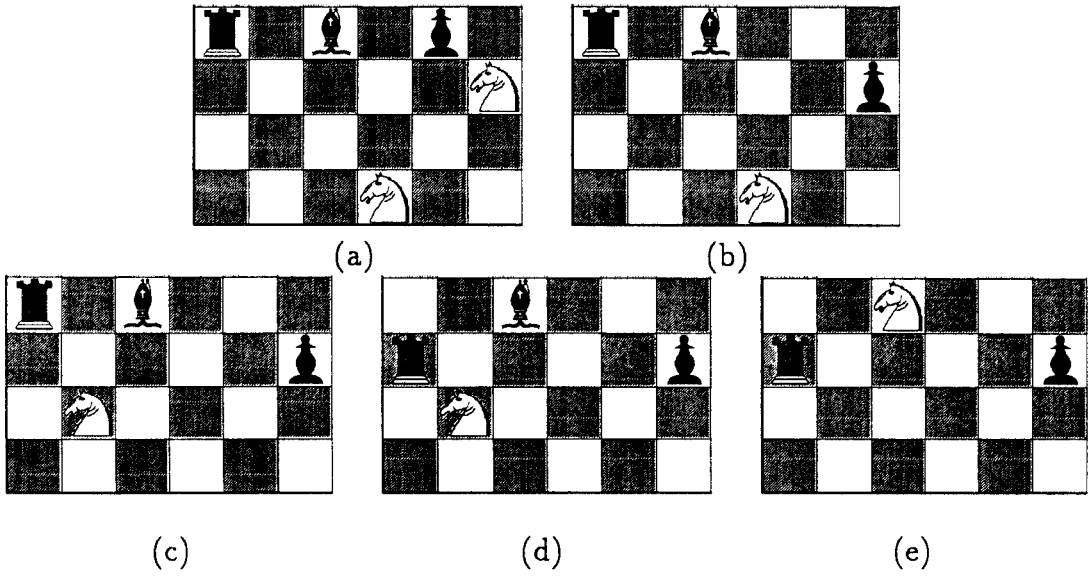


Figure 6.17: *Fork* example: Computer (black) to move

other knight with its pawn. When in board (b) the opponent applies the fork, resulting in board (c), the system has to choose which of its pieces to save. It decides to save the rook at the expense of the bishop (d), and the opponent proceeds to capture the bishop (e). CASTLE then has to find the fault in its decision-making mechanism that allowed it to get into such a situation, and learn how to avoid similar occurrences in the future. In particular, CASTLE must learn a rule that will enable it to recognize the fork early enough to avoid being trapped.

What expectation has failed in this example? We saw in chapter 3 that CASTLE responds to immediate threats using its **counterplanning** component, and assumes that threats will be detected early enough to be countered before they are carried out [Birnbaum *et al.*, 1990; Collins *et al.*, 1993]. Detection of threats that must be counterplanned against is handled by the **threat detection** component, which detects direct threats, as well as the **plan recognition** component, that detects more complex plans. CASTLE assumes that the combination of these components and its counterplanning component will be effective in countering all threats mounted by the opponent. We have previously called this the "firefighting" paradigm for dealing with threats: CASTLE assumes that threats can be detected and handled as and when they arise [Birnbaum *et al.*, 1990; Collins *et al.*, 1991a].

More precisely, CASTLE believes that it will be able to firefight successfully unless the opponent is able to initiate a strategy to force the system into a loss. The expectation that the system will be able to successfully firefight in figure 6.17(c) is thus based on three assumptions: The system will be able to detect any direct threats that will be enabled, it will be able to counterplan against these threats, and it does not think the opponent has a forced-outcome strategy to apply.

This assumption could thereby fail for three reasons: The threat detection mechanism

could fail to detect a direct threat (as in the discovered attacks example), the plan recognition component could fail to detect an opponent strategy, or the counterplanning component could fail to produce an adequate response.[5] In the case of the fork, CASTLE's counterplanning component is not at fault, because it is capable of generating counterplans to each of the opponent's threats individually, and because no counterplan existed that would handle both of the threats simultaneously. Rather, the problem is that CASTLE did not detect the possibility of the fork early enough: CASTLE should have known *before* the situation shown in in figure 6.17(a) that a problematic situation was coming up and should be avoided. In other words, the problem is with CASTLE's plan recognition component.

As we discussed in section 3.5, the relevant plan recognition method is one that uses CASTLE's *offensive strategy* component to recognize opponent plans. Offensive strategies are general sequences of moves, such as the *fork,* that result in an enemy piece being captured (or some other goal achievement). For CASTLE's firefighting paradigm to work properly in our example, the offensive strategy component would have to have had a rule to observe the opportunity to establish the fork. The lack of such a rule led to the failure. In response, then, CASTLE must learn a new offensive strategy rule that generates the *fork* as a possible plan when it is applicable.

6.4.1 Expecting to defend against threats

In order to monitor the success of the firefighting approach to handling threats, CASTLE maintains and monitors expectations regarding the threats that will exist in subsequent turns. The existence of threats that CASTLE is unable to handle indicates the system's inability to carry out the necessary firefighting. CASTLE posts an expectation specifically for this purpose, of the form:

```
(no-opps-at-time ?player ?time)
```

This expectation is posted whenever CASTLE believes that it has control of the threats available to the opponent. More specifically, after CASTLE makes a move, it checks to see if there are any opponent opportunities that it has for some reason not responded to. If there are not, CASTLE checks if its plan recognition rules can detect any offensive strategy that the opponent could use to capture a computer piece in a way that the computer can't prevent. In the absence of such a strategy, CASTLE assumes that at its next turn it will be able to handle all threats that the opponent has enabled in the interim move.

In our example, before the opponent's move in figure 6.17(a), CASTLE checks whether the opponent had any available threats, and determines that it did not. CASTLE then checks if its plan recognition component can determine any forced-move offensive strategy that would guarantee a capture, and, in the absence of a strategy rule for the fork, decides that the opponent has no strategy available. The system therefore executes an attack of its own, and assumes that on its next move it expects to handle any threats that the opponent may

[5]The proper functioning of these components is in fact not independent. For example, the **counterplanning** component assumes that the threats will be detected *in time* to be handled properly [Birnbaum *et al.*, 1990].

```
(def-brule recog-plan-2
  (recog-plan recog-strategy ?opp-player                    To recognize a possible
      (world-at-time ?time) ?plan ?goal)                    opponent plan
 <=
  (and (in-set ?strategy-meth strategy-meths)               Find a strategy method
       (strategy ?strategy-meth ?opp-player                 that gives a plan for the
          (world-at-time current-time) ?goal ?plan) ))      opponent
```

Figure 6.18: Recognizing opponent plans using strategy scripts

enable. Based on this assumption, CASTLE asserts an expectation that after its next turn (i.e., after figure 6.17(c)) there will be no threats from the opponent, that is, that on its next turn the system would be able to handle all existing threats and would not leave any open. This expectation fails in board (d), and CASTLE must diagnose and repair the fault in its decision-making mechanisms.

Consider the rules in CASTLE's planner that carried out this set of computations. In the situation shown in figure 6.17(a) the computer (playing black) must decide on a move to make. The first thing it does is invoke its plan recognition component to determine what potential threats the opponent has that should be addressed now. One such plan recognition rule, employing the system's offensive strategy component, is shown in figure 6.18. This rule says roughly:

TO COMPUTE: A plan the opponent can execute

DETERMINE: An offensive strategy script
which the opponent can currently apply

If CASTLE were equipped with a strategy rule for the fork strategy, it would recognize the opponent's ability to threaten its rook and bishop using one of his knights. After recognizing this potential plan, CASTLE could respond by moving one of the two pieces, or in general by disabling one of the two threats and thus removing the double-threat that is inherent to the fork strategy. In the absence of a fork strategy rule, however, CASTLE does not recognize the opponent's possible attack, and instead decides to capture the opponent's other knight with its pawn.

After executing this move, CASTLE's expectation generation mechanisms must determine whether to expect successful firefighting on the subsequent turn. Intuitively, CASTLE should expect that it will be able to handle all threats that arise on the next turn whenever:

1. CASTLE has handled all the pending opponent plans and goals on this turn

2. The opponent is not in the midst of executing an offensive strategy that forces CASTLE to a loss.

```
(def-brule exp-no-opp-plans
  (should-expect exp-no-opp-plans ?time                    Expect the opponent not
      (no-opps-at-time opponent (+ 2 ?time)))              to have opportunities
 <=
  (and (move-to-make ?move computer ?goal ?time)
       (no (and (active-goal opponent                      If there's no active
                 (goal-capture ?comp-piece ?comp-loc       goal of the opponent's
                               (move ?opp-move))
                 ?time)
            (not (counterplan ?cp-meth computer            which the recent move
                  (goal-capture ?comp-piece ?comp-loc      didn't counterplan
                                (move ?opp-move))
                  ?time (plan ?move done))) )) ))
```

Figure 6.19: Expecting to handle subsequent opponent threats

Rather than check again whether any offensive strategies are in motion, CASTLE relies on the fact that its plan recognition component has done this already, and (as we saw above) has posted an active opponent plan whenever the opponent has the opportunity to apply an offensive strategy. Because of this, CASTLE can at this point check to see that all active opponent plans have been countered, and if so, CASTLE expects that it will be able to successfully handle all threats on the subsequent turn. This expectation rule is shown in figure 6.19, and says roughly:

TO COMPUTE: An expectation of no opponent opportunities

DETERMINE: The move to be made by the computer
and that no opponent goals are left unhandled

An expectation of this type fails whenever the the computer fails to handle all active threats from the opponent. In other words, after the computer makes a move, if a `no-opps-at-time` expectation has been posted, it is determined to have failed if any opponent threats remain active. This rule detects the failure of the expectation in the fork example after the computer saves its rook, since the opponent still has the option of capturing the computer's bishop.

6.4.2 Diagnosing and learning from the failure

To diagnose the expectation failure, CASTLE traverses the justification shown in figure 6.20, which gives the reasons underlying the system's expectation that the opponent would have no opportunities following the computer's turn. CASTLE's diagnosis engine determines that it did in fact counter all active opponent plans at the time the expectation was posted, but that it was not able to defend against all threats that were enabled on the subsequent turn. Because all the direct threats were in fact detected, and since the counterplanning

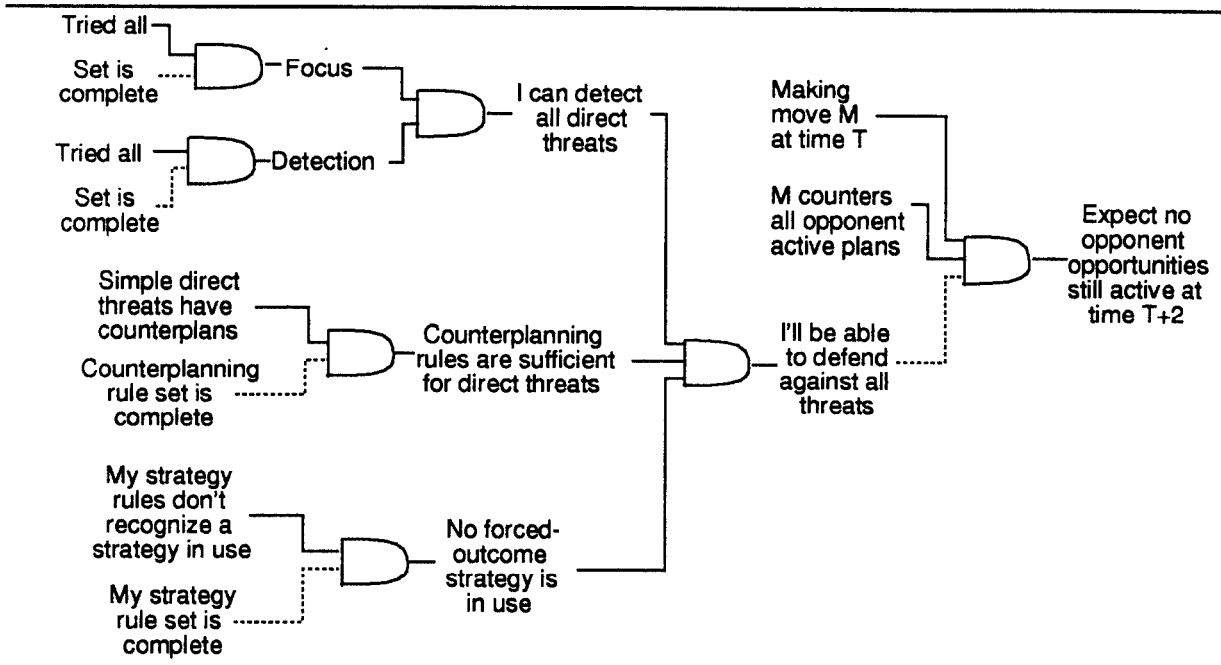


Figure 6.20: Justification of failed expectation in *fork* example

component was in fact able to generate counterplans for all of them, the system concludes that some scheme was used to enable several threats at once, and thus that there must have been an offensive strategy in use. Since the strategy rules did not recognize a strategy being applied, CASTLE concludes that its offensive strategy component rule set is incomplete.

To repair this rule set completeness failure, CASTLE first retrieves the *component specification* for the offensive strategy component, which is shown in figure 6.21. This specification simply defines the output of the forced-outcome strategy component in terms of a notion of *forced goal achievement.* This is defined separately instead of directly in the component specification to facilitate the use of a number of recursive inference rules used in

```
(def-brule method-precond-strategy
  (method-precond strategy                                    A plan is a forced-
    (strategy ?strategy-meth ?player (world-at-time ?time)    outcome strategy
              (goal-capture ?target-piece ?target-loc ?info)  for a goal
              ?plan))
<=
  (and (forced-goal-achieve ?plan                             If it is guaranteed
         (goal-capture ?target-piece ?target-loc ?info)       to satisfy the goal
         ?player (world-at-time ?time))))
```

Figure 6.21: Specification for the *offensive strategy* component

```
(def-brule forced-goal-1                                    A single-move plan
  (forced-goal-achieve (plan ?move done)                    forces the
      (goal-capture ?target ?loc2 (move ?move))             achievement of a
      ?player (world-at-time ?time) )                       piece capture
<=
  (and (= ?move (move ?player (capture ?target) ?piece      if the move is to
                      ?loc1 ?loc2))                         directly capture it
       (move-doable ?move (world-at-time ?time)) ))         and is doable


(def-brule forced-goal-2                                    A multi-move plan
  (forced-goal-achieve (plan ?enabler (next ?move2 ?rest))  forces the
     ?goal ?player (world-at-time ?time))                   achievement of a goal
<=
 (and (= ?cp-time (1+ ?time)) (= ?opp (player-opponent ?player))
      (= ?enabler                                           if the first move
         (move ?player ?move-type ?en-piece ?loc1 ?loc2))   is currently doable
      (move-doable ?enabler (world-at-time ?time))          and the rest of the
      (forced-goal-achieve (plan ?move2 ?rest) ?goal ?player plan achieves the
                           (world-at-time ?cp-time))        goal, and the first
      (expl-enabled ?enabler ?move2 ?time)                  enables the rest
      (= ?other-move
         (move ?player (capture ?other-target) ?other-piece and another move
               ?other-loc1 ?other-loc2))                    is enabled by the
      (forced-goal-achieve (plan ?other-move ?other-rest)   first move that
         ?other-goal ?player (world-at-time ?cp-time))      satisfies another
      (expl-enabled ?enabler ?other-move ?time)             goal,
      (not (= ?other-move ?move2))
      (no (and (counterplan ?cp-meth1 ?opp ?goal ?cp-time ?cp) and there is no
               (counterplan ?cp-meth2 ?opp ?other-goal      counterplan to both
                            ?cp-time ?cp))) ))              plan continuations
```

Figure 6.22: Two definition rules for *forced goal achievement*

the definition. Two rules for forced goal achievement that are used in this example are shown in figure 6.22. The first rule shown says simply that a move that plans to immediately capture a piece will necessarily satisfy the goal to capture the piece. The second rule represents one way that a future goal achievement is forced, namely if there is another threat enabled at the same time that the threatened player will have to attend to:

TO COMPUTE: How a multi-move plan necessarily satisfies a goal

DETERMINE: A feasible first move
and a subsequent plan that satisfies the goal
such that the first move enables the subsequent plan
and another enabled subsequent plan
that also achieves a goal
such that there is no one counterplan against both

The use of the intermediate concept of forced goal achievement raises several issues. First, the definition of forced-move strategies is not restricted to plans to capture pieces. Other goals can be incorporated by adding rules analogous to the first rule in figure 6.22. Secondly, the rules shown will in principle explain forced-move sequences of arbitrary length, as long as the system has rules for every method of forcing a move. The second rule in figure 6.22 represents forcing by enabling another threat; similar rules for other means of forcing will enable explanation of long forced-outcome plans.

The use of recursive concept definitions in EBL also raises a number of technical issues. The first is how the system will know to stop recurring. In our example, the question is how the system knows that it has reached the beginning of the forced-outcome strategy. CASTLE handles this in the current example by assuming that the forced-outcome strategy began at the time that the system applied its plan recognition rules. Additionally, the order in which inference rules are applied is designed to search first for explanations with minimal recursion. Several more complex approaches are possible. One is to reason at each stage whether the previous action was causally relevant to the plan being examined; if it's not then the recursive process should stop. Another is to check at each stage whether the explanation being constructed is sufficient to construct an effective repair; if so then further explanatory inference is unnecessary [Freed, 1991].[6]

Returning to the current example, the explanation of the specification (shown in figure 6.23) results in an explanation of why the strategy component should have generated the opponent plan to fork the computer's rook and bishop. Intuitively, this is true because:

1. The first move (in board (b)) enabled attacks on the computer's rook and bishop
2. Both were enabled by the attack's being from the new location of the moved piece
3. There was no double-counterplan

On the surface, however, CASTLE's explanation refers to three separate moves: the one that was made (that enabled the other two), the one that was about to be made (to capture the bishop), and the one that was disabled (the attack on the rook). Without analyzing

[6]Recursion has been a perennial problem in learning. PRODIGY [Minton, 1988a], for example, employs a special class of *discriminator functions*, which guide the selection of explanation rules at each point in explanation, in part for the purpose of bounding recursive explanations. Another issue in using recursive explanation rules is the generalization of iterative plans in a form that makes the iteration explicit [Shavlik, 1990]. CASTLE does not currently handle this.

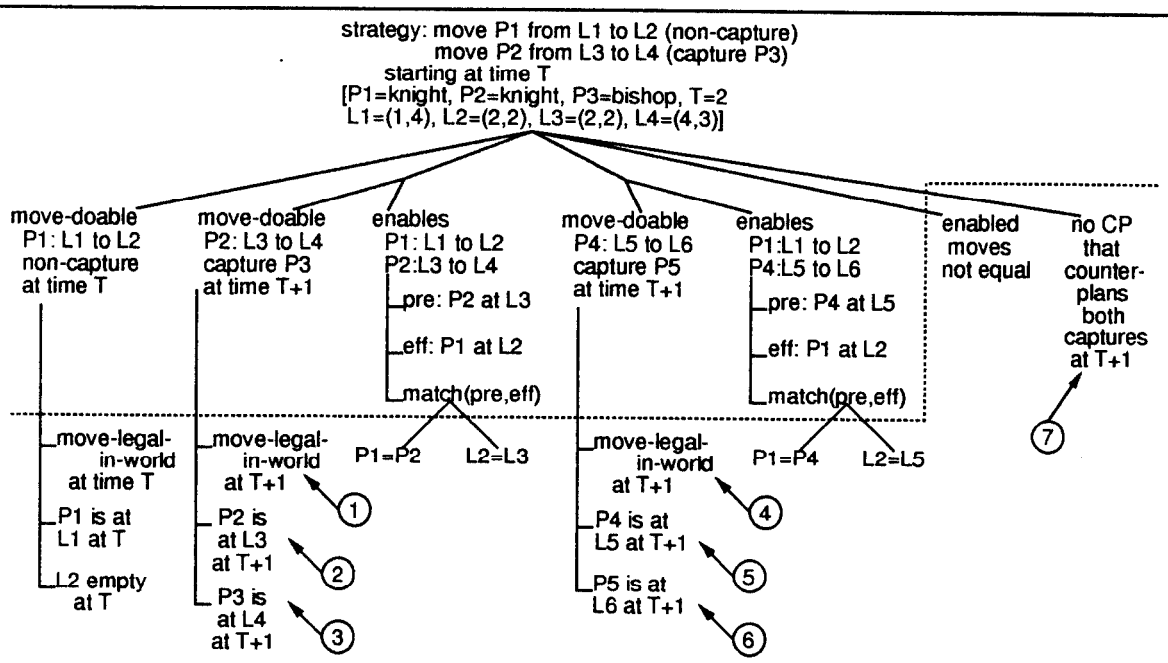


Figure 6.23: Explanation of the *fork* strategy

the explanation, CASTLE does not know the necessary relationship between the three moves, namely that all three are moving the same piece, and that the starting location of the two captures is the same as the destination of the first move. It is this information that must be determined using explanation-based learning. Besides the generalization of variables, the EBL process allows CASTLE to see the relationship between the three pieces (and the three corresponding locations), which to achieve the fork strategy must be codesignated.

One additional issue comes up in constructing the new fork rule that we have not seen in previous examples, namely the need to transform some of the conjuncts to a form that is *applicative* instead of *explanatory*, as discussed in section 5.6.1. Three sets of nodes in the fork explanation need to be transformed in this way: The leaves that support the `doable` beliefs for the two enabled moves, which assert that the enabled moves are doable at the intermediate time step (labeled in figure 6.23 as nodes **1-3** and **4-6** respectively), and the `(no (and (counterplan ...)))` node, which refers to the lack of a single counterplan at the intermediate time step to handle both enabled attacks (labeled node **7**). Because all of these beliefs refer to facts that will only be true *after* the strategy rule being learned must be applied, they must be backed up in time.

CASTLE transforms these expressions to be applicable at an earlier time using the approach discussed in section 5.6.1, with the following transformations:

```
(move-legal-in-world (move ?player (capture ?target-piece)
                           ?piece (loc ?r3 ?c3) (loc ?r2 ?c2))
                     (world-at-time ?time2))
```
The capture is a legal move at the middle time step
(The first leaf in the second branch, labeled **1***)*
```
 =>
(move-legal-in-world (move ?player (capture ?target-piece)
                           ?piece (loc ?r3 ?c3) (loc ?r2 ?c2))
                     (world-at-time ?time))
```
The capture is a legal move at the initial time step

```
(at-loc ?player ?piece (loc ?r3 ?c3) ?time2)
```
The forking piece is at the intermediate location in the middle time step
(Second leaf in second branch, labeled **2***)*
```
 => removed
```

```
(at-loc (player-opponent ?player) ?target-piece (loc ?r2 ?c2) ?time2)
```
The target piece is at the target location in the middle time step
(The third leaf in the second branch, labeled **3***)*
```
 =>
(at-loc (player-opponent ?player) ?target-piece (loc ?r2 ?c2) ?time)
```
The target piece is at the target location in the initial time step

```
(Above three duplicated for the alternative capture,
in the fourth branch, labeled nodes  4 through  6)

(no (and (counterplan ?cp-meth ?opp ... ?time3)
         (counterplan ?cp-meth2 ?opp ... ?time3)))
```
There is no double-counterplan at the final time step (leaf **7***)*
```
 =>
(no (and (counterplan ?cp-meth ?opp ... ?time)
         (counterplan ?cp-meth2 ?opp ... ?time)))
```
There is no double-counterplan at the initial time step

Once the transformation process is complete, the leaves are bundled into a new rule, shown in figure 6.24, which is added to the strategy component rule set. The rule is given a justification consisting of the generalized explanation in figure 6.23, which specifies why CASTLE believes the rule to be correct. The rule now gives CASTLE the necessary knowledge about the fork strategy, both for recognizing opponent opportunities, and for offensive application against the opponent. As in previous examples, we tested CASTLE on a number of fork variations, which are shown in figure 6.25, which the system handled without problem.

6.4.3 Discussion of the fork example

CASTLE's approach to transforming learned expressions to application-time vocabulary,

```
(BRULE learned-strategy-method8574
(strategy learned-strategy-method8574 ?player (world-at-time ?time)          A forced-outcome
          (goal-capture ?target-piece (loc ?r2 ?c2) ?info)                  strategy for
    (plan (move ?player move ?en-piece (loc ?r ?c) (loc ?r3 ?c3))           capturing a piece
          (next (move ?player (capture ?target-piece) ?en-piece             using two moves
                      (loc ?r3 ?c3) (loc ?r2 ?c2)) done)))
<=
 (and (= ?opp (player-opponent ?player)) (= (current-game) chess)
     (at-loc ?player ?en-piece (loc ?r ?c) ?time)                           Find a piece
     (move-legal-in-world                                                   that can move
        (move ?player move ?en-piece (loc ?r ?c) (loc ?r3 ?c3))
        (world-at-time ?time))
     (not (at-loc ?other-player ?other-piece (loc ?r3 ?c3) ?time))          to an empty square
     (at-loc ?opp ?target-piece (loc ?r2 ?c2) ?time)                        and a piece of the
     (move-legal-in-world (move ?player (capture ?target-piece) ?en-piece   opponent's that
                                (loc ?r3 ?c3) (loc ?r2 ?c2))                can be attacked
        (world-at-time ?time))                                              by the moved piece
     (at-loc ?opp ?other-target (loc ?r4 ?c4) ?time)                        and another piece
     (move-legal-in-world (move ?player (capture ?other-target) ?en-piece   of the opponent's
                                (loc ?r3 ?c3) (loc ?r4 ?c4))                that can be taken
        (world-at-time ?time))                                              by the same piece
     (not (= (move ?player (capture ?target-piece) ?en-piece
                   (loc ?r3 ?c3) (loc ?r2 ?c2))
             (move ?player (capture ?other-target) ?en-piece
                   (loc ?r3 ?c3) (loc ?r4 ?c4))))
     (no (and (counterplan ?cp-meth ?opp
                (goal-capture ?target-piece (loc ?r2 ?c2)                   such that there is
                   (move (move ?player (capture ?target-piece) ?en-piece    no counterplan
                               (loc ?r3 ?c3) (loc ?r2 ?c2))))               for the opponent
                ?time ?cp)                                                  that handles both
              (counterplan ?cp-meth2 ?opp                                   of the attacks
                (goal-capture ?other-target (loc ?r4 ?c4)
                   (move (move ?player (capture ?other-target) ?en-piece
                               (loc ?r3 ?c3) (loc ?r4 ?c4))))
                ?time ?cp)))))
```

Figure 6.24: Learned strategy rule for the fork

discussed in section 5.6.1, assumes that all conjuncts that refer to facts *later* than the time of rule application will either (a) regress straightforwardly, in that they correspond to a fact that was true at the time the rule was applied, or (b) they are made true by other conjuncts in the rule or by the context in which the rule is applied. Thus, for each conjunct that refers to a fact later than the time at which the rule would have been applied, the system checks whether the conjunct was true at the rule-application time. If the conjunct was true at the earlier time, CASTLE assumes that this will always be the case, and the fact is transformed into an identical fact about the earlier time step. In the example, the `move-legal-in-world` expressions are assumed to regress, because CASTLE sees that in the example the moves were in fact legal at the earlier time as well as at the later time. On the other hand, the `at-loc` belief that states that the forking piece will be at the intermediate location at the middle time step (node **2**) does not regress, because it obviously was not true at the beginning of the plan. CASTLE therefore assumes that this belief, because it is not regressable, is established by other conjuncts in the rule. This is true in our example, because the piece being at the intermediate location could (in principle) be explained as a consequence of the enabling move

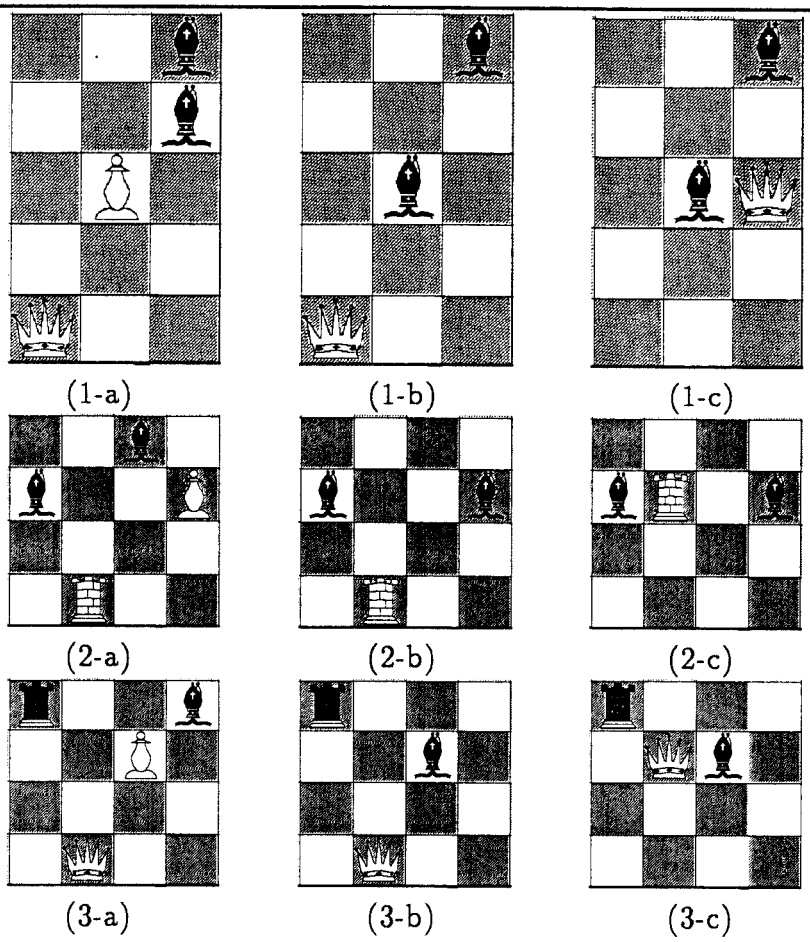


Figure 6.25: Variant examples of the fork

being the first move in the constructed plan.

The potential problem with both these assumptions is that CASTLE treats all the conjuncts as independent, and does not attempt to prove the relations (or lack thereof) that it is assuming. When the system sees that the `move-legal-in-world` beliefs were true at rule-application time, it does not check whether there are any other beliefs that must be true in order for this fact to be maintained between the two time steps. Conversely, when the `at-loc` belief is seen not to be regressable and is assumed to be established by other conjuncts in the plan, CASTLE does not compute how this establishment comes about and whether any additional assumptions are necessary. Clearly CASTLE uses an approximation for a more complete approach which would explain the necessary interactions between the conjuncts in the learned rule. CASTLE currently handles any deviations from its independence assumptions using an ad-hoc approach that we will see in the next example.

Another issue that is illustrated by this example is that CASTLE does not reason about the reliance of its proofs on implicit knowledge of the game of chess. For example, one assumption made by the explanation in the fork example is that the opponent will only be able to make one move at a time. This is why both attacks cannot be disabled unless

there is a single move that disables both. In other domains, however, this assumption may not be true, and more complex reasoning may be required. Some work has been done in trying to model these assumptions [Birnbaum *et al.*, 1990], but it is not reflected in CASTLE's implementation. These assumptions will have to be handled explicitly if CASTLE is to be applied to more than one domain, or to attempt to transfer knowledge from one domain to another [Collins and Birnbaum, 1988b; Krulwich *et al.*, 1990b].

Lastly, the question arises as to the efficiency of the learned strategy rule. CASTLE in general has been designed to avoid brute-force lookahead, preferring to search only within the constraints of particular rules. With the addition of the fork strategy rule, however, this is called into question, because the search that is performed approaches full two-move lookahead search.[7] One way to reduce this overhead is to further specialize the learned rules into forms that could take advantage of efficient geometrical representations of the necessary board positions.[8] For example, a fork rule that is specialized for horizontal (or vertical) straight-line forks could simply search for two opponent pieces on the same row (or column) with no pieces in between them, and then look for a rook or queen that could move somewhere between them on that line. A set of rules of this type would allow a system to find instances of a strategy without as much search.

6.5 Learning about simultaneous attacks

Consider the situation shown in figure 6.26, in which CASTLE makes a mistake similar to that of the previous example. Prior to the situation in board (a) CASTLE had determined that the opponent had no available offensive strategy, and therefore expected that at the end of its subsequent turn it would have handled all opponent threats. In figure 6.26(a), however, the opponent uses an unknown offensive strategy to establish two threats, one against the system's bishop and one against the system's rook. CASTLE detects the failure of its expectation, and as a result should learn a new offensive strategy.

The strategy the opponent employed is called *"simultaneous attacks,"* and is an offensive variant of the *discovered attacks* idea discussed in section 6.2. It is very similar to the *fork,* in that the opponent has made a move that enables two new attacks. In this case, however, the two attacks do not originate from the same piece, rather the second attack is enabled using a discovered attack.

The justification structure for the expectation in this case is the same as in the fork example, shown in figure 6.20. The diagnosis is also the same, and once again the system concludes that its offensive strategy component rule set is incomplete. CASTLE proceeds to retrieve the component specification, and invokes the explanation engine to explain why the opponent's move is an offensive strategy. The explanation, shown in figure 6.27, is similar to that of the fork example (figure 6.23): both concern the simultaneous enablement of two threats by a single opening move. The difference is that in our current example the

[7]The learned rule is still more efficient than full search, because it only examines intermediate opponent moves insofar as they counterplan against the two attacks.

[8]The idea of learning abstract component methods and geometrically specializing them for efficient use came out of a number of discussions with Eric Jones.

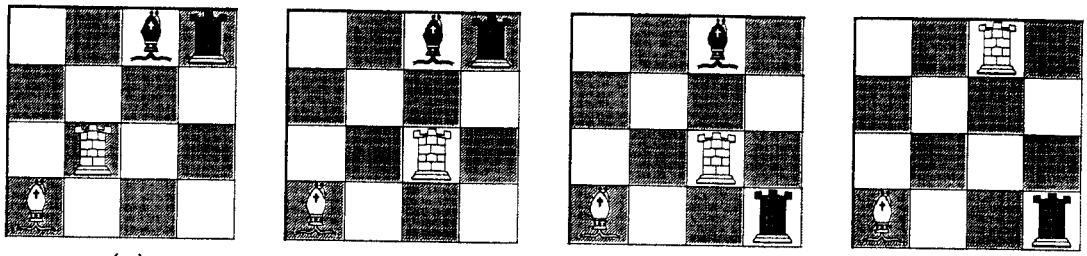


Figure 6.26: *Simultaneous attacks* example: Opponent (white) to move

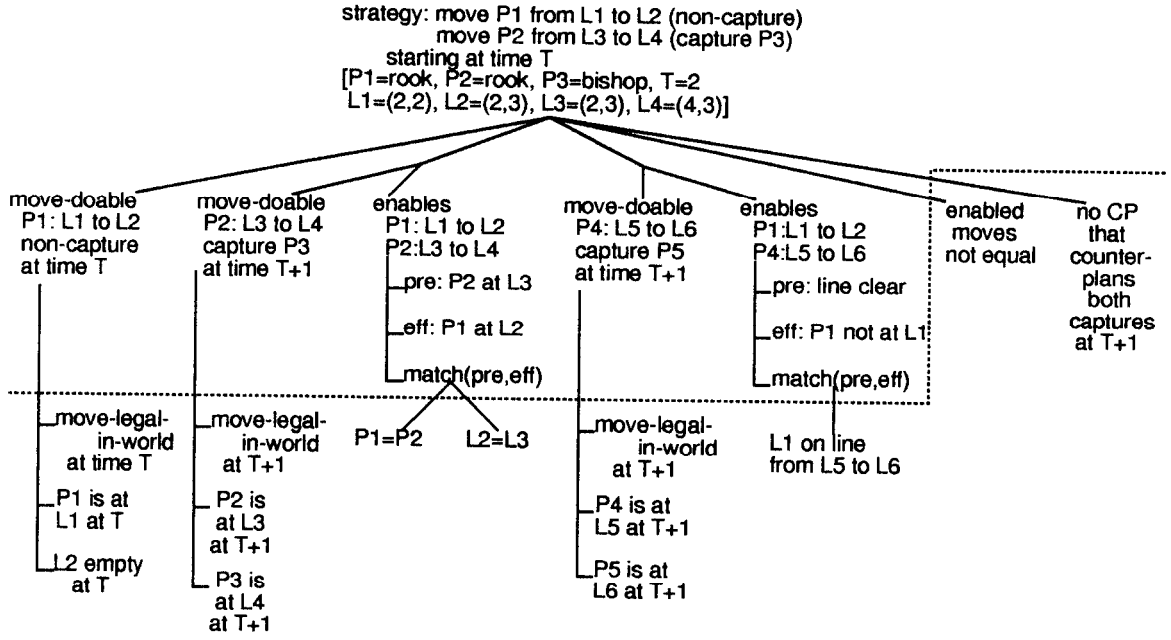


Figure 6.27: Explaining the *simultaneous attacks* strategy

"alternative" attack, which was not carried out, was enabled by a discovered attack (vacating a square along a line of attack), rather than by the piece having been moved to the attacking location.

After EBL is used to generalize the proof tree, the conjuncts are transformed for use at application time, as previously discussed. While most of the conjuncts are the same as in the fork example, one of the `move-legal-in-world` conjuncts is different and needs a different transformation to application time. The reason for this is that in the fork example both attacks were legal moves in the initial board situation, except that the attacking piece was not in position. In our current example, however, the bishop attack on the opponent's rook is not a legal move until *after* the computer's rook is moved out of the line of attack. Because

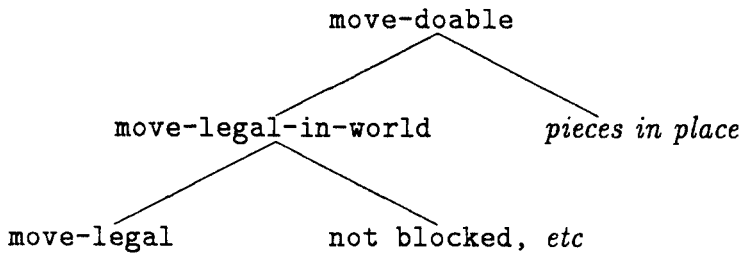


Figure 6.28: Legal move predicates

of this the invocation of `move-legal-in-world` would fail in the initial situation, and so the conjunct cannot be directly transformed back in time. Rather, the conjunct must be replaced with one that checks the legality of the move in principle, without worrying about whether it can in fact be made in the current situation. This is accomplished in CASTLE with an invocation of the `move-legal` predicate, so the conjunct is transformed by replacing it with an invocation of `move-legal`.

While this transformation is handled in CASTLE as an arbitrary specification,[9] its conceptual basis lies in the decomposition of `move-legal-in-world` into its components. Figure 6.28 shows the relationship between the predicates `move-doable`, `move-legal-in-world`, and `move-legal`. In the explanation of our current example, `move-doable` is decomposed into its components, and it is these components that are deemed operational and are thus included in the learned rule. In the process of backing the conjuncts up in time, we see that `move-legal-in-world` cannot itself be used, and that we must decompose it in turn. One of the resulting conjuncts—that the attack will not be blocked—will be made true by the plan to move the piece out of the way. This is why the `move-legal` expression is the only conjunct to be included in the new rule shown in figure 6.29.

One limitation of CASTLE's explanation, and thus of the learned rule, is that the attack by the moved piece is the attack that must actually be carried out, while the discovered attack (in this case of the bishop on the rook) must be the threatened attack that the opponent disables. This is a consequence of the fact that the specification for forced-outcome strategies as currently written refers to two particular moves. Fixing this requires rewriting the component specification in a way that reasons about sets of available attacks but not about particular ones.

[9]The system does not implement the inferences specified below, nor is it able to reenter the EBL algorithm and further decompose conjuncts that cannot be transformed to application time. Both of these steps are approximated in the system's transformation rules that have been discussed previously.

```
(BRULE learned-strategy-method10486
(strategy learned-strategy-method10486 ?other-player2                    A forced-outcome
    (world-at-time ?time2)                                               strategy for
    (goal-capture ?target-piece (loc ?r3 ?c3) ?info)                     capturing a piece
    (plan (move ?other-player2 move ?other-piece2 (loc ?r ?c) (loc ?r2 ?c2))   using two moves
          (next (move ?other-player2 (capture ?target-piece)
                      other-piece2 (loc ?r2 ?c2) (loc ?r3 ?c3))
                done)))
<=
(and (= ?opp (player-opponent ?other-player2)) (= (current-game) chess)
     (at-loc ?other-player2 ?other-piece2 (loc ?r ?c) ?time2)             Find a piece to
     (move-legal-in-world                                                move to a new
       (move ?other-player2 move ?other-piece2 (loc ?r ?c) (loc ?r2 ?c2)) location
       (world-at-time ?time2))
     (not (at-loc ?other-player ?other-piece (loc ?r2 ?c2) ?time2))      and an opponent
     (at-loc ?opp ?target-piece (loc ?r3 ?c3) ?time2)                    piece that can be
     (move-legal-in-world                                                attacked by the
       (move ?other-player2 (capture ?target-piece)                      attacking piece
             ?other-piece2 (loc ?r2 ?c2) (loc ?r3 ?c3))                  from its new
       (world-at-time ?time2))                                           location, and
     (not (= ?other-piece3 knight))                                      another non-knight
     (newest-loc-on-line (loc ?r ?c) (loc ?r4 ?c4) (loc ?r5 ?c5))       piece that can
     (at-loc ?other-player2 ?other-piece3 (loc ?r4 ?c4) ?time2)          attack another
     (at-loc (player-opponent ?other-player2) ?other-target              opponent piece
             (loc ?r5 ?c5) ?time2)                                       along the line of
     (move-legal (move ?other-player2 (capture ?other-target)            attack emptied by
                       ?other-piece3 (loc ?r4 ?c4) (loc ?r5 ?c5)))       moving the first
     (not (= (move ?other-player2 (capture ?target-piece)                piece
                   ?other-piece2 (loc ?r2 ?c2) (loc ?r3 ?c3))
             (move ?other-player2 (capture ?other-target)
                   ?other-piece3 (loc ?r4 ?c4) (loc ?r5 ?c5))))
     (no (and (counterplan ?cp-meth ?opp                                 such that no
              (goal-capture ?target-piece (loc ?r3 ?c3)                  single opponent
                (move (move ?other-player2 (capture ?target-piece)       move can
                            ?other-piece2 (loc ?r2 ?c2) (loc ?r3 ?c3))))  counterplan
              ?time2 ?cp)                                                both attacks
             (counterplan ?cp-meth2 ?opp
               (goal-capture ?other-target (loc ?r5 ?c5)
                 (move (move ?other-player2 (capture ?other-target)
                             ?other-piece3 (loc ?r4 ?c4) (loc ?r5 ?c5))))
               ?time2 ?cp)))) )
```

Figure 6.29: Learned rule for the *simultaneous attacks* example

6.6 Learning about pinning

We will now consider the last set of examples that have been implemented in CASTLE, which involve learning methods for the `option-limiting` planner component. Option-limiting plans are actions that significantly constrain the opponent's ability to carry out his own plans. The `option-limiting` planner component consists of rules that generate plans for limiting the opponent's options. We will see that this is often carried out by liming the mobility of the opponent's pieces.

Consider the example shown in figure 6.30. In board (a) the opponent (playing white) chooses to move the queen to the right, to a square from which it can subsequently be moved to pin the computer's rook against the king. The computer (playing black) doesn't detect

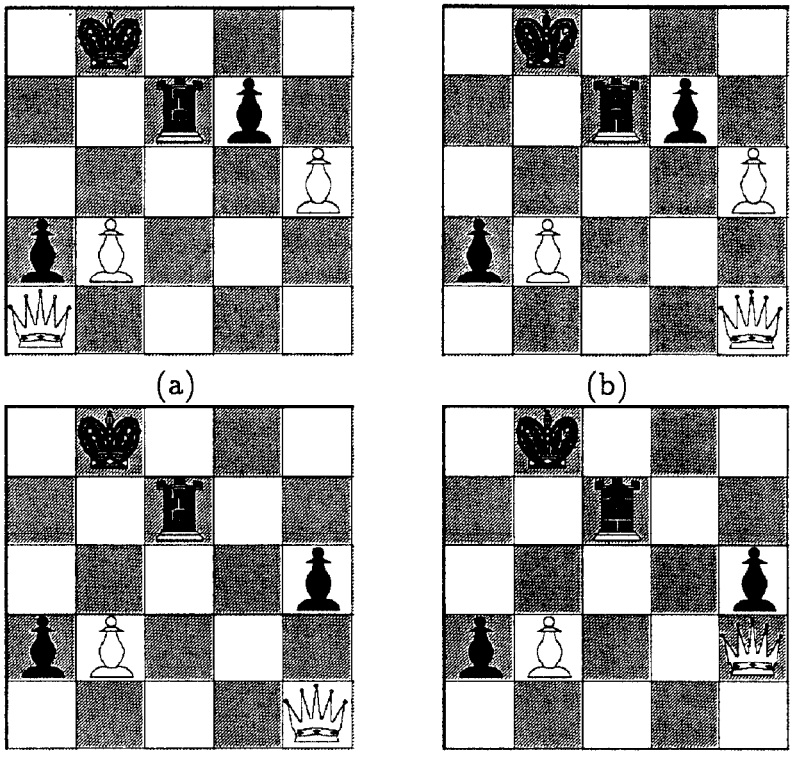


Figure 6.30: *Pin* example: Opponent (white) to move

any strategies (offensive or option-limiting) that the opponent can execute, so it goes ahead with its own plan to capture the opponent's pawn. In board (c) the opponent moves its queen to pin the computer's rook, and the computer finds itself in board (d) with its rook pinned and a pawn (or the queen itself) able to make the capture in two moves.

The expectation that failed in this example relates to an assumption that is implicitly made by all intentional systems, that in general there will exist reasonable options—plans—that can be carried out for any goal that arises. This assumption underlies CASTLE's belief that it will be able to achieve its goals over the course of the game. One instantiation of this assumption, which CASTLE can directly monitor, is that the natural prerequisites to carrying out the plan, namely the ability to move pieces, will in general be met. In other words, the system's pieces will have mobility. It is common needs such as mobility that motivates the **option-limiting** planning component in the first place.

Since the opponent presumably would like to limit the system's options, CASTLE uses its **option-limiting** planning rules during the plan recognition phase to check whether the opponent has the ability to limit CASTLE's options. If so, CASTLE will try to counterplan. If not, CASTLE will assume that its pieces will remain mobile until the subsequent turn. This process serves two purposes. First, any plans of the opponent's to limit CASTLE's options will

```
(def-brule recog-plan-limit
  (recog-plan recog-limit ?opp-player (world-at-time ?time)
     ?plan (goal-limit ?piece ?loc ?info))
 <=
  (and (in-set ?limit-meth limit-methods)
       (limiting-plan ?limit-meth ?opp-player
           (world-at-time ?time)
           ?plan (goal-limit ?piece ?loc ?info))
       (not (active-goal ?opp-player
              (goal-limit ?piece ?loc ?info) ?time)) ))
```

To determine an opponent plan to limit the system's options, Retrieve an option-limiting plan method that generates a plan for the opponent that's not already a known active goal

Figure 6.31: Recognizing opponent plans to limit computer options

hopefully be anticipated and countered. Second, if CASTLE fails to detect an opportunity for the opponent to limit the system's options, and the opponent takes advantage of the opportunity, CASTLE could learn a new **option limiting** planning rule in response to the failure. This is exactly what happens in the pin example.

Let's step through what happens in the example in more detail. In board (b) it is CASTLE's turn, and one of the steps in the plan recognition phase is to apply the **option-limiting** planner component from the perspective of the opponent to see if the opponent has any available plans to limit the system's options. The rule for doing this is shown in figure 6.31, which says roughly:

TO COMPUTE: A plan the opponent can execute

DETERMINE: An option-limiting plan method
that generates a plan for the opponent
that he can currently use

In the situation in board (b) no such plan is detected, and CASTLE proceeds to capture the opponent's pawn, simultaneously satisfying the goal to capture the pawn and the goal to counter the threat on the system's pawn. CASTLE additionally checks which of its pieces are considered "mobile,"[10] and posts an expectation that they will remain mobile after the opponent's turn (since the opponent has no option-limiting plans). The rule that generates this expectation, shown in figure 6.32, says roughly:

TO COMPUTE: A piece mobility expectation to generate

DETERMINE: The pieces on the board
that can be moved to more than two places
that the opponent has no plan to limit

[10]CASTLE currently defines mobility as the ability to move to more than two places, and lack of mobility (for detecting the failure) as being able to move to less than two locations. There are many other conceivable schemes, such as detecting a significant change in the number of possible moves, or detecting a percentage reduction. Such schemes can easily be encoded in CASTLE's rules for expectation posting and monitoring.

```
(def-brule exp-mobility
  (should-expect exp-mobility ?time                                  To determine what
      (mobile-pieces computer ?piece-set ?exp-time))                 to expect regarding
<=                                                                   piece mobility
  (and (move-to-make (move computer ?the-move-type ?the-piece
                           ?the-loc1 ?the-loc2)                      Retrieve the move
         computer ?the-goal ?time)                                   I've chosen to make
       (is-set-of ?piece-set (var ?piece-loc-pair)                   and construct a set
         (and (at-loc computer ?piece ?loc ?time)                    by looking at each
              (not (= ?piece ?the-piece))                            unmoved piece,
              (no (active-goal opponent                              whose mobility the
                    (goal-limit ?piece ?loc ?info)                   opponent can't
                    ?time))                                          limit,
              (num-such-that (vars ?num ?loc2 3)                     which can move
                 (move-doable (move computer ?move-type              to at least two
                                    ?piece ?loc ?loc2)               locations
                              (world-at-time ?time)))
              (> ?num 2)                                             and store the piece
              (= ?piece-loc-pair (piece-loc ?piece ?loc)) ))         and its location in the
       (not (= ?piece-set (empty-set)))                              set to expect to be
       (= ?exp-time (+ ?time 2)) ))                                  mobile next turn
```

Figure 6.32: Expecting pieces to remain mobile

In the situation in board (b) CASTLE observes that the rook will remain mobile, since it can move to more than two places and no limiting method applies. CASTLE thus posts the expectation that the rook will still be mobile in board (d).

After CASTLE makes its move, the opponent moves its queen in board (c), resulting in the situation in board (d) in which the computer's rook is pinned. This causes the mobility expectation to fail, which is detected by a failure detection rule which says:

IF There is a set of pieces expected to be mobile
and one of them is not currently mobile

THEN Signal the failure of the expectation

This rule, given in appendix A, uses an approach similar to the expectation-generation rule to examine each piece/location pair in the set and check its mobility.

This expectation failure invokes CASTLE's diagnosis and repair mechanisms. The diagnosis engine traverses the justification for the expectation, shown in figure 6.33, and determines that there must have been a plan executed by the opponent to limit the computer's options that was not generated by a method in the option-limit planning component. This fault description is then passed to CASTLE's repair module.

There are, of course, many alternative approaches to detecting lack of options. One would be to expect the system's plan generation components to be able to generate good plans in

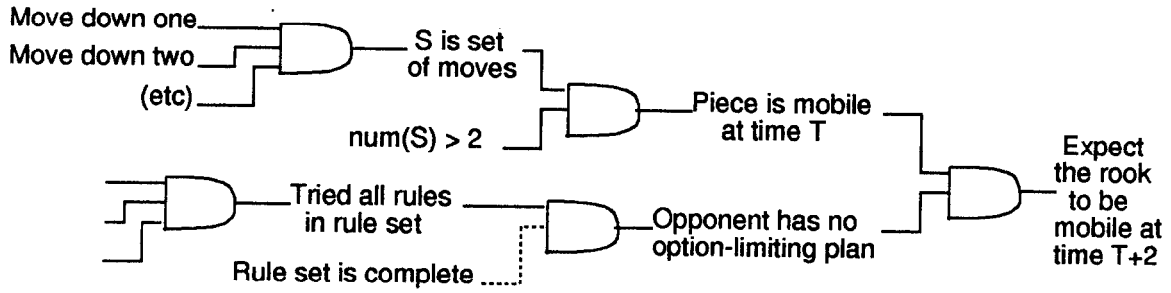


Figure 6.33: Justification of the mobility expectation of the *pin* example

any situation, and to specify an underlying assumption that hypothetical plans will in fact be feasible because the computer's options aren't artificially limited. In this approach, the failure would go undetected until the opponent takes advantage of the pinned piece (e.g., by advancing its pawn to capture the rook). At that point all suggested counterplans would be infeasible, resulting in an expectation failure. The alternative justification shown in figure 6.34 would then lead the diagnosis engine to the correct conclusion. This alternative approach is not implemented in CASTLE, but is completely plausible within CASTLE's framework for planning and expectation monitoring.

Once CASTLE has isolated the fault as a lack of an option-limiting planning rule, the repair mechanism takes over and first generates an explanation of why the opponent's move constituted an option-limited plan. The component specification for the option-limiting planning component is shown in figure 6.35, which says roughly that an option-limiting plan is a single move that disables more than two opponent moves by the same piece for the same reason. This specification is used to generate the explanation shown in figure 6.36.

The explanation uses a move precondition rule, given in appendix B, which says that a precondition for a move is that there is no opponent piece with a possible king threat through the square being vacated. The effect of the queen move, that the queen is now at its new location, conflicts with this precondition, because the queen in its new location does

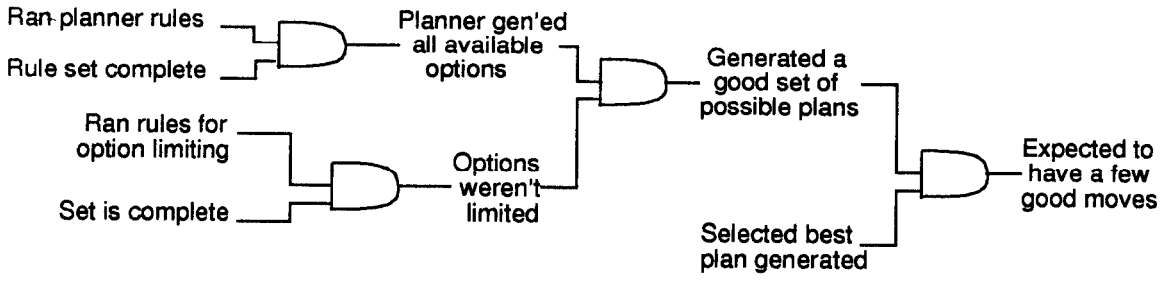


Figure 6.34: Alternative *pin* justification: Expecting plan feasibility

```
(def-brule meth-precond-limit
  (method-precond limiting-plan
    (limiting-plan ?limit-meth ?player (world-at-time ?time)      A one-move plan
       (plan ?the-move done)                                      is an option-
       (goal-limit ?dis-piece ?dis-loc1 (move ?the-move))))       limiting plan if
<=
  (and (= ?opp (player-opponent ?player)) (= ?limited-time (1+ ?time))
       (= ?the-move (move ?player ?the-move-type ?the-piece      the plan was
                          ?the-loc1 ?the-loc2))                  feasible
       (move-doable ?the-move (world-at-time ?time))
       (= ?dis-move1                                             and an opponent
          (move ?opp ?dis-move-type ?dis-piece                   move was feasible
                ?dis-loc1 ?dis-loc2))                            before the plan was
       (at-loc ?opp ?dis-piece ?dis-loc1 ?time)                  carried out
       (move-legal-in-world ?dis-move1 (world-at-time ?time))
       (not (move-doable ?dis-move1                              but not after the
             (world-at-time ?limited-time)))                     plan was executed
       (expl-disabled-reasons ?the-move ?dis-move1               and the plan
                              ?time ?pre1 ?eff)                  actually disabled
       (num-such-that (vars ?num ?dis-move 3)                    the move
           (and (= ?dis-move
                   (move ?opp ?dis-move-type-a ?dis-piece        and there are more
                         ?dis-loc1 ?dis-loc2a))                  than two moves
                (move-doable ?dis-move (world-at-time ?time))    disabled by the
                (not (move-doable ?dis-move                      plan due to the
                       (world-at-time ?limited-time)))           same effect of the
                (expl-disabled-reasons ?the-move ?dis-move       opponent move
                                       ?time ?pre ?eff) ))
       (> ?num 2) ))
```

Figure 6.35: The specification for the option-limiting plan component

in fact have a possible king threat through the rook's location. This conflict explains why the queen's move disables six previously possible rook moves.

The learned rule for the pin as an option-limiting plan is shown in figure 6.37. The rule says roughly that:

TO COMPUTE: A plan to limit the opponent's options

DETERMINE: Find a piece to move and a location to move to
and an opponent piece to pin
such that the moved piece can legally attack the king
and the attack is blocked only by the defending piece

This rule correctly predicts the opponent's ability to pin the computer's rook in the situation in figure 6.30(a), and can also be used offensively by CASTLE's planner to devise plans to limit the opponent's options.

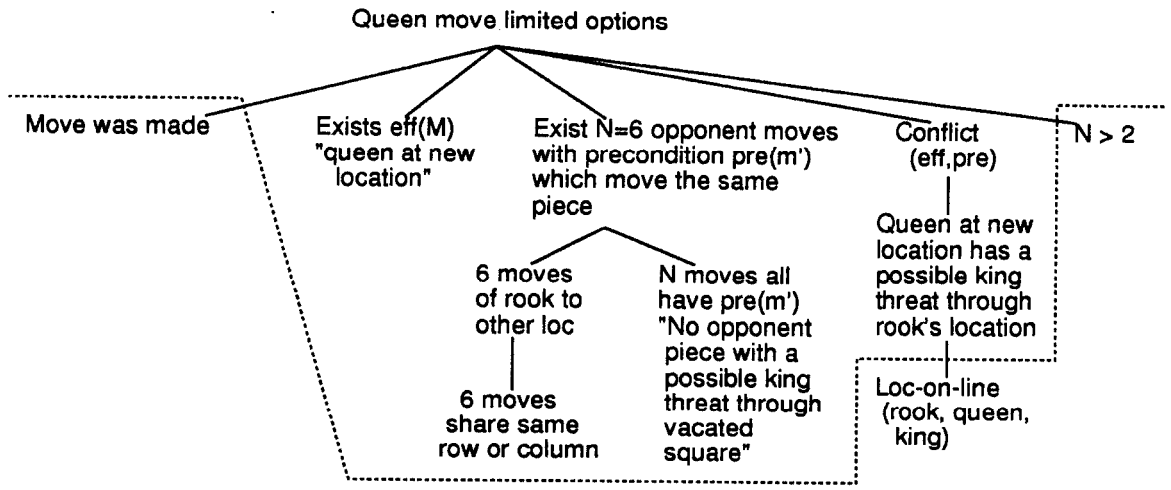


Figure 6.36: Explanation for the pin being an option-limiting move

There are two problems with the rule as it is learned by CASTLE. The first is inefficiency, in that the rule iterates over all pieces that CASTLE can move, and all locations that it can move to, and all opponent moves, and only then checks for the blocked attack on the opponent's king. A much more efficient scheme would be to look at the lines of attack on the opponent's king, and search for opponent pieces that can be pinned, and only then check for computer pieces that can be moved into the pinning location. This would cause the rule to execute much more quickly, with the same results.[11]

A more significant problem with the learned rule is that it only supports pinning pieces against the king. This is a consequence of basing the rule on explanations of *disablement,* because moves are strictly speaking only disabled when they prevent a king attack. It would be preferable, however, for CASTLE to be able to reason about the *undesirability* of a move, and how the effect of one move was to make another move undesirable. We will discuss this point in the following chapter, in the context of extensions to the current system.

6.7 Learning to box in

As a final example of CASTLE in action, consider the situation shown in figure 6.38, in which the opponent limits CASTLE's options by constraining the system's king not to move to the next row. In board (a) CASTLE does not recognize any opponent plan that must be countered, so it proceeds to capture the opponent's pawn. The opponent advances his rook, effectively *boxing in* the computer's king, since the rules of the game specify that a king cannot move to a location that is under attack. When this happens, CASTLE has a failure of a mobility expectation, because the king was previously mobile and now is not. The diagnosis proceeds as it did in the previous example, and CASTLE handles the failure by attempting to learn a

[11]This is another instance of the *sub-expression ordering problem* discussed in section 5.6.2.

```
(BRULE learned-limiting-plan-method3105
(limiting-plan learned-limiting-plan-method3105 ?opp                    To limit the
   (world-at-time ?time)                                               options
   (plan (move ?opp move ?other-piece2
               (loc ?r2 ?c2) (loc ?att-r ?att-c))
         done)
   (goal-limit ?def-piece (loc ?def-r ?def-c)                          of a defending
      (move (move ?opp move ?other-piece2                              piece
               (loc ?r2 ?c2) (loc ?att-r ?att-c)))))
<=
 (and (= ?def (player-opponent ?opp)) (= (current-game) ?chess)
      (at-loc ?opp ?other-piece2 (loc ?r2 ?c2) ?time)                 Find a piece
      (move-legal-in-world (move ?opp move ?other-piece2              that can move to a
                             (loc ?r2 ?c2) (loc ?att-r ?att-c))       new location
                           (world-at-time ?time))
      (not (at-loc ?other-player ?other-piece                         that's empty
                   (loc ?att-r ?att-c) ?time))
      (at-loc ?def ?def-piece (loc ?def-r ?def-c) ?time)              and a defending
      (move-legal-in-world (move ?def move ?def-piece                 piece that can move
                             (loc ?def-r ?def-c) (rc->loc ?r ?c))
                           (world-at-time ?time))
      (at-loc ?def king (loc ?to-row ?to-col) ?time)                  such that the piece
      (move-legal (move ?opp (capture king) ?other-piece2             can legally attack
                   (loc ?att-r ?att-c) (loc ?to-row ?to-col)))        the opponent's king
      (newest-loc-on-line (loc ?def-r ?def-c)                         but is blocked by
        (loc ?att-r ?att-c) (loc ?to-row ?to-col))                    the defending piece
      (no (and (newest-loc-on-line (loc ?other-r ?other-c)
                 (loc ?att-r ?att-c) (loc ?to-row ?to-col))           and such that no
          (not (= ?other-r ?def-r)) (not (= ?other-c ?def-c))         other square on the
          (at-loc ?other-player2 ?other-piece3                        line of attack
                  (loc ?other-r ?other-c) ?time)))))                  is occupied
```

Figure 6.37: Learned rule for limiting options by pinning

new **option-limiting plan** method.

CASTLE proceeds to construct an explanation of why the opponent's move was an option-limiting plan. This explanation, shown in figure 6.39, says roughly that *the rook move limited the system's options,* because *one effect of the move is that the rook was at location (2,1),* and *there are three king moves that have the precondition that no opponent piece is in place to attack the destination location, for which the rook is in place (at its new location) to attack.* The learned rule is shown in figure 6.40.

One aspect of the above explanation that is less than ideal is that it does not take notice of the fact that all three potential moves are disabled because the three destination locations for the king are on the same row as the rook, or even that the destination locations are by definition one row or column away from the king's present location. This is because

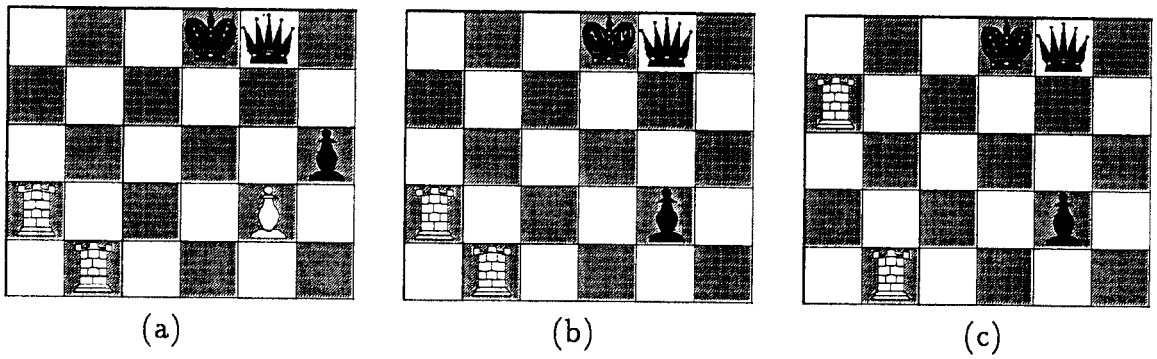


Figure 6.38: *Box in* example: Computer (black) to move

the explanation of the conflict merely notes that all three moves conflict with the effect of the rook move, and this conflict (and the precondition itself) is expressed in terms of move legality that do not take note of the geometrical pattern involved. Even though CASTLE realizes that the reasons that each of the potential attacks is legal is that the rook shares the destination row, and that rooks move along rows, it does not induce the pattern that all three moves are legal for this same reason. The same is true of the fact that the legal king moves are all one row or column away from the king's current location. In a sense the resulting explanation is more general, but the resulting rule is slightly different from what one might intuitively expect.

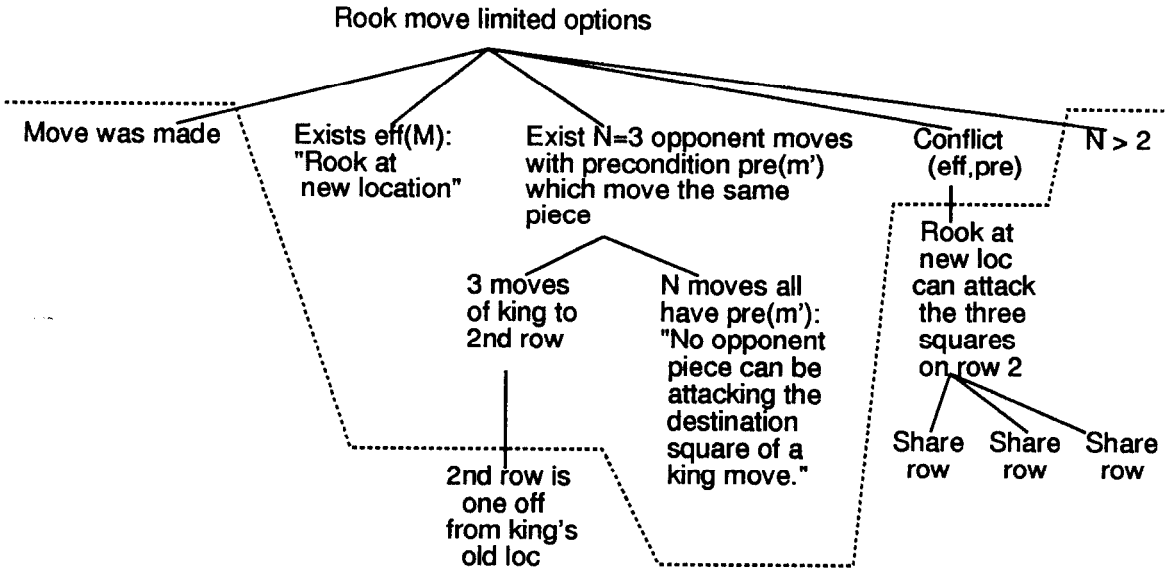


Figure 6.39: Explanation for boxing-in being an option-limiting move

```
(BRULE learned-limiting-plan-method13791
(limiting-plan learned-limiting-plan-method13791                    To limit the
 ?opp2 (world-at-time ?time)                                       options
 (plan (move ?opp2 move ?other-piece2 (loc ?r ?c) (loc ?r4 ?c4))
       done)
 (goal-limit king (loc ?r2 ?c2)                                    of the defending
   (move (move ?opp2 move ?other-piece2                            king
               (loc ?r ?c) (loc ?r4 ?c4)))))
<=
(and (= ?opp (player-opponent ?opp2)) (= (current-game) chess)
     (at-loc ?opp2 ?other-piece2 (loc ?r ?c) ?time)                Find a piece to
     (move-legal-in-world (move ?opp2 move ?other-piece2           move to a new
                            (loc ?r ?c) (loc ?r4 ?c4))             location
          (world-at-time time))
     (not (at-loc ?other-player ?other-piece                       that's vacant
                  (loc ?r4 ?c4) ?time))
     (at-loc ?opp king (loc ?r2 ?c2) ?time)                        such that the
     (move-legal-in-world (move ?opp move king                     defending king can
                            (loc ?r2 ?c2) (loc ?r3 ?c3))           move to a location
          (world-at-time time))
     (move-legal-in-world (move ?opp2 (capture king) ?other-piece2
                            (loc ?r4 ?c4) (loc ?r3 ?c3))           that the moved
          (world-at-time time))) )                                 piece can move to
```

Figure 6.40: Learned option-limiting rule for *boxing-in*

6.8 Evaluating CASTLE's success

The main point of this chapter has been to demonstrate that CASTLE's approach to learning can be used to learn a variety of different types of rules in a variety of different situations. CASTLE handles nine types of examples, learning rules for the counterplanning, detection focusing, forced-outcome strategy, and option-limiting plan components. It is also able to learn more than one type of rule from a single expectation failure.

The analyses of these examples certainly do not exhaust the possible ways in which these concepts could be learned, or the only way to learn from the situations. The point is rather that given a decision-making architecture that is structured in terms of components, and given a set of performance expectations to be monitored during execution, CASTLE can diagnose failures of the expectations in terms of faulty components and can learn new methods for them. There are certain limitations with respect to the scope of the examples that have been implemented, reflecting the fact that the decision-making model presented in chapter 3 is not rich enough to support the inferences that are necessary for significantly more complex cases. However, were the decision-making model to be extended, CASTLE's approach to learning and diagnosis could learn appropriately.

The further goal of this chapter has been to convey an intuition for how decision-making

Example	Component
Interposition	Counterplanning
Run away	Counterplanning
Counterattack	Counterplanning
Discovered attacks	Detection focusing
En-passant	Threat detection
Fork	Offensive strategy
Simultaneous attacks	Offensive strategy
Pin	Option limit
Boxing in	Option limit

Figure 6.41: Summary of implemented examples

processes can be effectively modeled. This includes the proper level at which to represent cognitive tasks, as well as how to represent the interactions between components and their specified purposes. In the absence of a formal theory of effective planner modeling, we have attempted to convey such an intuition through the set of examples that are effectively and synergistically handled by CASTLE. In chapter 7 we will see how the approach could be used to learn more complex concepts within a richer decision-making model, and will discuss the extensions necessary to the model to support this learning.

Chapter 7

Enhancements to CASTLE's model

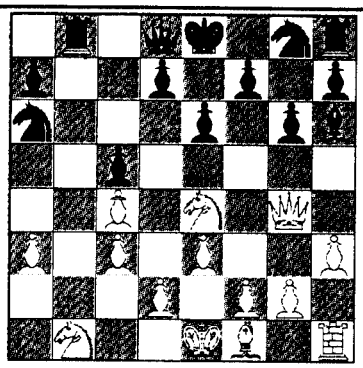
Up to this point, the self-models that we have discussed, and the examples of the learning behavior that they facilitate, have all been implemented in CASTLE. These models have been rich enough to demonstrate CASTLE's ability to learn flexibly. However, the models that have been implemented in CASTLE have a number of limitations, and these in turn impose limitations on the sophistication of rules that the system can learn.

This chapter goes beyond what has been implemented in CASTLE. We will discuss several complex examples that illustrate the sort of learning behavior that would be possible given richer models, and describe the enrichment of CASTLE's models that would be necessary in order to implement them. We will also reexamine the interposition example discussed earlier and see how it can be handled better using the model enhancements.

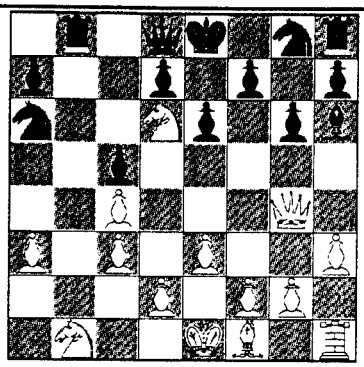
7.1 Learning to buy time

The idea of *buying time* is to delay a bad event until you can do something about it. This comes up in chess when a player puts the opponent in check in such a way that something new can be achieved on the subsequent turn. Figure 7.1 shows an example. In board (a) the computer (playing black) threatens the opponent's knight (on the bottom row) with its rook. The opponent is unable to disable the attack in board (b), but instead puts the computer in check, using its other knight. This forces the computer to delay carrying out the attack in order to save itself from check. In board (c) the system moves its king out of check, and the opponent (in board (d)) disables the original attack by interposing the second knight along the line of attack. We would like CASTLE to learn a new rule for counterplanning by buying time.

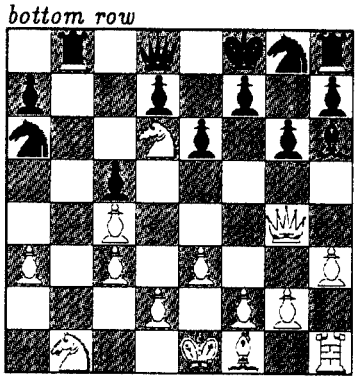
Given CASTLE's models as they have been discussed, CASTLE's expectation that its plan will succeed will fail in board (b), when the system is put in check and must postpone the continuation of its original plan. The repair module would then attempt to formulate a new counterplanning rule. Given an explanatory theory that specified that a precondition of a move is that either (a) the computer is not in check, or else (b) the move results in getting out of check—which is a rule of the game—CASTLE would learn a new rule for counterplanning by putting the opponent in check. This rule would apply whenever the opponent's attempted



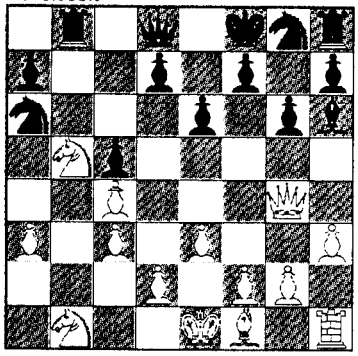
(a) *Computer's (black) rook threatens opponent knight on bottom row*



(b) *Opponent (white) moves other knight to put computer in check*



(c) *Computer moves king out of check*



(d) *Opponent interposes the second knight to block the original attack*

Figure 7.1: Counterplanning by *buying time*

threat would not itself get him out of check. In other words, the system would learn a simple counterplanning rule for disabling the attack's precondition that the attacker not be in check. In learning this rule, CASTLE would miss the point of putting the attacker in check in such a way as to enable another counterplan, because it would not realize that putting the attacker in check merely postpones the threat but does not disable it.

We can see from this example that CASTLE is missing a key concept in its representation of causality of threats, that a threat is more than a one-time potential attack. Threats persist over time, and are blocked and reinstated. Intuitively there is a difference between reinstating a threat that was temporarily disabled and establishing one that is altogether new. This distinction is paramount in learning the concept of buying time.

Another issue that arises in formulating a new rule for buying time is that it is not necessary to put the attacker in check to delay the threat; it's enough to threaten a piece that's more valuable than the one being attacked. In other words, if the opponent is attacking your bishop, you can buy time by threatening the opponent's queen as much as by putting the opponent into check. In this situation the attack isn't disabled *per se,* but you can

assume that the attacker will not carry out the attack until the subsequent turn.

This variant avoids the problem of distinguishing between temporary disablement and real disablement, because the attack is not seen as being disabled in the intermediate turn. Rather it has simply been rendered not in the attacker's best interests. On the other hand, proper treatment of the counterthreat move in buying time requires the notion that the move puts the attacker in a situation whereby he would rather defend against the counterthreat than carry out the original attack. This requires an explanatory theory of the factors that weigh in choosing a move to make. CASTLE can use the knowledge in its own plan selection component for such an explanation, but this example still requires that some additional factors be made explicit for use in explanation. In particular, the system's notion of the *consequences* of a course of action must be significantly extended.

There are several additional issues involved in buying time that do not come up within CASTLE but would arise in a more complex decision-making model or in a more complex domain. Suppose that a player is under attack, and is unable to formulate a counterplan, but sees that it can put the opponent in check (or in some way establish a threat that the opponent will have to attend to before carrying out the attack). Even if the defender doesn't know that this response will necessarily enable a counterplan on the subsequent turn, it will often still be a good move to make. One reason is that in many domains there are limited computational resources allotted to each decision, so that buying time would give the defender additional time to consider potential responses. Another reason for buying time is that there are often indirect consequences of a move that are difficult to see in advance, that could nevertheless be helpful in counterplanning the original attack. Buying time may enable a counterplan that the defender does not initially foresee. Lastly, many domains have forces external to the attacker and defender that can influence the situation, and in such domains the defender can buy time hoping for a serendipitous event to help him out of his predicament.

7.1.1 Model enhancements seen in buying time

The buying time example has presented several areas in which CASTLE's models can be enhanced:

- Persistence and reestablishment of threats
- Explanatory reasoning about desirable consequences of actions

Additionally, there are several issues that would arise in a more complex domain:

- Reasoning about limited computational resources
- Reasoning about the likelihood of serendipitous events
- Reasoning about the possibility of unforseeable consequences

The effect of these model enhancements on CASTLE's learning ability can be assessed by the degree to which the same enhancements come up in other examples as well. In fact, we will see some of them—chiefly reasoning about desirable consequences of actions—arising in other examples in this chapter.

7.2 Learning about pawn line defenses

What would be necessary for CASTLE to learn the concept of *pawn line defenses,* in which a player arranges a line of pawns to systematically block potential incoming attacks? CASTLE already has knowledge of blocking threats by interposing pieces, but is missing several notions that are necessary in order to understand pawn line defenses:

1. An inability to formulate effective plans may be a consequence of intentional activity by the opponent.

2. Goals can be *amortized,* in that they can refer to a set of possible targets rather than one specific target.

3. A plan may be formulated and carried out in anticipation of likely threats, even if they are not at all evident from the current board situation.

Consider how CASTLE could learn about pawn line defenses from seeing the opponent using them. Suppose the opponent established an effective pawn line defense. What expectation might fail as a result? Intuitively, the consequence of an opponent's pawn line defense is an inability to construct plans that involve board regions on the other side of the line. In general, planners assume that they will be able to generate plans for a reasonable number of their goals, and a pawn line defense is one way that an opponent can frustrate this assumption. An inability to generate plans should cause an expectation failure, and this expectation failure should lead to learning about the cause of the inability, namely the pawn line defense.

What component is at fault here? There are several intuitive possibilities, one of which is that the opponent's pawn line defense should have been foreseen as an *option limiting* plan, so the option-limiting planning component is at fault for not having a rule corresponding to pawn line defenses. To make this diagnosis, CASTLE must be able to realize that its expectation that it will be able to construct effective plans is based on the assumption that it can, in general, move its pieces to any areas of the board that it wants to. Its belief that it will be able to do this is in turn justified by its not having recognized any opponent plans to limit the mobility of its pieces to the other side of the board. Using such a justification structure, CASTLE's diagnosis engine can conclude that the fault is a missing option-limiting plan rule.

CASTLE now has to explain why the establishment of the pawn line was an option-limiting plan, which brings us to our second model enhancement for this example, namely extending the notion of option-limiting to allow for limiting the movement of a group of pieces instead of just one. In other words, the pawn line defense doesn't strongly limit a single piece's options, as the pin does. Rather, it limits the collective options of all of the player's pieces. This requires extending the notion of option-limiting goals to amortize over a number of possible moves.

The last difficulty in learning this idea is that the pawn line was established *before* the moves that it blocks could be made, and in fact this must be so due to the time required to establish the pawn line. This idea of *planning in advance* is new to CASTLE, which always

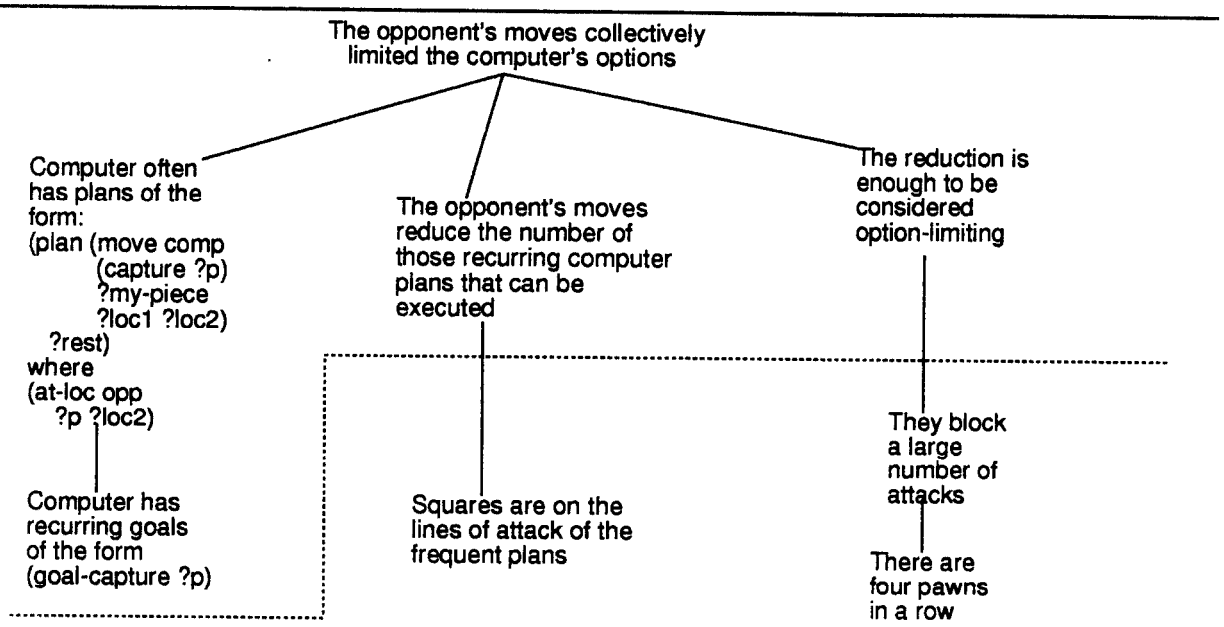


Figure 7.2: Explanation of pawn line defenses

plans for goals that currently exist given the state of the board. CASTLE's specifications for its planning components must include the notion that plans either satisfy existing goals, or they satisfy one of a class of goals that are likely to arise in the future.

With this knowledge, CASTLE can explain that the opponent's moves in setting up a pawn line serve the goal of blocking moves by all of CASTLE's pieces to the other side of the board, and does so in anticipation of CASTLE's trying to move its pieces there. This explanation is shown in figure 7.2. Given such an explanation, CASTLE should be able to learn that a pawn line with 4 pawns in a row is an effective means of limiting opponent options.

We have been discussing the establishment of a pawn-line defense as a plan for limiting opponent options. In fact, it reflects an entirely different approach to self-defense from those we have discussed previously. We have described CASTLE's approach to defending itself against threats as the "firefighting" paradigm (in section 6.4), by which CASTLE tries to anticipate specific types of threats and respond to instances as they arise. There is a contrasting approach to defense, however, which is to establish a specific defensive posture designed to eliminate all potential threats in a given class. We have previously called this the "Great Wall of China" approach [Birnbaum *et al.*, 1990]. Once CASTLE established a pawn-line defense, it could relax its vigilance in detecting plans against the safe area of the board, because the need for "firefighting" will have hopefully been obviated. This would require that CASTLE be able to represent beliefs such as "this region of the board is safe" and use them to determine what needs to be done in the future.

As with buying time, there are several issues regarding lines of defense that do not arise in chess but would in other domains. One of these is that in a limited information domain, a line of pieces could be used to limit the opponent's perception of CASTLE's resources.

This application of lines of defense could be termed a "smokescreen," and in such a domain CASTLE would need a representation of the goal of limiting the opponent's information, and how this goal effects the opponent's abilities to plan, to learn this use of lines of defense.

7.2.1 Model enhancements seen in pawn line defenses

The following model enhancements have come up in discussing how CASTLE could learn the concept of pawn line defenses:

- Expect to be able to formulate effective plans
- Failure to formulate plans can be due to intentional opponent actions

These two notions augment the idea of *mobility* that we discussed in the pin example (section 6.6). The mobility expectation monitored the ability of each piece to move, and limitations of mobility were attributed to the opponent's intentional actions. The new notion regards the system's overall ability to execute the plans it formulates, and failures to do so can also be attributed to intentional actions of the opponent.

- Goals can apply to classes of pieces and moves

This concept, like the previous two, was used here to extend the notion of limiting piece mobility to the more general idea of limiting overall opponent options. However, it potentially applies to all other goals as well.

- Planning in advance for common goals

Many plans, among them pawn line defenses, must be established before their exact purpose is clear. The system should be able to reason about possible uses of plans that are not yet clear.

- Represent and utilize beliefs about safe board regions

This is necessary in order to have the system's decision-making reflect pawn line defenses that have been established.

The elaboration of these concepts would allow CASTLE to learn about pawn line defenses, and also about a variety of other concepts. These notions comprise the machinery needed to learn other plans to limit overall player options, such as *midboard domination,* as well as any plans that are started in advance of their applicability.

7.3 Learning to sacrifice

As a final example, consider the concept of a *sacrifice*. An action is taken that appears to present the other player with an opportunity, but the opportunity in fact allows the first player to gain even more subsequently. Examples of this are very common in sports games, such as *bunting* in baseball, in which a hitter hits the ball in a way that he can easily be tagged out, in order to allow another runner to advance without fear of being tagged.

In chess the notion of a sacrifice arises when a player moves a piece to a location in which it can be captured, when it would be advantageous to the player to have the piece captured. The most obvious such tactic is when there is another piece that can capture the attacker after the sacrificed piece is taken, in a way that the opponent cannot easily see. For example, many players have a hard time seeing captures that can be made by long moves through crowded board areas. Moving a piece to a location which is guarded by a piece on the other end of the board, through a crowded board area, may trick the opponent into capturing the piece and being subsequently captured himself.

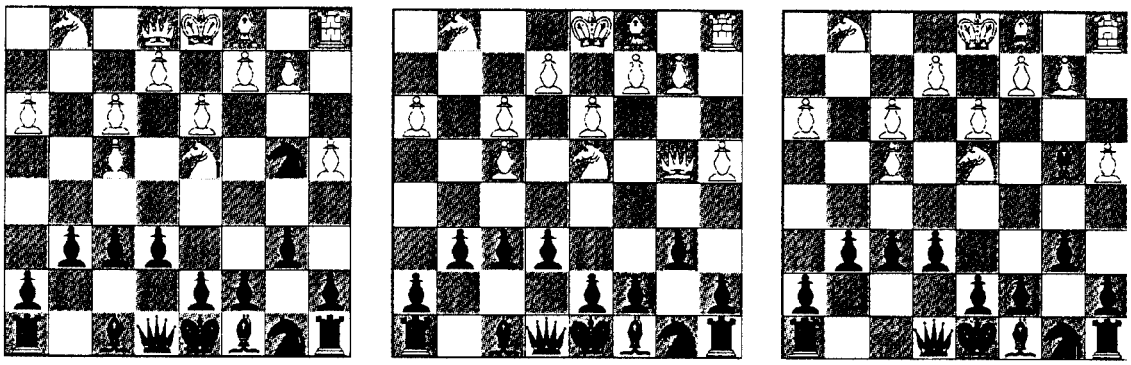
The first question we must ask is what situation could arise in which CASTLE could learn about sacrifices. In most of the examples discussed so far, CASTLE learns strategies that were employed by the opponent. It's not clear how this could happen here, however, since, CASTLE's threat detection and perception components aren't susceptible to distractions such as complex board layouts.[1]

Another possibility is that CASTLE notice in the opponent's behavior that he would fall prey to a particular strategy. Suppose CASTLE moves a piece to a position which is vulnerable to attack but is guarded by another piece on the other side of the board. CASTLE would assume that the opponent would see the deterrent, and thus would not make the attack, but if the deterring piece were far away, or if the line of attack were hard to follow, the opponent may capture CASTLE's piece anyway and subsequently be captured himself. CASTLE's expectation that the opponent would make some other move would fail, and the system would have to diagnose why the opponent made an unexpected move.

The first model enhancement that is needed in this example is that CASTLE be able to reason about captures that can be made but which are not advantageous to make. One instance of this idea that applies in our example is the notion of deterrence [Collins and Birnbaum, 1988a; Collins and Birnbaum, 1988b]. We saw previously that CASTLE expects the opponent to make the best available move on its next turn, and as such does not take into account any deterrents that may prevent the opponent from capturing a piece. Furthermore, CASTLE does not have a counterplanning method for establishing deterrents in the first place. Adding such a counterplanning rule would enable CASTLE to establish deterrents to guard its pieces, and this could be coupled with the notion of advance planning that we saw previously (section 7.2) to enable CASTLE to establish a deterrent prior to moving a piece to a dangerous location. Its expectation generation rule would then be modified to expect the opponent to make the best possible attack that is not guarded in some way.

With this enhancement, CASTLE would not expect the opponent to capture its knight with his queen, because the knight is guarded by the bishop and because the queen is more

[1] Although they probably would be in a more complex and realistic architecture.



(a) Computer (black) knight vulnerable to attack by queen

(b) Opponent captures the computer's knight

(c) and computer responds by capturing the queen

Figure 7.3: *Sacrifice:* Opponent (white) to move

valuable than the knight. This expectation would fail in board (b) when the opponent captures CASTLE's knight. The diagnosis engine would assign blame for the failure to the portion of the expectation rule that concluded that the capture was not in the opponent's best interests. This belief is supported by the following: *the bishop guards the knight,* and *the queen is more valuable than the knight,* and *the opponent isn't being coerced to make a harmful move.* This last assumption is this example's second enhancement to CASTLE's model, and it encodes the fact that people do not do things against their best interests unless they're being coerced. If coercion is considered to be related to forced-outcome strategies, this assumption also provides a link to the forced-outcome strategy component, and will enable CASTLE to learn a forced-outcome strategy for the sacrifice.

Of course, the sacrifice isn't really a forced-outcome strategy, because the opponent isn't forced to capture the sacrificed piece. On the contrary, if the opponent is being careful he will not make the move. The third model enhancement in this example is to loosen the constraints on the forced-outcome strategies to be *"likely-outcome strategies,"* and have them generate an associated likelihood of outcome that is used in evaluating them for execution. The strategies we saw earlier (fork, simultaneous attacks, pin, etc) would have a likelihood of 100% due to their being forced-outcome, but a strategy like the sacrifice would have a lower likelihood of success. CASTLE's meta-rules for applying strategies would then have to be modified to take the likelihoods of success into account when selecting a plan to execute. Given this enhancement, the sacrifice can be seen as a likely-outcome strategy in cases where the opponent is very likely to oversee the deterrent.

Once CASTLE's diagnosis engine has determined that there was in fact an unintended strategy coercing the opponent to make a bad move, it has to explain why the moves that it made were a likely-outcome strategy. CASTLE's explanatory model must therefore be extended to include the notion of likely outcomes. It also has to include a model of perceptual difficulties and distractions, to explain why the opponent didn't see the deterrent. These extensions to the explanatory model would allow CASTLE to construct a new strategy rule

for sacrificing.

While we have been discussing the use of sacrifices to capture enemy pieces, there are also other uses to which this strategy can be put. One is when a player wants to execute a plan that requires that an opponent piece be moved from its current location. While CASTLE would currently not consider such a plan to be viable, the planner could be extended to instead attempt to satisfy the subgoal of forcing the opponent piece to move. A sacrifice could be a method for achieving this, provided that the benefit was worth the loss of the sacrificed piece. This could result in a variant on *discovered attacks,* in which the opponent is enticed to open up an attack on himself.

The sacrifice is very prevalent in other competitive domains. One instance is the strategy of *bunting* in baseball, as was mentioned before. The sacrifice is also common in limited-information games, such as war-games (and warfare), in which weak pieces are left open to attack with stronger pieces nearby ready to counterattack.

7.3.1 Model enhancements seen in sacrifices

The following model enhancements arose in discussing how CASTLE could learn the concept of sacrificing:

- Reasoning about viable but non-advantageous captures
- Expecting the opponent not to make harmful moves
- Likely-outcome strategies
- Reasoning about probabilistic outcomes

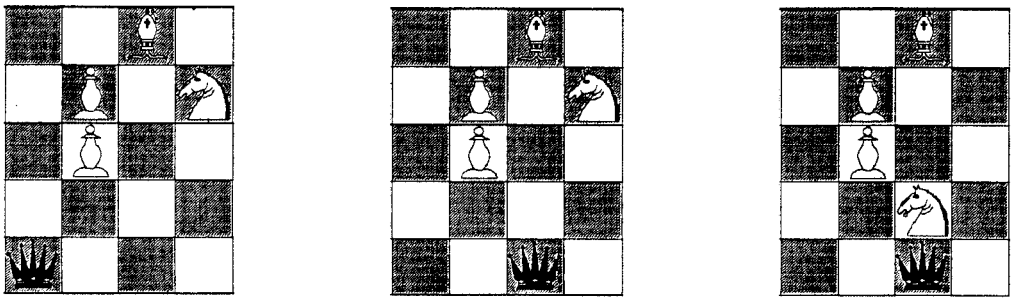
These are crucial for CASTLE to be able to construct plans that are likely to work but may not, and to reason about opponent plans of this sort.

- Explaining perceptual difficulties and distractions

This is necessary for CASTLE to understand and explain why the opponent didn't see the deterrent.

- Subgoaling on forcing the opponent to move a piece

This is a planning issue whose underlying idea is reasoning about the opponent's thought processes.



(a) (b) (c)

Figure 7.4: *Interposition* example: Computer (black) to move

7.4 Interposition yet again

We've now seen a number of enhancements to CASTLE's planning and explanatory models that would allow it to learn more complex concepts in a wider variety of situations. With these enhancements in mind, let's go back to our primary example, learning to interpose pieces, and see how this learning process can be improved.

When CASTLE learned the idea of counterplanning by *interposition* in chapter 2, it missed an essential part of why the moves made in figure 7.4 are an effective counterplan. CASTLE's learned counterplanning rule said roughly *To counterplan against an attack, move a piece to a square along the line of attack, providing that the attack isn't being made by a knight.* This rule ignores the fact that such a move is enabling an attack on the interposed piece. In the example this was not an issue, because the opponent's pawn served as a deterrent against such an attack, but this fact was not represented in the rule that CASTLE learned. A consequence of this is that CASTLE interposes pieces indiscriminately, without making sure that they are defended in their new positions.

It certainly is possible to extend CASTLE's specification for counterplans to specify that they cannot enable higher-valued attacks for the opponent in the course of blocking the threat, but this is too narrow a view of what could or should be avoided. What if the counterplan doesn't enable another attack, but instead puts the player in a significantly weaker position in some other way?

This problem can be solved if the system has explicit knowledge of *consequences,* which we alluded to above in section 7.1. Suppose the counterplanning specification said that *the counterplanning component should generate a plan that disables a given threat, as long as the consequence of the counterplan is better than the consequence of the threat being carried out.* The knight interposition in our example is a good counterplan, because the consequence is that the queen will not attack the knight for fear of being subsequently captured by the pawn. Assuming that the explanation rules for consequences are properly written, the supporting beliefs for the move having a better consequence would include the fact that the pawn prevents the queen from capturing the knight and continuing to capture the bishop.

Another, more technical, issue is the need to improve CASTLE's ability to explain

negations. As we discussed in section 4.7, CASTLE currently explains negations by checking if it can find a way to make the negated expression true, and if it can't then it assumes that the expression is necessarily false [Hewitt, 1969; Roussel, 1975; Reiter, 1978]. In such a case, however, there is no specific logical support for the truth of the negation, in that there is no belief in CASTLE's memory that supports it. Because of this, EBL is unable to include more specific expressions than the negation itself. In our current example, however, we would like CASTLE to prove that there is no bad consequence of the interposition, and to see that this is supported by the fact that there is a deterrent piece to prevent an attack on the interposed piece. To do this CASTLE's explanatory theory must be extended to include rules for proving negations. On top of this, even when a negation is proven by failure, EBL should be able to determine that the negation is true because a set of other negations are themselves true, namely the negations of the antecedents that failed to prove the sub-expression. For example, a belief that there are no bad consequences of a move could be supported by the belief that there are no opponent pieces that can attack the piece at its new location. In this case EBL should include the more specific negation in the learned rule.

7.5 Discussion

This chapter has explored the power that CASTLE would gain by incorporating additional model knowledge. The following model enhancements arose in our discussion of complex learning examples:

1. Persistence and reestablishment of threats
2. Desirable and undesirable consequences of actions
    - Reasoning about viable but non-advantageous captures
    - Expect the opponent not to make moves that end up being harmful
3. Formulating effective plans
    - Expect to formulate effective plans
    - Failure to do so can be due to intentional opponent actions
4. Abstract reasoning about pieces and locations
    - Goals can apply to classes of pieces and moves
    - Planning in advance for common goals
    - Represent and utilize beliefs about safe board regions
5. Likelihood and probability
    - Likely-outcome strategies (as opposed to forced-outcome)

- Reasoning about probabilistic outcomes
- In other domains: Likelihood of serendipitous events and unforseeable consequences

6. Reasoning about perceptual difficulties and distractions
7. Limited computational resources
8. Subgoaling on forcing the opponent to move a piece
9. Explaining why something isn't the case

These enhancements would enable CASTLE to learn substantially more complex concepts, using the learning framework discussed earlier in the thesis.

A great deal of future research is clearly involved in incorporating these model enhancements into CASTLE. The representational issues involved in several of them are research endeavors in their own right. Additionally there are difficulties involved in using models such as these in learning. These issues nonwithstanding, we have seen in this chapter that CASTLE's framework should in principle be able to incorporate them and utilize them in learning.

Chapter 8

Related work: Comparison and discussion

This chapter will address how previous research has dealt with the issues we have been discussing in this thesis. The first section briefly summarizes some of the most relevant previous efforts. The second section discusses how CASTLE's approach to determining what to learn contrasts with that taken by other models. The third section takes a step back, and discusses how other approaches to learning to plan compare to CASTLE's in more general terms.

We will be focusing our discussion on several previous efforts, including PRODIGY, SOAR, and CHEF, which we describe briefly in section 8.1. The use of these systems for comparison purposes should not in any way be taken as particular criticism of these systems. Indeed, it reflects the belief that they constitute the best approaches to these issues to date.

8.1 Summary of related research programs

PRODIGY [Minton, 1988a; Minton *et al.*, 1989; Carbonell *et al.*, 1990] is a research project addressing a wide variety of issues in learning and planning within a comprehensive framework. The core system is based on a means-ends problem solver that constructs plans from sequences of STRIPS-like operators. At any point that the problem solver must make a decision (either which node to expand, which goal to work on, which operator to apply, or which objects to use) it invokes *search control* rules that make suggestions as to which option to choose. We will be looking primarily at the PRODIGY/EBL subsystem [Minton, 1988a; Minton, 1990] which uses explanation-based learning to acquire new search control rules, based on analyses of traces of the system's problem-solving. Other subsystems of PRODIGY that are relevant to our discussion include STATIC [Etzioni, 1990; Etzioni, 1991], which derives search control rules from a static analysis of the system's domain theory, ANALOGY [Carbonell and Veloso, 1988], which uses successful problem solving experiences to guide future problem solving, and ALPINE [Knoblock, 1990; Knoblock, 1991], which learns to transform problems into more abstract problems, presumably easier to solve, and the solutions to which can then be transformed back into solutions for the original problems.

SOAR [Laird and Newell, 1983; Laird *et al.*, 1986b; Newell, 1990] is a problem solver that expresses all aspects of decision-making in terms of *problem spaces.* SOAR uses heuristic search to find a sequence of operators (i.e., a plan) that can transform an initial state into a solution state. Whenever a decision-point is reached (e.g., operator selection, operator implementation, evaluation), SOAR attempts to use production rules to determine a course of action (in a manner somewhat similar to PRODIGY's search control rules). In the absence of rules that completely specify the answer (a situation called an "impasse"), SOAR does not search through the various possibilities directly. Rather, it uses a process called *universal subgoaling* [Laird, 1983; Laird *et al.*, 1986b] to design a new problem space for solving the sub-problem. The subgoal is solved by recursively applying heuristic search of the new problem space, and the answer is passed to the higher-level search process, which continues from there. At that point an explanation-based learning procedure called *chunking* is used to construct a new production rule that encapsulates the results of the subgoal search [Laird *et al.*, 1986a]. In other words, SOAR uses search to solve subgoals for which it does not have production rules, and then learns new rules so that the search does not need to be duplicated in later problem solving episodes.

CHEF [Hammond, 1986a; Hammond, 1989] is a case-based planner that constructs cooking plans by adapting old plans from a case library. The adaptation process first attempts to anticipate problems and correct them. The system then simulates the plans and learns from any failures that arise [Hammond, 1986b]. In particular, the system learns new plans to retrieve in future problem solving, it learns to anticipate failures, and it learns to repair failures that have arisen.

8.2 Determining what to learn

How do other programs determine what to learn? In most cases there is only one type of concept to learn, so the determination is made in a fixed way by the program's control structure. CBG [Minton, 1984], for instance, learns a forced-move sequence schema whenever it loses a game. Theo-Agent [Mitchell, 1990] similarly learns a stimulus-response rule whenever its planner produces a new plan.

CHEF uses a fixed scheme of this sort to learn more than one type of construct. Whenever plan transformation is complete, CHEF stores the resulting plan back into its case library. Whenever a bug is found during simulation that was not anticipated earlier, the system learns a bug anticipator rule. Finally, whenever a bug is repaired, the system learns a generalized bug repair rule. The first thing to note is that the three types of things that CHEF can learn correspond to three components of a case-based planner, namely case retrieval, indexing, and adaptation (respectively). CHEF's relatively simple approach to the task of determining which of these three things to learn is highly effective for a number of reasons. First, there are only three types of things to learn, and they are highly distinctive. There is not much chance of confusing which category is applicable, as there might be if more subtle distinctions (e.g., between different types of indices relevant to different aspects of the planning process) were considered. Second, CHEF's plan simulator returns a complete causal analysis of the bugs that arise, so there is no need for complex inference to determine which of its components

needs to be repaired, as there might be if CHEF were to learn in response to problems that arose later in time during plan execution. If either of these two conditions did not hold, a more complex approach would probably be needed.

SOAR uses a similar scheme to learn chunks (represented as production rules) after resolving impasses. In SOAR's case, however, the chunks fit a variety of purposes because they are learned in response to impasses in a variety of stages of decision-making. SOAR's impasses thus serve a purpose similar to CASTLE's expectation failures. In the absence of explicit reasoning about what to learn, SOAR relies on its knowledge representations to direct the learning of new rules. In other words, decisions about when and what to learn are tied directly to the representation of problems and sub-problems, and thus to the expressiveness of the system's vocabulary. The lack of explicit control over the learning process often leads to the creation of undesirable chunks, and the only solution within the SOAR philosophy has been to limit the expressiveness of the system's vocabulary [Tambe and Rosenbloom, 1988a; Tambe and Rosenbloom, 1988b]. In contrast, CASTLE's approach is to reason explicitly about what rules should be learned, in a sense taking the opposite approach of *extending* the system's vocabulary. While we have not addressed the issue of the computation time required by learned rules, such constraints can and should be expressed in the system's self-model, to be used in diagnosis and rule repair. It may be possible to extend SOAR to carry out explicit reasoning of this sort as another heuristic search problem, but this has not yet been investigated.[1]

PRODIGY is probably the closest in spirit to CASTLE, in that it performs dynamic inference to determine what to learn in a given situation. The system maintains a trace of its problem-solving behavior, and examines this trace to find concepts to learn. The "concepts" are search-control decisions that should have been made at that point in the planner's execution. In particular, the system has a set of *example recognizers* that look for conditions indicating that a node in the problem-solving trace is an instance of a particular type of concept. For example, if a node refers to a choice of an operator to apply, and the choice ended up failing, an example recognizer will signal the node as an example of an operator that should not have been chosen. This node, along with the concept it is believed to be an instance of, is passed to an explanation-based learning mechanism which constructs a new search control rule.

This process appears to satisfy the criteria discussed in this thesis for analytical approaches to determining what to learn. The system has a set of concepts that it can learn (corresponding to the types of search control rules it uses), and has explicit methods for finding examples of each. This "pattern-matching" approach to determining what to learn seems quite appealing, because it appears to require fairly little inference (certainly less than CASTLE's diagnostic approach), and because it works using clear declarative knowledge about each of its types of rules.

In practice, however, assessment of PRODIGY's approach is more complicated, in part

[1] At first glance this idea would seem to be consistent with SOAR's methodology of expressing all aspects of problem-solving in terms of search. Such an approach would, however, raise the question of the source of SOAR's leverage in such a model, SOAR's architecture or the system's representational models. For this reason the approach might be said to undercut SOAR's goal of achieving intelligent behavior through a uniform architecture.

```
(defun sel-fails (n)
  (if (and (not (upper-node-failure-proved-so-ignore-lower-failure n))
           (not (node-alternatives n))
           (node-node-level-failure n)
           (or (not (node-added-after-restart n))
               (and (worth-gi-learning)
                    (above-is-supporting-failure n))))
      (let ((h (make-hist n)))
        (cond ((node-reset-alt n)
               (setf (get h 'dont-make-rule) 'is-reset)))
        (list h))))
```

Figure 8.1: PRODIGY selection heuristic for node failure

because the system additionally requires the use of procedurally encoded *example selection heuristics* to "eliminate uninteresting examples" [Minton, 1988a, sec. 4.2]. While the rationale to these heuristics sounds innocuous, they in fact embody quite sophisticated reasoning. The public release of PRODIGY 2.0 [Minton *et al.*, 1989] contains 35 of these heuristic LISP functions. Fourteen of them select interesting examples for particular rule types (one per rule type). Fourteen other functions are used to search for specific conditions that make learning a particular type of rule beneficial. The seven remaining functions are used for determining "interestingness" of a number of rule types. All of these are separate from the initial example recognition process. Additionally, these functions maintain a history of the process of searching for examples to learn that is used in subsequent example selection.

As a quick example, one of PRODIGY's sets of search control rules is to predict that expanding a particular node in the search tree will be ineffective. Examples of `node-fails` are in principle quite simple to recognize—any node that fails (that was not predicted to fail) is a candidate for learning. After this recognition is made, PRODIGY uses the selection heuristic shown in figure 8.1 to determine whether in fact to learn from the example. The functions `upper-node-failure-proved-so-ignore-lower-failure` and `above-is-supporting-failure` are used only for selecting node failures, while `worth-gi-learning` is used in heuristics for a number of rule types (mostly interaction predictions). The end of the function provides an example of how the selection heuristics maintain a history of their traversal of the problem-solving trace.

All in all, it seems clear that PRODIGY's determination of what to learn is far more complex than the simple recognition of patterns in the problem-solving trace. The process employs a great deal of heuristic information about what will constitute a good search control rule, and this information enables PRODIGY to learn effectively. Were it not for these heuristics, the learning process would spend an inordinate amount of time learning

pointless search control rules.[2]

The point, however, is not that PRODIGY is in any way wrong to carry out complex reasoning of this sort to determine what to learn—indeed, the claim of this thesis is that such inference is necessary. Rather, the point is that while PRODIGY relegates the complexities of example selection to procedurally-encoded heuristics, the research presented in this thesis attempts to make such information explicit. CASTLE's processes, while algorithmically fairly simple, are designed to enable the system to take into account any knowledge of this sort that is provided. By examining the knowledge that is provided in the system's justification structures and explanatory models, it can determine what lessons to learn at any point. An area of future research would be to apply CASTLE's approach to a search-based problem-solver like PRODIGY. The goal would be an explicit representation of the issues that PRODIGY encodes in its selection heuristics about what types of rules should be learned in different situations. An explicit, conceptual theory of the knowledge in PRODIGY's selection heuristics would constitute a more complete theory of learning search-control rules. A rudimentary theory of this sort has been developed in the STATIC subsystem, which reasons from first principles (not from examples) about what search control rules are expected to be effective. This thesis may provide a framework for knowledge of this type to be used for learning from experience.

8.3 Learning planning knowledge

Beyond the issue of determining what to learn, this thesis has attempted to present a uniform, knowledge-based, approach to learning to plan. The aim has been to view planning as composed of semantically meaningful subtasks, and to approach learning to plan in terms of these subtasks. This aim is met by the component-based modeling approach which CASTLE implements, in which the planner is built out of components which carry out specific subtasks. This section will contrast this approach with previous approaches to learning to plan, and discuss the contributions CASTLE makes to the state of the art.

PRODIGY (and similarly SOAR) learns rules for carrying out subtasks, but these subtasks exist at the more primitive level of the underlying search process, such as choosing operators, binding variables, etc. While this enables PRODIGY to learn to plan more effectively, it is limited to domains which are best handled by a search engine. The level of this model is entirely different from that embodied in CASTLE. PRODIGY is modeled as an inference engine, not as a planner. The system's learning is thus not sensitive to the type of task it is carrying out, whether planning, design, scheduling, or any other. For this reason, the rules that are learned can only reflect concepts in the domain in an indirect fashion, e.g., when to choose a particular operator, and not directly. More importantly, they can reflect concepts about the task itself (i.e., planning in the domain) only indirectly. For example, PRODIGY's approach cannot be straightforwardly extended to learn new domain operators. It is very difficult to envision how PRODIGY could learn about *discovered attacks,* for example, or in fact how they could even be represented in PRODIGY's framework.

[2]Thanks to Steve Minton for personal communication clarifying the issues involved in PRODIGY's selection heuristics.

Of course, it is possible in principle to express many concepts to be learned as search-control concepts. The notion of counterplanning by interposition, for instance, could be expressed as *"when choosing an operator, if the opponent has an opportunity to capture a piece of mine, and another piece can interpose, consider the operator that carries out the interposition."* While a rule like this certainly would do the job, it is much less straightforward than learning specifically for the task of counterplanning.

CASTLE might be viewed as a radical reformulation of PRODIGY from a search engine to a component-based planner. Both have rule sets for carrying out subtasks, and both use explanation-based learning for learning new component rules. In this light, one of CASTLE's key contributions is relating the components—and thus the learning process—to subtasks of decision-making in the domain, as opposed to search engine subtasks.

In contrast to SOAR and PRODIGY, one system that does learn concepts for semantically meaningful subtasks of the planning process is CHEF. CHEF's case-based approach to planning is broken into the subtasks of case retrieval, plan adaptation, bug anticipation, and bug repair, and the system uses this decomposition to tailor its learning to the subtask being improved. Preliminary research in applying CASTLE's approach to case-based planning appears promising [Birnbaum *et al.*, 1991], but future research is needed before any success can be measured.

Most other research in learning to plan falls into two broad categories. One is to specifically address learning within the context of classical linear or non-linear planning (e.g., [Chien, 1990; DeJong *et al.*, 1993]). Such systems start with a planning method, and incorporate a simplification that makes planning more efficient. The system then learns from successes or failures, and refines its application of the simplification. The relationship of these approaches to the research presented in this thesis is similar to that discussed above for CHEF.

The second common approach is to generalize and store planning results in the form of macro-operators or reaction rules, that can be used in the future to make planning more efficient (e.g., [Fikes *et al.*, 1972; Mitchell, 1990]). This approach is orthogonal to CASTLE's, in that the learning is not in response to a performance failure, other than the inability to retrieve the planning result before planning [Birnbaum *et al.*, 1991].

A number of ongoing research projects are similar to CASTLE in a number of ways. One approach being taken employs *knowledge acquisition goals,* which are explicit representations of goals that the system has for learning [Hunter, 1989; Ram and Hunter, 1992; Leake and Ram, 1993]. Another important concept being developed in this work is the *meta-explanation* of desired computer performance, using constructs called *meta-XP's,* for the purpose of selecting a learning strategy [Cox and Ram, 1992; Ram and Cox, 1993]. Another approach being taken describes a knowledge-based system as a set of generic tasks [Chandrasekaran, 1987], and uses task descriptions to generate error candidates in response to a mistake [Weintraub and Bylander, 1991]. These projects are close in spirit to CASTLE, but are different enough in aim that it would appear quite fruitful to attempt to integrate the approaches.

8.4 Conclusions

We have seen that CASTLE's approach to modeling planning in terms of semantically meaningful components makes two primary contributions to the field. One is the ability to reason dynamically about what to learn. This is made possible because of the system's ability to reason explicitly about the tasks and subtasks of its architecture. Previous research has determined what to learn in a fixed way, either by wiring the determination into the control structure of the program, or by providing complex *ad hoc* methods for making the determination.

The other contribution CASTLE makes is in learning rules that relate to subtasks of the planning process, as opposed to subtasks of an *implementation* of planning in terms of more primitive computational processes such as search. This allows CASTLE's learner to focus on issues that are relevant to learning for particular cognitive tasks.

Recent trends in A.I. literature, particularly in the area of machine learning, have favored quantitative methods of evaluating contributions and comparing approaches (e.g., [Minton, 1988b; Mooney *et al.*, 1989]). Such quantitative methods seem appropriate for comparing systems for which the problem is clearly specified, a number of alternative solutions have been proposed, and their performance along relevant dimensions is easily measured. It should be clear from our discussion in this chapter, however, that this is hardly the case in the problem of learning to plan, considered in full. Issues such as what should be learned, what flexibility is desired, what aspects of the learning process can be predetermined by a programmer, and indeed what it means to model a planner, are all far from clearly specified.

So how then do we evaluate CASTLE's contributions? The bottom line has to be *"Can* CASTLE *carry out learning tasks that could not previously be carried out?"* The point of this chapter has been to argue that the answer is "yes."

Chapter 9

Learning, modeling, and intelligence

The goal of the research presented in this thesis has been to develop a model of how to learn lessons from experiences in a *flexible* manner. Chapter 1 presented a scenario in which a hypothetical agent burned rice pilaf while cleaning another room, and was able to learn a wide variety of lessons from the experience. The insight that we drew from analyzing this example was that lessons to learn correspond to cognitive tasks whose performance should be improved. Generating lessons to learn thus involves determining which cognitive tasks should have their behavior improved, and then generating an improvement for each one.

The rest of the thesis has discussed how this notion can be incorporated in a computer planning system, to enable the system to learn flexibly from expectation failures. In the approach presented and implemented in CASTLE, cognitive tasks are embodied in explicit and distinct *components* of the system's architecture. The first step in learning, determining the cognitive tasks whose performance should be improved, is instantiated by a diagnosis module that searches for the components responsible for an expectation failure. This search utilizes justification structures, which encode the relationship between the system's decision-making architecture and its actions and beliefs. Learning the lessons for each faulty component is implemented by an explanation-based learning module, using an explicitly-represented model of the cognitive tasks that correspond to the components.

Two questions must be answered in evaluating the research that has been presented:

- How well does the approach capture the desired behavior?
- How well does the implementation demonstrate the approach?

This chapter will attempt to answer these two questions, and will close with a discussion of the implications of this work for machine learning and for artificial intelligence in general.

9.1 Rice Pilaf revisited

Chapter 1 introduced a situation in which an agent plans to cook rice pilaf while cleaning another room. The rice boils over and is ruined, and our agent wants to learn from the experience so that similar problems won't occur in the future. Section 1.1 discussed the

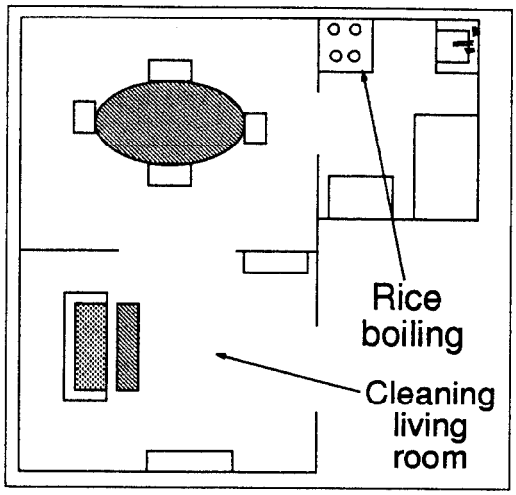


Figure 9.1: Cooking rice pilaf while cleaning

following lessons that could be learned, each of which corresponds to a cognitive task implicated in the failure:

1. Listen harder for bubbling or the lid bouncing: *Perceptual tuning*
2. Adjust the flame more carefully: *Plan step elaboration*
3. Leave the lid ajar: *Plan step elaboration*
4. Don't do loud things while cooking: *Scheduling/interleaving*
5. Stay in the kitchen while cooking: *Scheduling/interleaving*
6. Don't cook over high flame when busy: *Scheduling/interleaving*

The agent in the example approached learning from the failure by performing self-diagnosis, in the form of an internal dialogue aimed at determining the origin of the problem:

1. *Why was the rice ruined?*

   The rice boiled over.

2. *Could I have done something differently to prevent it?*

   Yes, I could have lowered the flame or uncovered the pot.

3. *Without doing this, could I have prevented it from boiling over?*

   Yes, if I had heard it.

4. *Could I have heard it boiling over?*

   Maybe I could have, if I'd paid more attention.

5. *Why couldn't I hear it boiling over?*

   I was using the vacuum in the living room.

6. *Could I have planned things differently to enable me to hear?*

   Yes, I could have delayed vacuuming or stopped every few minutes to check the rice.

7. *Why didn't I?*

   I didn't think about the inability to hear from the other room.

This dialogue led the agent to discover several of its cognitive tasks that could be improved by learning from the failure, corresponding to the lessons listed above.

How well would CASTLE handle this example? The first step is to formulate the agent's decision-making in terms of components that could carry it out. When presented with two conjunctive goals, cooking the rice and cleaning the living room, the agent might first construct independent plans for each of them, and then merge them into a single plan. Let's suppose that the agent first formulates skeletal plans, and then elaborates them only as much as necessary at plan-time, postponing complete elaboration until execution-time. We can model this approach with two components (each of which presumably have a number of sub-components), one for constructing skeletal plans and one for elaborating plan steps as necessary.

The next step is to merge the two plans into one master plan. An initial merging is first carried out to produce a plan that contains the steps of both sub-plans. Then the agent tries to predict and repair potential interactions between steps that could cause problems. We model this with three components, one for the initial merge, one for predicting interactions, and one for repairing them.[1]

Lastly, the agent must be able to detect and handle problems that arise during execution. This would be unnecessary were the agent truly able to predict all problems at plan-time, but of course this is not truly feasible, so our agent (and human beings in general) must trust his ability to catch and handle problems at execution-time that were not predicted and handled by the planner. We can model this ability using components that are invoked during plan execution. One of these components might tune the agent's perceptual apparatus to attend to particularly important perceptual inputs. Another component might interpret the perceived sounds to extract meaningful information. A third component might respond to this information as appropriate. These components are invoked along with the components that are used to execute the particular plan steps.[2] The agent trusts that these components will be able to detect and handle any problems that arise during execution.

Given such an architecture, figure 9.2 depicts a possible justification structure for the agent's belief that he would successfully carry out both goals. The numbers that label some of the beliefs in the justification structure correspond to questions in the agent's self-dialogue on page 180; the system's traversal of this justification structure should result in diagnosis

[1]The use of problem predictor rules and repair rules in this way is similar to many previous approaches to planning, e.g., [Sussman, 1974; Hammond, 1989].

[2]We envision components being used to translate plan steps into primitive actions [Firby, 1989; Simmons, 1991].

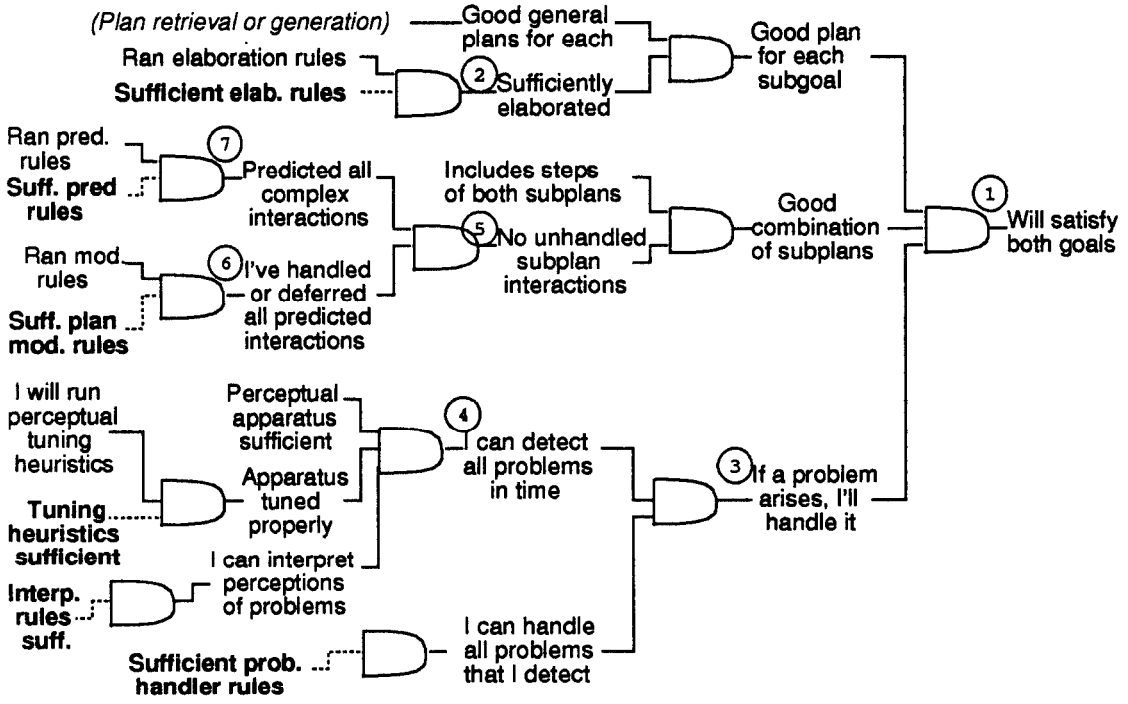


Figure 9.2: Justification structure for the rice pilaf example

queries similar to those in the dialogue. Of the six component completeness assumptions (shown in the figure in bold type), five of them could potentially be at fault in the example:

1. *Plan step elaboration:* Specify flame height or lid ajar
2. *Interaction prediction:* Predict problem hearing while vacuuming
3. *Plan modification:* Don't interleave, check periodically, etc.
4. *Perceptual tuning:* Listen harder for boiling over
5. *Perceptual interpretation:* Sizzling sounds indicate boiling over

Which of these faults should be repaired depends on a number of factors, such as CASTLE's perceptual abilities and the time constraints under which the system was operating. Once the set of components to repair has been selected, CASTLE should generate an explanation of the desired behavior of each one. One such explanation, for the interaction prediction component, would say roughly: *I should have detected an interaction between cooking the rice and vacuuming,* because *Cooking on the stove requires listening periodically for problems,* and *vacuuming makes it impossible to hear because it is so noisy.* Such an explanation might be generalized in a number of ways. One rule that could be learned from this explanation is: *There is a bad interaction between cooking on the stove and doing something noisy in another room, namely that it will be hard to hear how the cooking is going.* Such rules could then be added to the component for interaction prediction. Another rule might be: *There*

is a bad interaction between vacuuming and carrying out a task that requires listening. Yet another more general rule might be: *There is a bad interaction between loud tasks and tasks that require careful listening.* An agent learning from this explanation would have to reason about the various rules that can be learned, and the tradeoffs between their generality and their utility.

In sum, given the necessary decision-making architecture, modeled in terms of components, and the necessary retrospective diagnosis rules, component specifications, and explanatory knowledge, the learning processes that we have presented in this thesis appear capable of accounting for the diversity and flexibility of every day human learning.

9.2 CASTLE's limitations

There are a number of issues that arise in the rice pilaf example that have not arisen in chess examples, that reflect limitations in CASTLE's actual ability to handle the example. Two significant issues are:

1. Non-independent component faults

2. Choosing among repairs

The first problem, CASTLE's inability to handle non-independent faults, was discussed in section 4.8. The system is unable to diagnose and repair faults that arise from an interaction among components, rather than from an individual component. One interaction of this sort in the rice pilaf example is the interplay between the prediction/modification components, which predict interactions and modify the plan to handle them, and the tuning/interpretation/handler components, which detect and handle problems at execution-time. These components must be repaired jointly, within the context of the other components in the groups given, and not merely individually, since the agent's ability to carry out the tasks affect the plausibility of repairing other components in the same group. In other words, repairing the execution-time problem handler component is only plausible if the tuning and interpretation components can also be repaired sufficiently. Similarly, repairing the subplan interaction prediction component is only reasonable if the plan modification component can be repaired as well.

CASTLE can perform reasoning of this type when it is hard-wired into its models of the individual components. This clearly violates the principle of "no function in structure" [DeKleer and Brown, 1984; Davis, 1990] commonly held in model-based reasoning, which in our terms requires that the component models be given in a form that is independent of the properties of the overall system. While it is generally accepted that this principle is not always possible to achieve in practice, it is nonetheless fundamental to CASTLE's approach to diagnosis and repair that the component breakdown be as complete as possible. What CASTLE needs is a model of the interactions between the components, along with the desirable properties of these interactions. Such a model would allow the system to infer the relationships between the components in the rice pilaf example, and in general to reason

about interactions between components being repaired. This is a topic of ongoing research [Freed *et al.*, 1992].

Another aspect of the rice pilaf example that CASTLE is unable to handle is selecting among faults to repair. This kind of reasoning requires knowing how good a solution will result from each of the possible repairs, as well as the effect that each of the repairs will have on the rest of the system's operations. Sometimes more than one repair should be made, either redundantly or with different applicability conditions.

In our example, the choice of a good repair depends heavily on the agent's perceptual abilities and time constraints. If he were in principle able to hear cooking noises in the kitchen while vacuuming in the living room, then learning to attend more to his hearing will solve the problem without disrupting the plans in action. Alternatively, if the agent thinks that it's possible to adjust the flame on the stove carefully enough to keep the pot cooking well without boiling over, thereby obviating the need to listen, this is a good solution. In each case the agent needs knowledge of his abilities to determine which solution is feasible. Knowledge of time constraints also plays a role here. If time is not an issue, then simply avoiding the interleaving of cooking and cleaning in other rooms would solve the problem quite easily. On the other hand, if time is of the essence, another lesson must be learned instead.

In addition to taking all of these factors into account in deciding what to learn, the agent must also reason about which should be incorporated into learned concepts. For example, the agent may choose to learn lessons for both the time-critical and non-time-critical situations, and remember in the future to use the solution that best matches the active goals.

9.3 Implications of this work

The research presented in this thesis makes contributions to several areas of artificial intelligence.

9.3.1 Machine learning: What to learn

The primary contributions made by this thesis are in developing an approach to learning flexibly. The key problem that we have addressed is the question of determining what to learn. CASTLE's analysis of expectation failures and their associated justification structures enables it to determine a generalization task to pass to the EBL module. The dynamic generation of a learning task is necessary for incorporating learning algorithms into larger systems.

While this thesis has addressed the problem in terms of explanation-based learning, it can be applied to other learning methods as well. Regardless of the learning algorithm employed, CASTLE can determine the component whose behavior should be repaired, and a description of the desired behavior of that component. In CASTLE's current implementation, a specification of the component, along with the description of the desired behavior, is passed to the learner to use as an EBL target concept.

Suppose, however, that the task of a given component was carried out not by a set of rules, but rather by simply categorizing the inputs in some way. A component of this sort could be used for tasks such as perceptual interpretation or run-time opportunity recognition. If the component fails to classify something as a member of a category that in fact was a member, CASTLE would determine that the component was faulty and should have returned a positive answer. An inductive learning algorithm (e.g., [Quinlan, 1986; Utgoff, 1989]) could then be used to update the category description.

Suppose, alternatively, that one of CASTLE's components performed its task using case-based reasoning [Schank, 1982; Kolodner, 1987; Hammond, 1989; Riesbeck and Schank, 1989]. CASTLE would isolate the faulty component, and would determine its desired behavior. This could then be used as inputs to a case storage module that would augment the set of cases used by the component. A variant of this approach would be if the component performed its computation using traditional search techniques, but used past experiences to guide the search [Carbonell, 1986; Carbonell and Veloso, 1988]. The desired component behavior could again be passed to a case storage module, to be used subsequently to guide the search to the desired solution.

CASTLE's approach could even be applied to a component that used a neural network to perform its computation. The results of CASTLE's diagnosis module could be used as a desired input-output mapping [Hinton *et al.*, 1984; Rumelhart *et al.*, 1986; Shavlik and Towell, 1989] or as an example with desirable patterns [Rumelhart and Zipser, 1985; Pearlmutter and Hinton, 1986; Becker and Hinton, 1989].

In all of these cases we see that CASTLE's approach can be used to determine what learning task to pass to a given learning algorithm. Thus the approach allows any of these learning algorithms to be incorporated into a larger, more complex system. In fact, a system could be constructed with a hybrid architecture, performing various tasks with a mixture of rule sets, category descriptions, case-based reasoning, and neural networks (e.g., [Hendler, 1985; Golding and Rosenbloom, 1991]), and CASTLE's approach could enable learning to occur uniformly. Future research will determine how well this can be carried out in practice.

9.3.2 Model-based reasoning: Modeling planners

Most research programs in *model-based reasoning* have modeled and diagnosed circuits or other relatively simple devices. This thesis has contributed a model of a much more complex device—namely a planner—and has specified how to handle the representational and algorithmic issues that it presents.

There are two primary issues that must be addressed in applying model-based reasoning techniques to a planner: How the planner can be represented so as to be effectively reasoned about, and how adjustments to be planner can be characterized [Collins *et al.*, 1991a]. The first problem is addressed by decomposing the planner into components that are related to the cognitive tasks in which the planner engages. The second is addressed by representing assumptions about the completeness of the component rule sets, so that faults of these assumptions correspond to directives to augment the rule sets.

Other representational issues also arose and were discussed in chapter 4. Assumptions that are quantified pose problems for traditional model-based reasoning techniques, as

do assumptions that vary over time. CASTLE's approach incorporates solutions to these problems.

9.3.3 Diagnosis: Retrospective analysis

Another issue that arose in applying model-based reasoning to planning is the need to perform complex inference to determine the after-the-fact truth of a belief that was previously thought to be true. Section 4.4 discussed *retrospective belief testing*, the need to examine beliefs being diagnosed "retrospectively," applying knowledge that was not previously known to see if the belief should still be considered true.

In previous model-based reasoning research this problem didn't arise explicitly because testing circuit values explicitly determines the current value, not the previously expected value. This issue becomes more important when the beliefs being tested cannot be mapped easily onto an observable fact. In many cases it requires inference to determine what observation to make to test the belief. In other cases it will not be possible to test the belief directly, and inference must be carried out to determine a value to use. This issue seems likely to arise in other diagnosis domains, including diagnosis of complex devices that cannot be measured directly.

9.3.4 Planning: Architectures

The development of architectures for planning and decision-making has long been a central focus of AI research. The goals of these architectures are numerous, including complex planning, reactive execution, and adaptivity. This thesis presents an architecture that supports learning, and discusses an approach to modeling complex decision-making that supports adaptation in response to experience.

9.4 Conclusions: Self-knowledge and intelligence

The guiding theme throughout this thesis has been the use of self-knowledge in learning to plan. Several forms of self-knowledge have been delineated, including component specifications, justifications, and explanatory models, and learning algorithms have been developed and adapted to use this knowledge properly. The reification of this knowledge and methods for using it effectively form the bulk of the contributions made by the thesis.

The CASTLE system, while demonstrating the viability of the approach, is only a first step in implementing a system that learns using self-knowledge, and there are many areas of open research in extending the application of these ideas. One such area is more complex approaches to decision-making in planning, such as non-linear planning, case-based reasoning, or hierarchical planning, in which the approach consists of repeated application of a number of sub-processes. Another area is in more complex domains of application, such as robotic planning, complex route planning, or scheduling.

In a broader sense, the research suggests an agenda exploring the use of self-models in learning, planning, and understanding. The formulations of self-knowledge that are

useful in learning should also give leverage into planning, execution, knowledge acquisition, communication, understanding, and design. Research into some of these areas has already begun. In planning, self-models have been used for application of abstract strategies [Jones, 1992] and for meta-level reasoning during planning or plan execution [Kuokka, 1990]. In knowledge acquisition and communication, self-models and models of other agents involved in related tasks can be used to focus interactions. The work in this thesis provides a concrete example of how a computer system can usefully apply self-knowledge and introspection to carry out its tasks.

Appendix A

Planner rules

This appendix gives the rules that constitute CASTLE's planner. Each rule is given in its implementation format with annotations, and then in English in a somewhat more conceptual form. Each rule also includes notes as to where in the body of the thesis it appears (if anywhere).

The appendix is organized by cognitive task:

The execution of the planner takes place by cycling through the meta-rules in the order listed above. This appendix includes the rules that CASTLE learned:

Threat detection

Detection meta-rule

```
(def-rule focused-detection comp-threats                                  ; If possible threats
  (and (computed-possible-moves ?player                                   ; were recorded last
           (world-at-time (1- current-time)) ?prev-moves)                 ; time step,
       (is-set-of ?move-set (var ?move)                                   ; Construct a new set
         (and (or (in-set ?move ?prev-moves)                              ; that includes the old
                  (and (focus ?f-method ?player ?move                     ; set, and also apply
                              (world-at-time current-time))               ; the focus rules and
                       (is-threat-in-world ?t-detector ?t-type            ; then apply the threat
                          (player-opponent player)                        ; detectors within the
                          (world-at-time current-time))))                 ; focus constraints
              (move-doable ?move (world-at-time current-time)))))         ; such that all moves
=>                                                                        ; are feasible
  ((computed-possible-moves ?player current-time ?move-set)))             ; record the new set
```

IF There is a threat set from the previous turn
and all areas to focus on for new threats are determined
and all the new threats in those areas are generated
and all threats that can no longer be executed are filtered out

THEN Record the new set of possible threats as **old + new - disabled**

Appears on page 39.

Focus rule "new source"

```
(def-brule focus-new-source
    (focus focus-moved-piece ?player                                      ; Focus on
           (move ?player ?move-type ?piece ?loc1 ?loc2)                   ; threats
           (world-at-time ?time))                                         ; from the new
  <=                                                                      ; location of
    (move-to-make (move ?player ?prev-move-type ?piece ?old-loc ?loc1)    ; the recently
                  ?player ?goal (1- ?time)) )                             ; moved piece
```

TO COMPUTE: The location of potential new threats

DETERMINE: The new location of the most recently moved piece
and the class of threats from this location

Appears on page 41.

Focus rule "new target"

```
(def-brule focus-new-target
    (focus focus-moved-target ?player
           (move ?player (capture ?target) ?piece ?loc1 ?loc2)
           (world-at-time ?time))
  <=
    (move-to-make (move (player-opponent ?player) ?prev-move-type
                        ?target old-?loc ?loc2)
                  (player-opponent ?player) ?goal (1- ?time))  )
```

TO COMPUTE: The location of potential new threats

DETERMINE: The new location of the most recently moved piece and the class of threats from this location

Focus rule for "discovered attacks" (learned)

```
(def-brule learned-focus-method25
  (focus learned-focus-method25 ?player                          To find possibly
    (move ?player (capture ?taken-piece) ?taking-piece           enabled new threats
          (loc ?row1 ?col1) (loc ?row2 ?col2))                  to focus on
    (world-at-time ?time2))
<=
  (and (move-to-make (move ?other-player move-move ?interm-piece
                           (loc ?r-interm ?c-interm)             See where the most
                           (loc ?r-other ?c-other))              recently moved
           ?player ?goal ?time1)                                 piece used to be
       (not (= ?taking-piece knight))                            and, for non-knights
       (loc-on-line ?r-interm ?c-interm                          find moves through
                    ?row1 ?col1 ?row2 ?col2) ))                  the vacated square
```

TO COMPUTE: Areas in which moves may have been enabled

DETERMINE: Lines of attack through the most-recently vacated square for new moves by pieces other than knights

Appears on page 130.

Threat detector for rook captures

Note: As discussed in chapter 3, this rule is a sample threat detection rule. The rules that are actually used in practice are written in LISP for efficiency.

```
(def-brule threat-detect-rook
  (is-threat threat-detect-rook move-capture ?target-player        There is a threat
             (move ?player (capture ?target-piece) rook            by a player's rook
                   (loc ?row1 ?col1) (loc ?row2 ?col2))            against the enemy
             (world-at-time ?time))                                at a given time
<=                                                                 if the rook and the
  (and (or (= ?row1 ?row2) (= ?col1 ?col2))                        target are on the
       (no (and (loc-on-line ?interm-row ?interm-col               same row or column
                             ?row1 ?col1 ?row2 ?col2)              and if no square
                (at-loc ?either-player ?other-piece                on the line of
                        (loc ?interm-row ?interm-col) ?time)))))   attack is occupied
```

TO COMPUTE: Threats that can be made by a particular rook

DETERMINE: Opponent pieces which are on the same row or column such that no square between the two pieces is occupied

Appears on page 40.

Plan recognition

The plan recognition meta-rule

```
(def-rule comp-plan-recog comp-opp-plans                           If I can
  (recog-plan ?method opponent (world-at-time current-time)        recognize a
              ?plan ?goal)                                         possible
 =>                                                                opponent plan,
  ((possible-opp-plan (world-at-time current-time) ?plan ?goal)))  record it
```

IF There is a plan recognition method that can recognize a currently-possible opponent plan

THEN Record the possible opponent plan

Appears on page 43.

Recognizing captures

```
(def-brule recog-plan-1
  (recog-plan recog-move ?opp-player (world-at-time ?time)         To determine an
     (plan ?move done) (goal-capture ?piece ?loc (move ?move)))    opponent plan to
 <=                                                                capture a piece,
  (and (computed-possible-moves ?opp-player current-time ?ms)       get the set of
       (in-set (move ?opp-player (capture ?piece)                  detected threats
                     ?opp-piece ?opp-loc ?loc)                     and find such a
               ?ms)                                                threat to capture
       (= ?move                                                    the piece
         (move ?opp-player (capture ?piece) ?opp-piece ?opp-loc ?loc)) ))
```

TO COMPUTE: A plan the opponent can execute

DETERMINE: The set of threats that the opponent can make
and an immediate capture that is in the set

Appears on page 44.

Recognizing opponent strategies

```
(def-brule recog-plan-2
  (recog-plan recog-strategy ?opp-player                    To recognize a possible
      (world-at-time ?time) ?plan ?goal)                    opponent plan
 <=
  (and (in-set ?strategy-meth strategy-meths)               Find a strategy method
       (strategy ?strategy-meth ?opp-player                 that gives a plan for the
          (world-at-time current-time) ?goal ?plan) ))      opponent
```

TO COMPUTE: A plan the opponent can execute

DETERMINE: An offensive strategy script
which the opponent can currently apply

Appears on page 135.

Recognizing opponent option-limiting plans

```
(def-brule recog-plan-limit
  (recog-plan recog-limit ?opp-player (world-at-time ?time)   To determine an
     ?plan (goal-limit ?piece ?loc ?info))                    opponent plan to limit
 <=                                                           the system's options,
  (and (in-set ?limit-meth limit-methods)                     Retrieve an option-
       (limiting-plan ?limit-meth ?opp-player                 limiting plan method
           (world-at-time ?time)                              that generates a plan
           ?plan (goal-limit ?piece ?loc ?info))              for the opponent
       (not (active-goal ?opp-player                          that's not already
             (goal-limit ?piece ?loc ?info) ?time)) ))        a known active goal
```

TO COMPUTE: A plan the opponent can execute

DETERMINE: An option-limiting method that generates an opponent plan
that he can currently use

Appears on page 149.

Goal generation

The meta-rule

```
(def-rule comp-goals-comp comp-goals                                  If I can generate
  (is-goal ?method computer (world-at-time current-time) ?goal)       a goal to pursue,
=>                                                                    record it as an
  ((active-goal computer ?goal current-time)))                        active goal
```

IF The goal recognition component can find a goal that is relevant in the current situation

THEN Assert that the goal is a currently-active goal

Appears on page 45.

Goals to capture a piece

```
(def-brule is-goal-1
  (is-goal ?player (world-at-time ?time)                              To generate a
      (goal-capture ?opp-piece ?loc2 (move ?move)))                   goal to pursue,
<=
  (and (computed-possible-moves ?player ?time ?move-set)              retrieve the set
       (in-set (move ?player (capture ?opp-piece)                     of active threats
                     ?comp-piece ?loc ?loc2)                          against an opponent
               ?move-set)                                             piece, and consider
       (= ?move (move ?player (capture ?opp-piece)                    the goal of
                      ?comp-piece ?loc ?loc2))  ))                    capturing it
```

TO COMPUTE: A goal for a player

DETERMINE: The set of threats available to the player and a direct capture that is in the set

Appears on page 46.

Goal to capture exposed piece

```
(def-brule is-goal-2
  (is-goal ?player (world-at-time ?time)                              To generate a
           (goal-capture ?piece (loc ?row ?col) exposed))             goal to pursue,
<=
  (and (at-loc (player-opponent ?player) ?piece                       find an opponent
               (loc ?row ?col) ?time)                                 piece that
       (piece-looks-exposed ?piece ?row ?col ?time) ))                looks exposed
```

TO COMPUTE: A goal for a player

DETERMINE: A piece of the opponent's which looks exposed

Goals to counterplan

```
(def-brule is-goal-cp                                          To generate a
  (is-goal computer (world-at-time ?time)                      counterplanning
          (goal-counterplan ?plan))                            goal to pursue,
 <=                                                            retrieve an
  (possible-opp-plan (world-at-time ?time) ?plan ?goal))       opponent plan
```

TO COMPUTE: A goal for a player

DETERMINE: A possible opponent plan to counterplan again

Appears on page 46.

Plan generation meta-rules

Meta-rule for grabbing opportunities

```
(def-rule comp-plan-move comp-plans
  (and (sorted-eval-gen (vars ?goal ?goal-eval ?a-goal)          If the active
         (and (active-goal computer ?a-goal current-time)        goal with the
              (eval-goal ?a-goal ?goal-eval) ))                  highest value
       (not (did-goal-planning current-time))                    before planning
       (opportunity computer (world-at-time current-time)        has an opportunity
                   ?goal ?plan))
=>                                                               Assert the
  ((possible-plan computer current-time ?plan ?goal)             opportunity
   (did-goal-planning current-time) ))                           as a possible plan
```

IF There is an active goal
and there is a move which will achieve it immediately

THEN Assert a possible plan to seize the opportunity

Appears on page 47.

Meta-rule for plan continuation

```
(def-rule comp-plan-continue comp-plans                          If the previously
  (and (active-plan computer (- current-time 2) ?plan ?goal)     active plan has
       (seq-length ?plan ?length)    (>= ?length 2)              length more than 2
       (seq-rest ?plan ?new-plan)    (seq-first ?new-plan ?move) and the next move
       (move-doable ?move (world-at-time current-time)) )        can be made
=>                                                               Then assert the rest
  ((possible-plan computer current-time ?new-plan ?goal) ))      as possible
```

IF There was a plan from the previous turn
that has some steps still to be executed
and the next step in the plan is applicable now

THEN Assert as a possible plan the rest of the previous plan

Meta-rule for counterplanning

```
(def-rule comp-counterplan comp-plans                                  If there's an active
 (and (active-goal opponent ?opp-goal current-time)                    opponent goal, and
      (counterplan computer ?opp-goal current-time ?counterplan))      I have a counterplan,
=>                                                                     Then assert the plan
 ((possible-plan computer current-time ?counterplan                    as possible to execute
                 (goal-counterplan ?opp-goal))))
```

IF The opponent has an active goal
and I can generate a counterplan against it

THEN Assert the counterplan as a possible plan

Appears on page 49.

Meta-rule for forced-outcome strategies (goal-directed)

```
(def-rule comp-strategy comp-plans                                     If no plan has been
  (and (not (possible-plan computer current-time ?p ?g))               suggested, and the
       (sorted-eval-gen (vars ?goal ?goal-eval ?a-goal)                highest-valued
           (and (active-goal computer ?a-goal current-time)            active goal
                (eval-goal ?a-goal ?goal-eval)  ))                     such that a
       (in-set ?strategy-meth strategy-meths)                          strategy method
       (strategy strategy-meth computer                                returns a plan
                 (world-at-time current-time) ?goal ?plan)             and the plan's
       (seq-1st ?plan ?move)                                           first move is
       (move-doable ?move (world-at-time current-time)) )              currently feasible
=>                                                                     Assert the plan as
  ((possible-plan computer current-time ?plan ?goal) ))                a possible plan
```

IF There is no plan suggested yet
and the highest-valued active goal
has an offensive strategy that can satisfy it
which can be executed in the current situation

THEN Assert the plan as a possible plan

Appears on page 52.

Forward search (goal-directed) meta-rule

```
(def-rule comp-forward-plan comp-plans                                 If there is not
  (and (not (possible-plan ?computer current-time ?p ?g))              a plan suggested
       (sorted-eval-gen (vars ?goal ?goal-eval ?a-goal)                then retrieve the
           (and (active-goal computer ?a-goal current-time)            highest-valued
                (eval-goal ?a-goal ?goal-eval)))                       active goals
       (plan-forward computer current-time ?plan ?goal 0) )            and plan for them
=>                                                                     and assert results
  ((possible-plan computer current-time ?plan ?goal) ))                as possible plans
```

IF There is not yet a possible plan
and the best active goal has a plan found by brute-force search

THEN Assert the plan as a possible plan

Appears on page 55.

Option-limiting plans meta-rule

```
(def-rule comp-limit comp-plans                                  ; If there is not a plan
  (and (not (possible-plan computer current-time ?p ?g))         ; already suggested,
       (in-set ?limit-meth limit-methods)                        ; and there is an option-
       (limiting-plan ?limit-meth computer                       ; limiting plans method
          (world-at-time current-time) ?plan ?goal)              ; that currently applies
       (= ?plan (plan ?move ?plan-rest))                         ; whose first move can
       (move-doable ?move (world-at-time current-time)) )        ; now be executed
=>
  ((possible-plan computer (current-time) plan goal) ))          ; Assert the possible plan
```

IF There is not yet a possible plan
and an option-limiting planning method generates a currently executable plan

THEN Assert the plan as a possible plan

Meta-rule for forced-outcome strategies (non-goal-directed)

```
(def-rule comp-strategy-2 comp-plans                             ; If no plan has been
  (and (not (possible-plan computer current-time ?p ?g))         ; suggested, and
       (in-set ?strategy-meth strategy-meths)                    ; a forced-outcome script
       (strategy ?strategy-meth computer                         ; currently applies
          (world-at-time current-time) ?goal ?plan)              ; and generates a plan
       (= ?plan (plan ?move ?plan-rest))                         ; whose first move
       (move-doable ?move (world-at-time current-time)) )        ; can be executed
=>
  ((possible-plan computer (current-time) plan goal) ))          ; Assert the possible plan
```

IF There is no possible plan asserted yet
and an offensive strategy yields a plan

THEN Assert the possible plan

Forward search (non-goal-directed) meta-rule

```
(def-rule comp-forward-plan-2 comp-plans                              If no plan has been
  (and (not (possible-plan computer (current-time) p g))              suggested, and
       (= min-val 0)                                                  a two-move plan
       (plan-forward computer (current-time) plan goal min-val))      can be generated
=>                                                                    Assert it as
  ((possible-plan computer (current-time) plan goal) ))               a possible plan
```

IF There is not yet a possible plan
and the planner can generate a two-move-plan

THEN Activate the plan

Counterplanning rules

Running away

```
(def-brule counterplan-run
  (counterplan cp-run ?player                                         To counterplan
    (goal-capture ?piece ?loc                                         against an opponent
     (move ?opponent (capture ?piece) ?opp-piece ?opp-loc ?loc))      attack against a
     ?time ?the-response)                                             player's piece
<=                                                                    Find a place the
 (and (move-legal (move ?player ?move ?piece ?loc ?new-loc))          attacked piece can
      (no (and (at-loc ?opponent ?other-opp-piece                     move to that no
                       ?other-opp-loc ?time)                          opponent piece can
               (move-legal                                            attack
                 (move ?opponent (capture ?piece)
                       ?other-opp-piece ?other-opp-loc ?new-loc))))
      (= ?the-response                                                and return the plan
         (plan (move ?player move ?piece ?loc ?new-loc) done))))      to make the move
```

TO COMPUTE: A counterplan to a capture

DETERMINE: A move from the targeted square
to a square which cannot be attacked

Appears on page 49.

Counterattacking

```
(def-brule counterplan-2
  (counterplan cp-counterattack ?player                              To counterplan
               (goal-capture ?target ?target-loc                     against an
                       (move (move ?opp-player (capture ?target)     opponent attack
                                 ?opp-piece ?opp-loc ?target-loc)))  against a
               ?time ?the-response)                                  computer piece
<=
 (and (at-loc ?player ?piece ?loc ?time)                             Find a piece that
      (move-legal (move ?player (capture ?opp-piece)                 threatens the
                  ?piece ?loc ?opp-loc))                             attacking piece
      (= ?the-response                                               and capture the
         (plan (move ?player (capture ?opp-piece) ?piece             attacker with it.
                     ?loc ?opp-loc)
               done)) ))
```

TO COMPUTE: A counterplan to a capture

DETERMINE: A piece on the board that can capture the attacker

Appears on page 33.

Counterplanning by interposition (learned)

```
(def-brule cp-learned-1
  (counterplan learned-cp-meth-1 ?player                  To counterplan against
     (goal-capture ?piece3 (rc->loc ?r2 ?c2)              an opponent threat
       (move ?other-player (capture ?piece3) ?piece       against a piece
             (loc ?r1 ?c1) (loc ?r2 ?c2)))
    ?time ?counterplan)
<=
 (and (= ?counterplan                                     plan to make a move
         (plan (move ?player move ?other-piece
                     ?other-loc (loc ?r3 ?c3)) done))
      (not (= ?piece knight))                             If the piece isn't a knight
      (loc-on-line ?r3 ?c3 ?r1 ?c1 ?r2 ?c2)               to a square on the line
      (move-legal-in-world                                of attack
          (move ?player move ?other-piece                 if there's a piece that can
                ?other-loc (loc ?r3 ?c3))                 move to that square
          (world-at-time ?time))
      (at-loc ?player ?other-piece ?other-loc)            and the square is empty
      (not (at-loc ?any-player ?any-piece (loc ?r3 ?c3))) ))
```

TO COMPUTE: A counterplan to an attack

DETERMINE: A location on the line of attack
that another piece can move to

Appears on pages 27 and 102.

Counterplanning by buying time

```
(def-brule counterplan-buy-time
 (counterplan cp-buy-time ?player                                    To counterplan
    (goal-capture ?target-piece ?target-loc                          against an opponent
         (move (move ?opp-player (capture ?target-piece)             attack against a
                     ?opp-piece ?opp-loc ?target-loc)))              player's piece
    ?time ?cp)
<=                                                                   (Unless the attacked
 (and (not (= ?target-piece king))                                   piece is a king)
      (= ?cp (plan (move ?player move ?cp-piece ?cp-loc ?interm-loc)
                   (next (move ?player (capture ?opp-piece)          First move to an
                               ?cp-piece ?interm-loc ?opp-loc)       intermediate square
                         done)))
      (at-loc ?opp-player king ?king-loc ?time)                      which threatens the
      (move-legal-in-world (move ?player (capture king) ?cp-piece
                                 ?interm-loc ?king-loc)              opponent's king
                           (world-at-time ?time))
      (move-legal-in-world                                           which from there
        (move ?player (capture ?opp-piece) ?cp-piece                 can capture the
              ?interm-loc ?opp-loc)                                  opponent piece
        (world-at-time ?time))                                       that is making the
      (move-doable (move ?player ?move-type                          original attack
                         ?cp-piece ?cp-loc ?interm-loc)
                   (world-at-time ?time))
      (no (and (at-loc ?opp-player ?other-piece ?other-loc ?time)
               (move-legal-in-world                                  As long as no
                  (move ?opp-player (capture ?cp-piece)              opponent piece
                        ?other-piece ?other-loc ?interm-loc)         threatens the
                  (world-at-time ?time)))) ))                        intermediate square
```

TO COMPUTE: A counterplan to a capture

DETERMINE: A location from which a piece can attack the king
and a piece on the board that can move to that location
such that the moved piece can counter the threat
from its new location

Appears on page 50.

Forced-outcome strategies

The fork (learned)

```
(BRULE learned-strategy-method8574
(strategy learned-strategy-method8574 ?player (world-at-time ?time)          A forced-outcome
          (goal-capture ?target-piece (loc ?r2 ?c2) ?info)                  strategy for
    (plan (move ?player move ?en-piece (loc ?r ?c) (loc ?r3 ?c3))           capturing a piece
          (next (move ?player (capture ?target-piece) ?en-piece             using two moves
                      (loc ?r3 ?c3) (loc ?r2 ?c2)) done)))
<=
 (and (= ?opp (player-opponent ?player)) (= (current-game) chess)
     (at-loc ?player ?en-piece (loc ?r ?c) ?time)                           Find a piece
     (move-legal-in-world                                                   that can move
        (move ?player move ?en-piece (loc ?r ?c) (loc ?r3 ?c3))
        (world-at-time ?time))
     (not (at-loc ?other-player ?other-piece (loc ?r3 ?c3) ?time))          to an empty square
     (at-loc ?opp ?target-piece (loc ?r2 ?c2) ?time)                        and a piece of the
     (move-legal-in-world (move ?player (capture ?target-piece) ?en-piece   opponent's that
                                (loc ?r3 ?c3) (loc ?r2 ?c2))                can be attacked
        (world-at-time ?time))                                              by the moved piece
     (at-loc ?opp ?other-target (loc ?r4 ?c4) ?time)                        and another piece
     (move-legal-in-world (move ?player (capture ?other-target) ?en-piece   of the opponent's
                                (loc ?r3 ?c3) (loc ?r4 ?c4))                that can be taken
        (world-at-time ?time)                                               by the same piece
     (not (= (move ?player (capture ?target-piece) ?en-piece
                   (loc ?r3 ?c3) (loc ?r2 ?c2))
             (move ?player (capture ?other-target) ?en-piece
                   (loc ?r3 ?c3) (loc ?r4 ?c4))))
     (no (and (counterplan ?cp-meth ?opp
                (goal-capture ?target-piece (loc ?r2 ?c2)                   such that there is
                   (move (move ?player (capture ?target-piece) ?en-piece    no counterplan
                               (loc ?r3 ?c3) (loc ?r2 ?c2))))               for the opponent
                ?time ?cp)                                                  that handles both
              (counterplan ?cp-meth2 ?opp                                   of the attacks
                (goal-capture ?other-target (loc ?r4 ?c4)
                   (move (move ?player (capture ?other-target) ?en-piece
                               (loc ?r3 ?c3) (loc ?r4 ?c4))))
                ?time ?cp))))
```

TO COMPUTE: A strategy for capturing a piece

DETERMINE: An piece that can be moved to an empty intermediate location
and an opponent piece that can be attacked
by the moved piece at the intermediate location
and another opponent piece that can be attacked
by the moved piece at the intermediate location
such that there is no single counterplan to both attacks

Rule appears on pages 13, 53 and 142.

Simultaneous attacks strategy (learned)

```
(BRULE learned-strategy-method10486
(strategy learned-strategy-method10486 ?other-player2                    A forced-outcome
    (world-at-time ?time2)                                               strategy for
    (goal-capture ?target-piece (loc ?r3 ?c3) ?info)                     capturing a piece
    (plan (move ?other-player2 move ?other-piece2 (loc ?r ?c) (loc ?r2 ?c2))   using two moves
          (next (move ?other-player2 (capture ?target-piece)
                      other-piece2 (loc ?r2 ?c2) (loc ?r3 ?c3)) done)))
<=
(and (= ?opp (player-opponent ?other-player2)) (= (current-game) chess)
     (at-loc ?other-player2 ?other-piece2 (loc ?r ?c) ?time2)             Find a piece to
     (move-legal-in-world                                                move to a new
       (move ?other-player2 move ?other-piece2 (loc ?r ?c) (loc ?r2 ?c2)) location
       (world-at-time ?time2))
     (not (at-loc ?other-player ?other-piece (loc ?r2 ?c2) ?time2))      and an opponent
     (at-loc ?opp ?target-piece (loc ?r3 ?c3) ?time2)                    piece that can be
     (move-legal-in-world                                                attacked by the
       (move ?other-player2 (capture ?target-piece)                      attacking piece
             ?other-piece2 (loc ?r2 ?c2) (loc ?r3 ?c3))                  from its new
       (world-at-time ?time2))                                           location, and
     (not (= ?other-piece3 knight))                                      another non-knight
     (newest-loc-on-line (loc ?r ?c) (loc ?r4 ?c4) (loc ?r5 ?c5))       piece that can
     (at-loc ?other-player2 ?other-piece3 (loc ?r4 ?c4) ?time2)          attack another
     (at-loc (player-opponent ?other-player2) ?other-target              opponent piece
             (loc ?r5 ?c5) ?time2)                                       along the line of
     (move-legal (move ?other-player2 (capture ?other-target)            attack emptied by
                       ?other-piece3 (loc ?r4 ?c4) (loc ?r5 ?c5)))       moving the first
     (not (= (move ?other-player2 (capture ?target-piece)                piece
                   ?other-piece2 (loc ?r2 ?c2) (loc ?r3 ?c3))
             (move ?other-player2 (capture ?other-target)
                   ?other-piece3 (loc ?r4 ?c4) (loc ?r5 ?c5))))
     (no (and (counterplan ?cp-meth ?opp                                 such that no
                (goal-capture ?target-piece (loc ?r3 ?c3)                single opponent
                  (move (move ?other-player2 (capture ?target-piece)     move can
                              ?other-piece2 (loc ?r2 ?c2) (loc ?r3 ?c3))))  counterplan
                ?time2 ?cp)                                              both attacks
              (counterplan ?cp-meth2 ?opp
                (goal-capture ?other-target (loc ?r5 ?c5)
                  (move (move ?other-player2 (capture ?other-target)
                              ?other-piece3 (loc ?r4 ?c4) (loc ?r5 ?c5))))
                ?time2 ?cp)))) )
```

TO COMPUTE: A strategy for capturing a piece

DETERMINE: A piece that can move off of a line of attack
and a piece that can attack along the line of attack
such that the moved piece can attack a second piece
from its new location

Rule appears on page 147.

Option-limiting plans

The pin (learned)

```
(BRULE learned-limiting-plan-method3105
(limiting-plan learned-limiting-plan-method3105 ?opp                    To limit the
   (world-at-time ?time)                                                options
   (plan (move ?opp move ?other-piece2
               (loc ?r2 ?c2) (loc ?att-r ?att-c))
         done)
   (goal-limit ?def-piece (loc ?def-r ?def-c)                           of a defending
      (move (move ?opp move ?other-piece2                               piece
               (loc ?r2 ?c2) (loc ?att-r ?att-c)))))
<=
 (and (= ?def (player-opponent ?opp)) (= (current-game) ?chess)
      (at-loc ?opp ?other-piece2 (loc ?r2 ?c2) ?time)                   Find a piece
      (move-legal-in-world (move ?opp move ?other-piece2                that can move to a
                            (loc ?r2 ?c2) (loc ?att-r ?att-c))          new location
                           (world-at-time ?time))
      (not (at-loc ?other-player ?other-piece                           that's empty
                   (loc ?att-r ?att-c) ?time))
      (at-loc ?def ?def-piece (loc ?def-r ?def-c) ?time)                and a defending
      (move-legal-in-world (move ?def move ?def-piece                   piece that can move
                              (loc ?def-r ?def-c) (rc->loc ?r ?c))
                           (world-at-time ?time))
      (at-loc ?def king (loc ?to-row ?to-col) ?time)                    such that the piece
      (move-legal (move ?opp (capture king) ?other-piece2               can legally attack
                   (loc ?att-r ?att-c) (loc ?to-row ?to-col)))          the opponent's king
      (newest-loc-on-line (loc ?def-r ?def-c)                           but is blocked by
        (loc ?att-r ?att-c) (loc ?to-row ?to-col))                      the defending piece
      (no (and (newest-loc-on-line (loc ?other-r ?other-c)
                (loc ?att-r ?att-c) (loc ?to-row ?to-col))              and such that no
          (not (= ?other-r ?def-r)) (not (= ?other-c ?def-c))           other square on the
          (at-loc ?other-player2 ?other-piece3                          line of attack
                  (loc ?other-r ?other-c) ?time)))))                    is occupied
```

TO COMPUTE: A plan to limit the opponent's options

DETERMINE: Find a piece to move and a location to move to
and an opponent piece to pin
such that the moved piece can legally attack the king
and the attack is blocked only by the defending piece

Appears on page 154.

Boxing in

```
(BRULE learned-limiting-plan-method13791
(limiting-plan learned-limiting-plan-method13791                    To limit the
 ?opp2 (world-at-time ?time)                                       options
 (plan (move ?opp2 move ?other-piece2 (loc ?r ?c) (loc ?r4 ?c4))
       done)
 (goal-limit king (loc ?r2 ?c2)                                    of the defending
   (move (move ?opp2 move ?other-piece2                            king
               (loc ?r ?c) (loc ?r4 ?c4)))))
<=
(and (= ?opp (player-opponent ?opp2)) (= (current-game) chess)
     (at-loc ?opp2 ?other-piece2 (loc ?r ?c) ?time)                Find a piece to
     (move-legal-in-world (move ?opp2 move ?other-piece2           move to a new
                           (loc ?r ?c) (loc ?r4 ?c4))              location
          (world-at-time time))
     (not (at-loc ?other-player ?other-piece                       that's vacant
                  (loc ?r4 ?c4) ?time))
     (at-loc ?opp king (loc ?r2 ?c2) ?time)                        such that the
     (move-legal-in-world (move ?opp move king                     defending king can
                           (loc ?r2 ?c2) (loc ?r3 ?c3))            move to a location
          (world-at-time time))
     (move-legal-in-world (move ?opp2 (capture king) ?other-piece2
                           (loc ?r4 ?c4) (loc ?r3 ?c3))            that the moved
          (world-at-time time))) )                                 piece can move to
```

TO COMPUTE: A plan to limit the opponent's king's options

DETERMINE: Find a piece to move and a location to move to **such that** the opponent's king can move to a location that the moved piece can move to

Appears on page 156.

Plan selection

Choose an activated plan

```
(def-rule comp-move comp-move                                      If the computer
  (and (not (in-check computer current-time ?move))                is not in check
       (sorted-eval-gen (vars ?plan ?goal-val ?a-plan)             choose the highest
         (and (possible-plan computer current-time                 valued active goal
                             ?a-plan ?a-goal)
              (eval-goal ?a-goal ?goal-val) ))                     for which there is
       (possible-plan computer current-time ?plan ?goal)           a possible plan
       (= ?plan (plan ?move ?plan-rest))                           such that the first
       (move-doable ?move (world-at-time current-time)) )          move is feasible,
=>                                                                 Then activate the
  ((active-plan computer current-time ?plan ?goal)                 plan and make the
   (move-to-make ?move computer ?goal current-time) ))             first move
```

IF The computer is not in check
and there is a possible plan that satisfies a goal
of higher value than the goals of all other possible plans
and whose first move is currently feasible

THEN Activate the plan for execution
and activate the first move in the plan as the current move

Appears on page 56.

Responding when in check

```
(def-rule comp-check-response comp-move                          If the computer
  (and (in-check computer (current-time) opp-move)               is in check, and has
       (possible-plan computer (current-time) plan               a play to counter,
         (goal-counterplan (goal-capture king loc (move opp-move))))
       (= plan (plan move plan-rest))                            whose first move
       (move-doable move (world-at-time (current-time))) )       can be executed
=>
  ((active-plan computer (current-time) plan                     Activate the plan
       (goal-counterplan (goal-capture king loc opp-move)))      and make the first
   (move-to-make move computer                                   move
      (goal-counterplan (goal-capture king loc (move opp-move)))
      (current-time)) ))
```

IF The computer is in check
and there is a currently executable counterplan

THEN Activate the plan for execution
and activate the first move in the plan as the current move

Make a random move

```
(def-rule comp-move-random comp-move                             If there's no possible
  (and (no (possible-plan computer (current-time) plan goal))    plan, choose a
       (choose-move-random computer move (current-time)) )       random move to make
=>
  ((move-to-make move computer (goal-none) (current-time))       and activate it
   (display-win planner "Making a random move: " move) ))
```

IF There is no active plan
and I can generate a random move

THEN Activate the random move for execution

In check with no response

```
(def-rule comp-check-no-response comp-move
  (and (in-check computer (current-time) move)                    If the computer is in
       (no (possible-plan computer (current-time) plan            check, and there's no
            (goal-counterplan (goal-capture king loc info)))) )   counterplan,
=>
  ((checkmate opponent (current-time))                            The opponent won
   (display "***CHECKMATE***")
   (display-win top "***CHECKMATE***") ))
```

IF The computer is in check
and there is no counterplan

THEN The opponent has won

Other rules used in planning

```
(def-brule plan-opportunity                                       To find an opportunity
  (opportunity ?player (world-at-time ?time)                      to capture a piece
     (goal-capture ?opp-piece ?loc (move ?move))
     ?the-plan)
<=
  (and (computed-possible-moves ?player ?time ?ms)                Retrieve the set of
       (in-set ?move ?ms)                                         available moves
       (= ?move                                                   and find a move in it
          (move ?player (capture ?opp-piece)                      that captures an
                ?piece ?player-loc ?loc))                         opponent piece
       (= ?the-plan (plan ?move done))) )
```

TO COMPUTE: An opportunity for a capture

DETERMINE: The already-computed set of possible moves
and a move in that set that captures a piece

Appendix B

Vocabulary for explanation construction

This appendix gives the inference rules in CASTLE's explanatory vocabulary. As in the previous appendix, each rule is given in its internal format, with annotations, and in plain English. The explanatory model is broken up into the following portions:

Component specifications

Counterplanning component specification

```
(def-brule meth-spec-cp
  (method-spec counterplan
    (counterplan ?player (goal-capture ?piece ?loc ?threat)
                 ?time ?counterplan))
<=
  (and (= ?counterplan (plan ?cp-move done))
       (expl-disabled ?cp-move ?threat)
       (move-doable ?cp-move (world-at-time ?time)) ))
```

The specification of the CP component invoked on a goal to capture a piece: a CP is a move that disabled the threat, and was feasible

TO COMPUTE: The reason that a move was a counterplan to an attack

DETERMINE: The move that made up the counterplan
that disabled the threat
and was a valid move at the time of the counterplanning

Appears on page 90.

Threat detection focusing component specification

```
(def-brule meth-precond-focus
  (method-precond focus
    (focus ?f-meth ?player ?move (world-at-time ?time)))
<=
 (and (= ?enable-time (1- ?time))
      (move-to-make ?prev-move ?other-player ?goal ?enable-time)
      (expl-enabled ?prev-move ?move ?enable-time)))
```

The specification of the detection focusing component: Focus on moves enabled by the previously-made move

TO COMPUTE: The reason that a possible move should be focused on

DETERMINE: The move made at the previous time
that enabled the possible new move

Forced-outcome strategy component specification

```
(def-brule method-precond-strategy
  (method-precond strategy
    (strategy ?strategy-meth ?player (world-at-time ?time)
              (goal-capture ?target-piece ?target-loc ?info)
              ?plan))
<=
  (and (forced-goal-achieve ?plan
         (goal-capture ?target-piece ?target-loc ?info)
         ?player (world-at-time ?time))))
```

A plan is a forced-outcome strategy for a goal

If it is guaranteed to satisfy the goal

Appears on page 137.

```
(def-brule forced-goal-1                                    A single-move plan
  (forced-goal-achieve (plan ?move done)                    forces the
      (goal-capture ?target ?loc2 (move ?move))             achievement of a
      ?player (world-at-time ?time) )                       piece capture
<=
  (and (= ?move (move ?player (capture ?target) ?piece      if the move is to
                      ?loc1 ?loc2))                         directly capture it
       (move-doable ?move (world-at-time ?time)) ))         and is doable
```

Appears on page 138.

```
(def-brule forced-goal-2                                    A multi-move plan
  (forced-goal-achieve (plan ?enabler (next ?move2 ?rest))  forces the
     ?goal ?player (world-at-time ?time))                   achievement of a goal
<=
 (and (= ?cp-time (1+ ?time)) (= ?opp (player-opponent ?player))
      (= ?enabler                                           if the first move
         (move ?player ?move-type ?en-piece ?loc1 ?loc2))   is currently doable
      (move-doable ?enabler (world-at-time ?time))          and the rest of the
      (forced-goal-achieve (plan ?move2 ?rest) ?goal ?player  plan achieves the
                           (world-at-time ?cp-time))        goal, and the first
      (expl-enabled ?enabler ?move2 ?time)                  enables the rest
      (= ?other-move
         (move ?player (capture ?other-target) ?other-piece  and another move
               ?other-loc1 ?other-loc2))                    is enabled by the
      (forced-goal-achieve (plan ?other-move ?other-rest)   first move that
         ?other-goal ?player (world-at-time ?cp-time))      satisfies another
      (expl-enabled ?enabler ?other-move ?time)             goal,
      (not (= ?other-move ?move2))
      (no (and (counterplan ?cp-meth1 ?opp ?goal ?cp-time ?cp)  and there is no
               (counterplan ?cp-meth2 ?opp ?other-goal      counterplan to both
                            ?cp-time ?cp))) ))              plan continuations
```

TO COMPUTE: How a multi-move plan necessarily satisfies a goal

DETERMINE: A feasible first move
and a subsequent plan that satisfies the goal
such that the first move enables the subsequent plan
and another enabled subsequent plan
that also achieves a goal
such that there is no one counterplan against both

Appears on page 138.

Option-limiting plan component specification

```
(def-brule meth-precond-limit
  (method-precond limiting-plan
    (limiting-plan ?limit-meth ?player (world-at-time ?time)      A one-move plan
       (plan ?the-move done)                                      is an option-
       (goal-limit ?dis-piece ?dis-loc1 (move ?the-move))))       limiting plan if
<=
  (and (= ?opp (player-opponent ?player)) (= ?limited-time (1+ ?time))
       (= ?the-move (move ?player ?the-move-type ?the-piece       the plan was
                          ?the-loc1 ?the-loc2))                   feasible
       (move-doable ?the-move (world-at-time ?time))
       (= ?dis-move1                                              and an opponent
          (move ?opp ?dis-move-type ?dis-piece                    move was feasible
                ?dis-loc1 ?dis-loc2))                             before the plan was
       (at-loc ?opp ?dis-piece ?dis-loc1 ?time)                   carried out
       (move-legal-in-world ?dis-move1 (world-at-time ?time))
       (not (move-doable ?dis-move1                               but not after the
             (world-at-time ?limited-time)))                      plan was executed
       (expl-disabled-reasons ?the-move ?dis-move1                and the plan
                              ?time ?pre1 ?eff)                   actually disabled
       (num-such-that (vars ?num ?dis-move 3)                     the move
           (and (= ?dis-move
                   (move ?opp ?dis-move-type-a ?dis-piece         and there are more
                         ?dis-loc1 ?dis-loc2a))                   than two moves
                (move-doable ?dis-move (world-at-time ?time))     disabled by the
                (not (move-doable ?dis-move                       plan due to the
                       (world-at-time ?limited-time)))            same effect of the
                (expl-disabled-reasons ?the-move ?dis-move        opponent move
                                       ?time ?pre ?eff) ))
       (> ?num 2) ))
```

TO COMPUTE: How a move limited opponent options

DETERMINE: A move that was feasible
and an opponent move that it prevents
and more than two opponent moves similarly prevented

Appears on page 152.

Enablement and disablement

Move enablement

```
(def-brule expl7                                    To explain how one move enabled
  (expl-enabled move1 move2 time)                   another move,
<=
  (and (= time2 (1+ time))
       (expl-move-precond move2 time2 pre)          find a precondition of the second
       (expl-move-effect move1 time eff)            and an effect of the first
       (expl-matches pre eff) ))                    that match
```

TO COMPUTE: Whether one move enabled another

DETERMINE: A precondition of the enabled move
and an effect of the enabling move
such that the precondition and effect match

Move disablement

```
(def-brule expl1                                    To explain how one move
  (expl-disabled ?move1 ?move2)                     disabled another move
<=
  (and (expl-move-precond ?move2 ?pre)              find a precondition of the second
       (expl-move-effect ?move1 ?eff)               and an effect of the first
       (expl-conflicts ?pre ?eff) ))                such that they conflict
```

TO COMPUTE: Whether one move disabled another

DETERMINE: A precondition of the disabled move
and an effect of the disabling move
such that the precondition and the effect conflict

Appears on page 91.

Preconditions of moves

Precondition: Free line of attack

```
(def-brule expl2
  (expl-move-precond (move ?player ?move-type ?piece      A move has a precondition
                           (loc ?r1 ?c1) (loc ?r2 ?c2))
    (not (and (loc-on-line ?r3 ?c3 ?r1 ?c1 ?r2 ?c2)       that there is no square
              (at-loc ?other-piece ?other-player           on the line of attack
                      (loc ?r3 ?c3) ?time))))              that is occupied
<=
  (and (not (= ?piece knight))) )                         if the piece is not a knight
```

TO COMPUTE: A precondition of a move to have no occupied square on the line of attack

DETERMINE: That the attacking piece isn't a knight

Appears on page 92.

Precondition: Piece at starting location

```
(def-brule expl2b
  (expl-move-precond
      (move ?player ?move-type ?piece ?loc1 ?loc2)        A precondition of a move
      ?time (at-loc ?player ?piece ?loc1 ?time))          is that the piece be at
 <= ())                                                   the starting location
```

TO COMPUTE: A precondition of a move that the piece is at the starting location

DETERMINE: (always true)

Appears on page 126.

Precondition: Target at destination location

```
(def-brule expl2c
  (expl-move-precond (move ?player (capture ?target)
                           ?piece ?loc1 ?loc2)
     ?time (at-loc ?opponent ?target ?loc2 ?time))
 <= (= ?opponent (player-opponent ?player)))
```

A precondition of a capture is that the target is at the location being attacked

TO COMPUTE: A precondition of a move that the target is in location

DETERMINE: (always true)

Appears on page 122.

Precondition: King not under threat

```
(def-brule expl2d
  (expl-move-precond                                          A precondition of a move
        (move player move-type piece loc1 loc2) time          that there not be an
    (not-and (at-loc opp other-piece other-loc time)          opponent piece that can
        (possible-king-threat opp other-piece other-loc       threaten the king
                              player piece loc1 time)))
<=
  (= opp (player-opponent player)) )
```

TO COMPUTE: A precondition of a move that the king not be in danger

DETERMINE: (always true)

Precondition: King not moved into threat

```
(def-brule expl2e
  (expl-move-precond
        (move player move-type piece loc1 loc2) time          A precondition of a move
    (not-and (at-loc opp other-piece other-loc time)          that there not exist an
             (move-legal-in-world                             opponent piece that can
              (move opp (move-take piece) other-piece         attack the destination
                    other-loc loc2)                           location
              (world-at-time time))))
<=
  (and (= piece king) (= opp (player-opponent player))))      if it's a king move
```

TO COMPUTE: A precondition of a king move
that the destination not be under attack

DETERMINE: (always true)

Effects of moves

Effect: Piece at new location

```
(def-brule expl3
  (expl-move-effect                                    An effect of a move
     (move ?player ?move-type ?piece ?loc1 ?loc2)      is that the piece is now
     (at-loc ?piece ?player ?loc2))                    at the new location
<=
  ())
```

TO COMPUTE: An effect of moving a piece that it's at its new location

DETERMINE: (always true)

Appears on page 92.

Effect: Piece not at initial location

```
(def-brule expl3b
  (expl-move-effect (move ?player ?move-type           An effect of a move
                          ?piece ?loc1 ?loc2)          is that the piece is no
    ?time                                              longer at the initial
    (not-true (at-loc ?player ?piece ?loc1 (1+ ?time)))) location
 <= ())
```

TO COMPUTE: An effect of moving a piece that the initial square is empty

DETERMINE: (always true)

Appears on page 122.

Effect: Target no longer at old location

```
(def-brule expl3c
  (expl-move-effect (move ?player (capture ?target)    An effect of a capture is
                          ?piece ?loc1 ?loc2) ?time    that the target is no
    (not-true (at-loc ?opponent ?target                longer at its old location
                      ?loc2 (1+ ?time))))
 <= (= ?opponent (player-opponent ?player)))
```

TO COMPUTE: An effect of a capture that the target has been removed

DETERMINE: (always true)

Appears on page 126.

Condition match and conflict

Condition match

```
(def-brule expl9                                    A condition that two things not
  (expl-matches (not-and conj1 conj2)               both be true matches a condition
                (not-true conj2))                   that the second not be true
<=
  (true conj1))                                     if the first is true
```

```
(def-brule expl9b
  (expl-matches cond1 cond2)                        Two conditions match
<=
  (= cond1 cond2))                                  If they are the same
```

Condition conflicts

```
(def-brule expl4                                    A condition that two things
  (expl-conflicts (not (and ?conj1 ?conj2)) ?conj2) aren't both true conflicts with
<=                                                  the second being true
  (true ?conj1))                                    if the first is already true
```

Appears on page 93.

```
(def-brule expl4b (conj1 conj2)                     A condition that two things aren't
  (expl-conflicts (not-and conj1 conj2)             both true conflicts with the first
                  conj1)                            one being true
<=
  (true conj2))                                     if the second is true
```

```
(def-brule expl4c (bel)                             A condition that something is true
  (expl-conflicts bel (not-true bel))               always conflicts with a condition
<= ())                                              that it is untrue
```

Auxiliary explanatory rules

```
(def-brule move-doable-1
  (move-doable (move ?player move ?piece ?loc1 ?loc2)   A non-capturing move is
    (world-at-time ?time))                              "doable"
<=
 (and (at-loc ?player ?piece ?loc1 ?time)               if the piece is at the
      (move-legal-in-world                              initial location, and
           (move ?player move ?piece ?loc1 ?loc2)       the move is currently
           (world-at-time ?time))                       legal, and the
      (not (at-loc ?other-player ?other-piece           destination square is
                   ?loc2 ?time))))                      empty
```

```
(def-brule move-doable-2
  (move-doable                                                    A capture is "doable"
    (move ?player (capture ?target) ?piece ?loc1 ?loc2)           at a certain time
    (world-at-time ?time))
<=
  (and (at-loc ?player ?piece ?loc1 ?time)                        if the piece is at the
       (at-loc (player-opponent ?player)                          initial location, and
               ?target ?loc2 ?time)                               the target is at the
       (move-legal-in-world                                       destination, and the
           (move ?player (capture ?target)                        move is currently legal
                 ?piece ?loc1 ?loc2)
           (world-at-time ?time)) ))
```

References

[Agre and Chapman, 1987] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 1987.

[Allen *et al.*, 1990] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning.* Morgan Kaufmann, San Mateo, CA, 1990.

[Azarewicz *et al.*, 1986] J. Azarewicz, G. Fala, R. Fink, and C. Heithecker. Plan recognition for airborne tactical decision making. In *Proc. AAAI-86*, pages 805–811, Philadelphia, PA, August 1986. AAAI.

[Becker and Hinton, 1989] S. Becker and G. Hinton. Spacial coherence as an internal teacher for a neural network. Technical Report CRG-TR-89-7, University of Toronto, 1989.

[Birnbaum *et al.*, 1989] L. Birnbaum, G. Collins, and B. Krulwich. Issues in the justification-based diagnosis of planning failures. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 194–196, Ithaca, NY, 1989.

[Birnbaum *et al.*, 1990] L. Birnbaum, G. Collins, M. Freed, and B. Krulwich. Model-based diagnosis of planning failures. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 318–323, Boston, MA, 1990.

[Birnbaum *et al.*, 1991] L. Birnbaum, G. Collins, M. Brand, M. Freed, B. Krulwich, and L. Pryor. A model-based approach to the construction of adaptive case-based planning systems. In *Proceedings of the 1991 Workshop on Case-Based Reasoning*, Washington, D.C., 1991.

[Birnbaum *et al.*, 1993] L. Birnbaum, M. Brand, and P. Cooper. Looking for trouble: Using causal semantics to direct focus of attention. In *Proceedings of the 1993 International Conference on Computer Vision*, Berlin, Germany, 1993.

[Brachman and Levesque, 1985] R.J. Brachman and H.J. Levesque, editors. *Readings in Knowledge Representation.* Morgan Kaufmann, Los Altos, CA, 1985.

[Braverman and Russell, 1988] M. Braverman and S. Russell. Boundaries of operationality. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 221–234, Ann Arbor, MI, 1988.

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

[Carbonell and Veloso, 1988] J. Carbonell and M. Veloso. Integrating derivational analogy into a general problem solving architecture. In *Proceedings of the First Workshop on Case-Based Reasoning*, pages 104–124, 1988.

[Carbonell *et al.*, 1990] J. Carbonell, Y. Gil, R. Joseph, C. Knoblock, S. Minton, and M. Veloso. Designing an integrated architecture: The PRODIGY view. In *Proceedings of the Twelfth Annual Conference of The Cognitive Science Society*, pages 997–1004, 1990.

[Carbonell, 1979] J.G. Carbonell. The counterplanning process: Reasoning under adversity. In *Proc. IJCAI-79*, pages 124–130, 1979. Extended paper: Carnegie-Mellon Computer Science Tech. Report.

[Carbonell, 1981] J.G Carbonell. Counterplanning: A strategy-based model of adversary planning in real-world situations. *AI*, 16(3):295–329, July 1981.

[Carbonell, 1986] J.G. Carbonell. Derivational analogy: A theory of reconstructive problem solving andexpertise acquisition. In *Machine Learning: An Artificial Intelligence Approach*, volume 2, pages 371–392. Morgan Kaufmann, Los Altos, CA, 1986.

[Chandrasekaran, 1983] B. Chandrasekaran. Towards a taxonomy of problem solving types. *AI Mag.*, 4(1):9–17, winter-spring 1983.

[Chandrasekaran, 1987] B. Chandrasekaran. Towards a functional architecture for intelligence based on general information processing tasks. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1183–1192, Milan, Italy, 1987.

[Charniak and McDermott, 1985] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, MA, 1985.

[Charniak, 1972] E. Charniak. *Toward a Model of Children's Story Comprehension.* PhD thesis, MIT, Cambridge, MA, 1972.

[Chi *et al.*, 1989] M. Chi, M. Bassok, M. Lewis, P. Reimann, and R. Glaser. Self-explanations: How students study and use examples to solve problems. *Cognitive Science*, 13:145–182, 1989.

[Chien, 1990] S. Chien. *An explanation-based learning approach to incremental planning.* PhD thesis, University of Illinois at Urbana-Champaign, 1990.

[Collins and Birnbaum, 1988a] G. Collins and L. Birnbaum. An explanation-based approach to the transfer of planning knowledge across domains. In *Proceedings of the 1988 AAAI Spring Symposium on Explanation-Based Learning*, pages 107–111, Palo Alto, CA, 1988.

[Collins and Birnbaum, 1988b] G. Collins and L. Birnbaum. Learning strategic concepts in competitive planning: An explanation-based approach to the transfer of knowledge across domains. Technical Report UIUCDCS-R-88-1443, University of Illinois, Urbana, IL, 1988.

[Collins *et al.*, 1989] G. Collins, L. Birnbaum, and B. Krulwich. An adaptive model of decision-making in planning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 511–516, Detroit, MI, 1989.

[Collins *et al.*, 1991a] G. Collins, L. Birnbaum, B. Krulwich, and M. Freed. A model-based approach to learning from planning failures. In *Notes of the AAAI Workshop on Model-Based Reasoning*, Anaheim, CA, 1991.

[Collins *et al.*, 1991b] G. Collins, L. Birnbaum, B. Krulwich, and M. Freed. Plan debugging in an intentional system. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 353–358, Sydney, Australia, 1991.

[Collins *et al.*, 1993] G. Collins, L. Birnbaum, B. Krulwich, and M. Freed. The role of self-models in learning to plan. In A. Meyrowitz and S. Chipman, editors, *Foundations of Knowledge Acquisition: Machine Learning*, pages 117–143. Kluwer press, Boston, MA, 1993. (Also appears as Technical Report #24, The Institute for the Learning Sciences, Northwestern University, 1992).

[Cox and Ram, 1992] M. Cox and A. Ram. Multistrategy learning with introspective meta-explanations. In *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland, 1992. Also appears as Georgia Tech report ER-92/03.

[Davis, 1980] R. Davis. Meta-rules: reasoning about control. *AI*, 15:179–222, 1980.

[Davis, 1984] R. Davis. Diagnostic reasoning based on structure and function: Paths of interaction and the locality principle. *Artificial Intelligence*, 24(1-3):347–410, 1984.

[Davis, 1990] R. Davis. *Representations of commonsense knowledge.* Morgan Kaufmann, San Mateo, CA, 1990.

[DeJong and Mooney, 1986] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1:145–176, January 1986.

[DeJong *et al.*, 1993] G. DeJong, M. Gervasio, and S. Bennett. On integrating machine learning with planning. In A. Meyrowitz and S. Chipman, editors, *Foundations of Knowledge Acquisition: Machine Learning*, pages 83–116. Kluwer press, Boston, MA, 1993.

[DeKleer and Brown, 1984] J. DeKleer and J.S. Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24(1-3):7–83, December 1984.

[deKleer *et al.*, 1977] J. deKleer, J. Doyle, G.L. Steele, and G.J. Sussman. Explicit control of reasoning. *SIGPLAN Notices*, 12(8), 1977.

[Doyle *et al.*, 1986] R.J. Doyle, D.J. Atkinson, and R.S. Doshi. Generating perception requests and expectations to verify the execution of plans. In *Proc. AAAI-86*, pages 81–87, Philadelphia, PA, August 1986. AAAI.

[Doyle, 1979] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[Etzioni, 1990] O. Etzioni. *A Structural Theory of Explanation-Based Learning.* PhD thesis, Carnegie Mellon University, 1990.

[Etzioni, 1991] O. Etzioni. STATIC: a problem-space compiler for PRODIGY. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 533–540, 1991.

[Fikes *et al.*, 1972] R.E. Fikes, P. E. Hart, and N.J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[Firby, 1989] R. James Firby. *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, Yale University, 1989. Available as Report RR-672.

[Freed *et al.*, 1992] M. Freed, B. Krulwich, L. Birnbaum, and G. Collins. Reasoning about performance intentions. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 242–247, Bloomington, IN, 1992.

[Freed, 1991] M. Freed. Learning strategic concepts from experience: A seven-stage process. In *Proceedings of the Thirteenth Annual Conference of The Cognitive Science Society*, pages 132–136, Chicago, IL, 1991.

[Genesereth, 1983] M.R. Genesereth. An overview of meta-level architecture. In *Proc. AAAI*, Washington, D.C., August 1983. AAAI.

[Golding and Rosenbloom, 1991] A. Golding and P. Rosenbloom. Improving rule-based systems through case-based reasoning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 22–27, 1991.

[Hammond, 1986a] K. Hammond. Chef: A model of case-based planning. In *Proc. AAAI-86*, pages 267–271, Philadelphia, PA, August 1986. AAAI.

[Hammond, 1986b] K. Hammond. Learning to anticipate and avoid planning problems through the explanation of failures. In *Proc. AAAI-86*, pages 556–560, Philadelphia, PA, August 1986. AAAI.

[Hammond, 1989] K. Hammond. *Case-based planning: Viewing planning as a memory task.* Academic Press, San Diego, CA, 1989. Also appears as Yale University Research Report #488.

[Hamscher and Davis, 1984] W. Hamscher and R. Davis. Diagnosing circuits with state: An inherently underconstrained problem. In *AAAI-84 Nat'l Conf.*, University of Texas at Austin, TX, August 1984. AAAI.

[Hayes-Roth and Hayes-Roth, 1979] B. Hayes-Roth and F. Hayes-Roth. Cognitive processes in planning. *Cognitive Science*, 3:275–310, 1979. Also in *Readings in Planning,* Allen, Hendler, and Tate, *eds.*, 1990.

[Hayes-Roth, 1982] F. Hayes-Roth. Using proofs and refutations to learn from experience. In *Machine Learning.* Tioga, Palo Alto, CA, 1982.

[Hendler, 1985] J. Hendler. Integrating marker-passing and problem solving. In *7th Annual Conference of the Cognitive Science Society.* Cognitive Science Society, August 1985. Also in *Readings in Planning,* Allen, Hendler, and Tate, *eds.*, 1990.

[Hewitt, 1969] C. Hewitt. Planner: A language for proving theorems in robots. In Walker, editor, *IJCAI-69*, pages 295–301, Washington, D.C., 1969. IJCAI, Mitre Co.

[Hinton *et al.*, 1984] G.E. Hinton, T.J. Sejnowski, and D.H. Ackley. Boltzman machines: Constraint satisfaction networks that learn. Technical Report Tech. Rept. CMU-CS-84-119, Carnegie-Mellon University, Pittsburgh, PA, 1984.

[Hunter, 1989] L. Hunter. *Knowledge-acquisition planning: Gaining expertise through experience.* PhD thesis, Yale University, 1989.

[Jones, 1992] E. Jones. *The flexible use of abstract knowledge in planning.* PhD thesis, Yale University, 1992. Published as Technical Report 28, The Institute for the Learning Sciences, Northwestern University.

[Kautz and Allen, 1986] H.A. Kautz and J.F. Allen. Generalized plan recognition. In *Proc. AAAI-86*, pages 32–37, Philadelphia, PA, August 1986. AAAI.

[Keller, 1987] R. Keller. *The role of explicit contextual knowledge in learning concepts to improve performance.* PhD thesis, Rutgers University, 1987. Technical report ML-TR-7.

[Keller, 1988a] R. Keller. Defining operationality for explanation-based learning. *Artificial Intelligence*, 35:227–241, 1988.

[Keller, 1988b] R. Keller. Operationality and generality in explanation-based learning: seperate dimensions or opposite endpoints? In *Proceedings of the 1988 AAAI Spring Symposium on Explanation-Based Learning*, pages 153–157, Palo Alto, CA, 1988.

[Knoblock, 1990] C. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928, 1990.

[Knoblock, 1991] C. Knoblock. *Automatically generating abstractions for problem-solving.* PhD thesis, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-120.

[Kolodner, 1987] J. Kolodner. Capitalizing on failure through case-based inference. In *Proceedings of the Ninth Annual Conference of The Cognitive Science Society*, pages 715–726, Seattle, WA, 1987.

[Krulwich *et al.*, 1989] B. Krulwich, G. Collins, and L. Birnbaum. Improving decision-making on the basis of experience. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 55–57, Ithaca, NY, 1989.

[Krulwich *et al.*, 1990a] B. Krulwich, L. Birnbaum, and G. Collins. Goal-directed diagnosis of expectation failures. In *Working notes of the 1990 AAAI Spring Symposium on Automated Abduction*, pages 116–119, Palo Alto, CA, 1990. Also appears in technical report 90-32, University of California, Irvine, Department of Information and Computer Science.

[Krulwich *et al.*, 1990b] B. Krulwich, G. Collins, and L. Birnbaum. Cross-domain transfer of planning strategies: Alternative approaches. In *Proceedings of the Twelfth Annual Conference of The Cognitive Science Society*, pages 954–961, Cambridge, MA, 1990.

[Krulwich *et al.*, 1992a] B. Krulwich, L. Birnbaum, and G. Collins. Learning several lessons from one experience. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 242–247, Bloomington, IN, 1992.

[Krulwich *et al.*, 1992b] B. Krulwich, L. Birnbaum, and G. Collins. Reasoning about multiple uses of learned concepts. In *Proceedings of the 1992 AAAI Spring Symposium on Knowledge Assimilation*, pages 89–96, Palo Alto, CA, 1992.

[Krulwich, 1991a] B. Krulwich. Determining what to learn in a multi-component planning system. In *Proceedings of the Thirteenth Annual Conference of The Cognitive Science Society*, pages 102–107, Chicago, IL, 1991.

[Krulwich, 1991b] B. Krulwich. Learning from deliberated reactivity. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 318–322, Evanston, IL, 1991.

[Kuokka, 1990] D. Kuokka. *The deliberative integration of planning, execution, and learning.* PhD thesis, Carnegie Mellon University, 1990. Technical report CMU-CS-90-135.

[Laird and Newell, 1983] J. Laird and A. Newell. A universal weak method: Summary of results. In *IJCAI-83*, pages 771–773, Karlsruhe, West Germany, August 1983.

[Laird *et al.*, 1986a] J. Laird, P. Rosenbloom, and A. Newell. Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986. Also appears in [Shavlik and Diettrich, 1990], and as Carnegie Mellon technical report CMU-CS-85-154.

[Laird *et al.*, 1986b] J. Laird, P. Rosenbloom, and A Newell. *Universal Subjoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies.* Kluwer Academic Publishers, 1986.

[Laird, 1983] John E. Laird. *Universal Subgoaling.* PhD thesis, Computer Science Department, Carnegie-Mellon University, 1983.

[Leake and Ram, 1993] D. Leake and A. Ram. Goal driven learning: Fundamental issues and symposium report. Technical Report 85, Indiana University, 1993.

[Leake, 1988] D. Leake. Evaluating explanations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 251–255, St. Paul, MN, 1988.

[Levinson *et al.*, 1991] R. Levinson, F. Hsu, T. A. Marsland, J. Schaeffer, and D. Wilkins. The role of chess in artificial intelligence research. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 547–552, 1991.

[London and Clancey, 1982] R.L. London and W. J. Clancey. Plan recognition strategies in student modeling: prediction and description. In *Proc. AAAI-82*, pages 335–338, 1982. (Also, Stanford University technical reports CS-82-909, HPP memo 82-7).

[McDermott, 1978] D. McDermott. Planning and acting. *Cognitive Science*, 2:71–109, 1978. Also appears in [Allen *et al.*, 1990].

[McDermott, 1989] D. McDermott. A general framework for reason maintanance. Technical Report YALEU/CSD/RR-691, Yale University Department of Computer Science, 1989.

[Michalski, 1983] R. S. Michalski. A theory and methodology of inductive learning. *AI*, 20(3):111–161, February 1983.

[Minton *et al.*, 1989] S. Minton, C. Knoblock, D. Kuokka, Y. Gil, R. Joseph, and J. Carbonell. Prodigy 2.0: The manual and tutorial. Technical Report CMUCS-89-146, Carnegie Mellon University, 1989.

[Minton, 1984] Steven Minton. Constraint-based generalization: Learning game-playing plans from single examples. In *Proceedings of the Third National Conference on Artificial Intelligence*, University of Texas at Austin, TX, August 1984.

[Minton, 1988a] S. Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, Carnegie Mellon University, 1988. Technical report CMU-CS-88-133. Also published by Kluwer Publishing company, 1988.

[Minton, 1988b] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569, 1988.

[Minton, 1990] S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–392, 1990. Revised version from AAAI-88; also appears in [Shavlik and Diettrich, 1990].

[Mitchell *et al.*, 1986] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), January 1986.

[Mitchell, 1982] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, March 1982. Also in *Readings in Machine Learning,* Shavlik and Dietterich, *eds.*

[Mitchell, 1990] T. Mitchell. Becomming increasingly reactive. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1051–1058, 1990.

[Mooney *et al.*, 1989] R. Mooney, J. Shavlik, G. Towell, and A. Gove. An experimental comparison of symbolic and connectionist learning algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 775–780, 1989. Also appears in [Shavlik and Diettrich, 1990].

[Mooney, 1988] R. Mooney. Generalizing the order of operators in macro-operators. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 270–283, 1988.

[Mostow, 1981] D. J. Mostow. *Mechanical Transformation of Task Heuristics into Operational Procedures.* PhD thesis, Carnegie-Mellon University, 1981.

[Mostow, 1983] D. J. Mostow. Machine transformation of advice into a heuristoc search procedure. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An artificial intelligence approach*, pages 367–403. Tioga Press, Palo Alto, 1983.

[Newell, 1990] A. Newell. *Unified theories of cognition.* Harvard University Press, Cambridge, MA, 1990.

[Ortony and Partridge, 1987] A. Ortony and D. Partridge. Surprisingness and expectation failure: What's the difference? In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 106–108, Milan, Italy, 1987.

[Pearlmutter and Hinton, 1986] B. Pearlmutter and G. Hinton. G-maximization: An unsupervised learning procedure for discovering regularities. In *Proceedings of the conference on neural networks for computing.* American Institute of Physics, 1986.

[Pryor and Collins, 1993] L. Pryor and G. Collins. Cassandra: Planning for contingencies. Technical report, Institute for the Learning Sciences, Northwestern University, 1993.

[Quinlan, 1986] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1:81–106, 1986. Also appears in [Shavlik and Diettrich, 1990].

[Ram and Cox, 1993] A. Ram and M. Cox. Introspective reasoning using meta-explanations for multistrategy learning. In *Machine Learning: A Multistrategy Approach*. Morgan Kaufmann, 1993. Also appears as Georgia Tech report GIT-CC-92/19.

[Ram and Hunter, 1992] A. Ram and L. Hunter. The use of explicit goals for knowledge to guide inference and learning. *Applied Intelligence*, 2(1):47–73, 1992. Also appears as Georgia Tech report GIT-CC-92/04.

[Ram, 1989] A. Ram. *Question-driven Understanding: An integrated theory of story understanding, memory, and learning*. PhD thesis, Yale University, 1989.

[Reiter, 1978] R. Reiter. On reasoning by default. In Waltz, editor, *TINLAP-2 (Theoretical Issues in Natural Language Processing)*, pages 210–218. University of Illinois, July 1978. Also appears in [Brachman and Levesque, 1985].

[Riesbeck and Schank, 1989] C. Riesbeck and R. Schank. *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Hillsdale, N.J., 1989.

[Rosenbloom and Laird, 1986] P.S. Rosenbloom and J.E. Laird. Mapping explanation-based generalization onto Soar. In *Proc. AAAI-86*, pages 561–567, Philadelphia, PA, August 1986. AAAI.

[Roussel, 1975] P. Roussel. *Prolog: Manual de reference et d'utilisation*, 1975. Groupe d'Intelligence Artificielle, Marseille-Luminy; September.

[Rumelhart and Zipser, 1985] D. Rumelhart and D. Zipser. Feature discovery by competitive learning. *Cognitive Science*, 9:75–112, 1985. Also appears in *Parallel Distributed Processing: Volume 1*, Rumelhart and McClelland, *eds.;* also appears in [Shavlik and Diettrich, 1990].

[Rumelhart *et al.*, 1986] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propogation. In D. Rumelhart and J. McClelland, editors, *Parallel distributed processing: Volume 1*, pages 318–362. MIT Press, Cambridge, MA, 1986. Also appears in [Shavlik and Diettrich, 1990], pp. 115-137.

[Sacerdoti, 1975] E.D. Sacerdoti. The non-linear nature of plans. In *IJCAI-75*, pages 206–214, Tbilisi, Georgia, USSR, 1975. IJCAI. Also Tech. Note 101, SRI.

[Schank and Abelson, 1975] R. C. Schank and R. P. Abelson. *Scripts, Plans, Goals, and Understanding*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1975.

[Schank *et al.*, 1986] R. Schank, G. Collins, and L. Hunter. Transcending inductive category formation in learning. *The Behavioral and Brain Sciences*, 9:639–686, 1986.

[Schank, 1982] R.C. Schank. *Dynamic Memory*. Cambridge University Press, Cambridge, England, 1982.

[Schmidt and Sridharan, 1977] C.F. Schmidt and N.S. Sridharan. Plan recognition: Using the hypothesize and revise paradigm. In *Proc. IJCAI-77*, MIT, Cambridge, MA, August 1977. IJCAI. Also DCS, Rutgers University, CBM-TR-77, March 1977.

[Segre, 1987] A. Segre. On the operationality/generality trade-off in explanation-based learning. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 242–248, Milan, Italy, 1987.

[Segre, 1988] A. Segre. Operationality and real-world plans. In *Proceedings of the 1988 AAAI Spring Symposium on Explanation-Based Learning*, pages 158–163, Palo Alto, CA, 1988.

[Shavlik and Diettrich, 1990] J. Shavlik and T. Diettrich, editors. *Readings in Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1990.

[Shavlik and Towell, 1989] J. Shavlik and G. Towell. An approach to combining explanation-based and neural learning algorithms. *Connection Science*, 1(3), 1989. Also appears in [Shavlik and Diettrich, 1990].

[Shavlik, 1990] J. Shavlik. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5:39–70, 1990. Also appears in [Shavlik and Diettrich, 1990].

[Simmons, 1988a] R. Simmons. *Combining associational and causal reasoning to solve interpretation and planning problems.* PhD thesis, MIT AI Lab, 1988.

[Simmons, 1988b] R. Simmons. A theory of debugging plans and interpretations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, St. Paul, MN., 1988.

[Simmons, 1991] R. Simmons. Coordinating planning, perception, and action for mobile robots. In *Working notes of the 1991 AAAI Spring Symposium on Integrated Intelligent Architectures*, pages 146–150, Palo Alto, CA, 1991.

[Smith *et al.*, 1985] R. G. Smith, H.A. Winston, T. Mitchell, and B.G. Buchanan. Representation and use of explicit justifications for knowledge base refinement. In *Proc. IJCAI-85*, Los Angeles, August 1985.

[Stallman and Sussman, 1977] R.M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, October 1977. Reprinted in 'AI-Massachusetts Institute Technology', Vol. 1,pp.31-91. Also MIT AI Memo 380, 1976.

[Sussman, 1974] G. Sussman. The virtuous nature of bugs. In *First conference of the Society for the Study of AI and and the Simulation of Behavior*, Sussex University, UK, 1974. Also in *Readings in Planning,* Allen *et. al., eds.,* pp. 111-117.

[Sussman, 1975] G.J. Sussman. *A Computer Model of Skill Acquisition.* American Elsevier, New York, 1975. based on Phd thesis, MIT, Cambridge, MA, 1973.

[Tambe and Rosenbloom, 1988a] M. Tambe and P. Rosenbloom. Eliminating expensive chunks. Technical Report CMU-CS-88-189, Carnegie Mellon University, 1988.

[Tambe and Rosenbloom, 1988b] M. Tambe and P. Rosenbloom. Why some chunks are expensive. Technical Report CMU-CS-88-103, Carnegie Mellon University, 1988.

[Tate, 1977] A. Tate. Generating project networks. In *IJCAI-77*, pages 888–893, MIT, Cambridge, MA, 1977. IJCAI.

[Utgoff, 1989] P. E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161–186, 1989.

[Warren, 1976] D. H. D. Warren. Generating conditional plans and programs. In *Proceedings of the AISB Summer Conference*, pages 344–354, University of Edinburgh, UK, 1976.

[Weintraub and Bylander, 1991] M. Weintraub and T. Bylander. Generating error candidates for assigning blame in a knowledge base. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 33–37, 1991.

[Wilensky, 1978] R. Wilensky. Why john married mary: Understanding stories involving recurring goals. *Cognitive Science*, 2:235–266, 1978.

[Wilensky, 1981] R. Wilensky. A model for planning in complex systems. *Cognition and Brain Theory*, 4(4), 1981. Also appears in [Allen *et al.*, 1990] and as Technical Memo UCB/ERL-M81/49.

[Wilkins, 1984] D. Wilkins. Domain-inderendent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984. Also appears in [Allen *et al.*, 1990].

[Winston, 1975] P.H. Winston. Learning structural descriptions from examples. In P.H. Winston, editor, *The Psychology of Computer Vision*, chapter 5, pages 157–209. McGraw Hill, New York, 1975. based on Phd thesis, MIT, 1970.