An Expert System for Temporal Planning with an Application to Runway Configuration Management

by

Lyman Robert Hazelton, Jr.

M.S., Theoretical Physics, University of Miami, 1972
B.S., Theoretical Physics, University of Miami, 1968

Submitted to the Department of Aeronautics and Astronautics
as Recommended by an Interdepartmental Committee
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Flight Transportation and Artificial Intelligence
at the
Massachusetts Institute of Technology

February 1991

©Massachusetts Institute of Technology 1991
All rights reserved

Signature of Author _____
Department of Aeronautics and Astronautics
December 12, 1990

Certified by _____
Professor Robert W. Simpson, Department of Aeronautics and Astronautics
Thesis Committee Chairman

Certified by _____
Professor Ramesh Patil
Department of Electrical Engineering and Computer Science

Certified by _____
Dr. Dennis F. X. Mathaisel
Department of Aeronautics and Astronautics

Certified by _____
Dr. John D. Pararas
Department of Aeronautics and Astronautics

Accepted by _____
Professor Harold Y. Wachman
Chairman, Department Graduate Committee

Aero
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
FEB 19 1991

An Expert System for Temporal Planning with an Application to Runway Configuration Management

Lyman R. Hazelton, Jr.[1]

Flight Transportation Laboratory
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology

January 7, 1991

[1]Research supported by NASA contract number NGL 22-009-640.

Abstract

This thesis describes an expert system to aid in the management of operations in complex qualitative domains characterized by multiple parallel activities with time-critical relationships.

An extension to "standard" temporal logic required for reasoning about inferred allocation of resources and a detailed representation of temporally dependent facts, including persistence, is presented. The non-linear planning paradigm commonly used in planning programs is extended into the temporal domain to facilitate scheduling as well as ordering of plan steps. This enhancement requires new structures and analytical methods for the detection and resolution of serendipitous interactions and conflicts between proposed schedules. A computer implementation of these concepts is discussed in detail.

The expert system is organized into three modules: the *time map manager* or temporal database manager which stores, organizes, and retrieves time dependent knowledge; the *temporal system analyzer* which uses this knowledge to forecast and analyze domain dynamics; and the *planner/scheduler* which formulates and schedules activities in order to satisfy goals generated by the temporal system analyzer.

Finally, *Tower Chief*, an application of the system to scheduling runway configuration changes and maintenance at large airports, is described.

Preface

This project turned out to be about twice as difficult as I thought it would when Bob Simpson and I first discussed it back in late 1984. At the start, I was more worried about making a nice graphic user interface than I was about the internal structures and processes that would be necessary in order to provide a powerful enough reasoning engine. As I began to understand the problem better, I became aware that the critical obstacle was temporal reasoning. But I had no idea how fundamental reasoning about time is to human cognition, nor how complicated the thought processes associated with time really are. Thus "two years to a field test" became "five years to a proof of concept."

I could not have done it alone. There were times when I wavered, times when I was stymied, and times when my resolve was all but gone. In every case, there have been people who have freely given me their time, their ideas, and their support. There are far too many to list them all here, but the following individuals each played a crucial role in making this dissertation a success:

Robert W. Simpson had the original idea of attempting to create a *Tower Chief* advisor, and coined the name of the system. His careful evaluation and criticism of my ideas kept me centered on the problem and his patience allowed me the time necessary to attack it.

Ramesh Patil provided pointers to important publications as well as ideas and corrections. He was always there when I needed him, and even a few minutes of his time invariably helped me greatly.

Antonio Elias was my mentor during the first year of the research. His excitement and the brilliance of his genius are an inspiration to me even now.

Dennis Mathaisel supplied invaluable assistance by actually applying the system to the classroom scheduling problem. In addition, he spent hours listening to my ideas and helping me clarify them.

John Pararas helped with the programming, shared ideas, listened carefully to my explanations of the work, and then explained it back to me so that I understood better what I had done.

Jacqueline Fritz read and corrected the manuscript, listened to my ideas, made suggestions for figures, and kept me sane, healthy, and loved. Her tenacity and patience are without peer. I met her just as I started this endeavor, and she has persevered through it all.

Bill Howell spent most of a semester at MIT listening to my ideas and giving me expert advice about management. He was also responsible for NASA funding of the entire effort through the FAA/NASA Joint University Grants Program.

Paul Brown gave me a comfortable place to work, listened to my ideas, read and commented on several papers, fed me, and allowed me to keep my wings.

"Buck" Farquhar arranged for me to enter and observe the operation of the Boston *TRACON*, introduced me to the magnificent staff there, and spent hours explaining how the system at Logan *really* works.

The entire staff of the **Boston TRACON**, who went out of their way to include me in every facet of the operation over a period of months.

Sandy Alexander listened carefully to my ideas on numerous bus rides to and from the Institute, and provided thoughtful, penetrating, and supportive comments.

Solly Ezekiel gave me ideas, friendship, enthusiasm, and support.

Mark Kolb helped with the programming and granted a patient ear during the exciting times when I had a breakthrough and just *had* to tell someone right away.

Lori Martinez did me many favors and gave me much good advice.

Bob Bruen listened to my ideas, was a consistent, solid friend, and kept the computers going so I could keep going.

Liz Zotos kept me informed and took care of the many administrative details necessary to actually receive the degree.

Marvin Minski was supportive of the project and made some good suggestions about literature I should read.

Bruce Ettenberg gave me references on the history of logic and was a constant source of enthusiasm and friendship.

Tom Russ showed me how to get LaTeX to gobble down PostScript files, which is how all the figures were produced in this document. Tom also shared his own views and conundrums while trying to solve a similar problem in medical care management.

Yoav Shoham took the time to clear up some confusion I had about his thesis and later work which was of paramount importance to this effort.

Guy Steele helped me debug the resource manager program when I upgraded it to be compatible with the new definition of Common Lisp.

Thank you, all of you. I shall carry my appreciation of your efforts with me always.



Cambridge, MA

Contents

1 Introduction **12**

1.1 Operations Management 13

1.2 Planning and Scheduling 16

1.3 Historical Background 17

1.3.1 Domain Independent Planning 18

1.3.2 An Example of Conjunctive Goal Planning 19

1.4 Requirements 30

1.4.1 Development History 35

2 Temporal Logic **39**

2.1 Propositions 40

2.1.1 Compound Propositions 40

2.2 Rules 41

2.2.1 Quantification 43

2.3 Default Logic 44

2.4 Defeasible Logics and Truth Maintenance 50

2.5 Temporally Dependent Propositions 52

2.5.1 Evanescence and Persistence 54

2.5.2 The Effects of Time on Defeasibility 55

2.6 Summary and Preview 58

3 The Time Map Manager **60**

3.1 A Structure for History 61
3.2 TIMEBOX 63
3.2.1 Temporal Relationships 64
3.2.2 Replacement 67
3.2.3 Addition 74
3.2.4 Functions for Updates 74
3.2.5 Denial of Information 78
3.2.6 Temporal Extent of Inferred Information 79
3.3 Summary 86

4 The Temporal System Analyzer **88**

4.1 Composition of Rule System Objects 89
4.2 Structure for Rules 90
4.3 Patterns with Variables 91
4.4 RULESYS 93
4.4.1 Creation of Rules 97
4.4.2 Rule Evaluation 100
4.4.3 Universal Quantification, Set Forming Rules, and Preference Rules 103
4.4.4 Existential Quantification, Cut, and Default Rules 104
4.4.5 Assumptions and Paradoxes 106
4.4.6 Disjunction and Run-rule 107
4.4.7 Requests to the Planner/Scheduler 108
4.4.8 Order of Execution 108
4.5 Reports and Queries 109
4.5.1 When Things Get Really Bad 113
4.6 System Resources 114
4.6.1 Managed Resources 114
4.6.2 Long and Short Term Memory 115

4.7 Summary 115

5 The Scheduler **117**

5.1 Historical Background 118

5.2 A Scheduler 120

5.3 Handling Uncertainty with Plan Refinement 125

5.4 Operation of the Scheduler 127

5.5 The Plan and Procedure Editors 128

5.6 Summary 133

6 Case Study **135**

6.1 Runway Configuration Management 135

6.2 Physical Description of Logan 141

6.3 Transition from Enroute to Terminal Area 145

6.4 The TRACON 146

6.5 Logan Configurations 147

6.5.1 Multiple Runway Configurations 148

6.5.2 Single Runway Configurations 153

6.5.3 Rule System Representation of Logan 153

6.6 Configuration Selection Rules 155

6.6.1 Weather Rules 156

6.6.2 Runway Availability 159

6.6.3 Dwell 160

6.6.4 Shift Changes and Workload 161

6.7 An Initial Rule Set 162

6.8 Scheduling Planned Sequences of Operations 167

6.9 Results 171

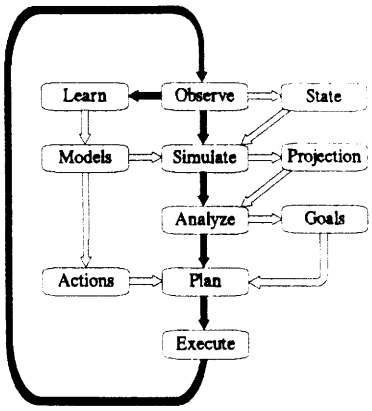
7 Conclusion **177**

7.1 Recapitulation of the Thesis 177

7.2 Areas for Further Research 180

To Mary Ann Moore,

who opened my eyes.



Chapter 1

Introduction

It is two o'clock on a hot, partly cloudy Friday afternoon at Boston's Logan International Airport. There is a fifteen knot breeze coming from the East with gusts up to twenty one knots. The airport is relatively calm now, but this will change. From three o'clock until eight in the evening, there will be over four hundred and fifty aircraft operations,[1] with peak activities of more than three operations per minute. The National Weather Service has forecast the arrival of a cold front, with an associated line of thunderstorms and switch in wind direction, sometime around six o'clock. A group of heavy international flights is due to arrive at about the same time. One of the runway's approach light system needs maintenance to replace two of the high intensity flash units. Unfortunately, it is the primary approach runway in use now.

Which runways should be used during the rest of the afternoon and evening? When should the changes from one runway system to another be made? When, if at all today, should the maintenance crews be allowed to fix the approach light system? Will the available resources be able to handle the onslaught of aircraft demanding use of the airport without major delays?

[1] A takeoff or landing.

1.1 Operations Management

Determining the answers to these questions is part of the overall task of managing the resources of the airport and scheduling their use. Many of the tasks that people face every day may be described as *operations management*. Examples of this kind of task are strategic control of a petro-chemical refining plant, running an electric power generation facility, or runway configuration management at large airports. Process management is characterized by the desire to control some system which operates in a dynamically changing time domain. Unlike a classical feedback control system, which is purely *reactive*, a manager can generally see (perhaps myopically) not only the state of the domain at the present moment, but into the future as well. The basic "goal" of a manager is to "keep the system going." Processes like those mentioned above are, at least theoretically, never ending unless catastrophe strikes. At the petro-chemical plant, crude oil flows into the system at one end, and a variety of chemical products flows out of the other. While deliveries of crude are not continuous, the feed from the stock they supply may be considered continuously available and the plant to be always in operation. A power generation facility must provide power at all times. This does not mean that none of the generators can ever be shut down (or break down), but it does mean that *when* they may be shutdown and for *how long* should be determined in advance. Large airports are supposed to stay in operation twenty four hours a day, every day. Again, this means that the order in which runways or other facilities are removed from service, the times that the closings will occur, and the length of time that any resource will not be available should be scheduled ahead of time.

Defining intelligent behavior is a difficult task because intelligence includes so many diverse activities, such as tool building, communication, self awareness, planning, and so on. Perhaps the most primitive aspect of intelligence is the ability to observe one's environment, create a mental model of the environment based on the observations, use the model as a simulation to predict future states of the environment, and make plans

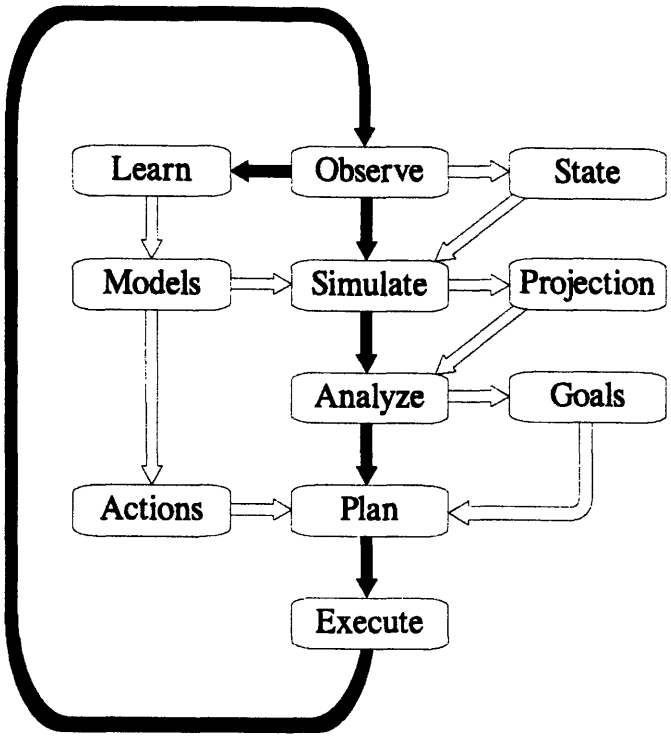


Figure 1.1: Intelligent Behavior Loop

involving actions in order to change the predicted behavior to something the observer believes will be more desirable (Figure 1.1). A continuous loop of this kind may be the basis for all intelligent behavior.

An operations manager generally has a *model* of the management domain. The model is used to project the domain operations starting from the present *known* state so that the manager can predict the future states the domain will attain. The process of running a model of a domain to predict its future is called *projection*. If the projection appears to the manager's liking, nothing further need be done. The manager can relax for a while and analyze the projection of the domain again at some later time. If, on the other hand, the projection reveals unacceptable states, the manager seeks to change the operating parameters so that the projected future states *are* acceptable. The determination of the changes to be made embody a *scheduled plan of actions* by the manager. The schedule thus devised is then executed in the *real* domain. Because the fidelity of the

projection over long periods is usually imperfect, differences (called *deviations*) usually appear between the projected future and the actual turn of events. Because of deviations, the projection is re-run from time to time with the most up-to-date observations as the starting point, and the resulting projections used to change the schedule if necessary.

Human managers use a mental model of the systems they oversee, and generally use previous experience and rules of thumb to project system activity. When deviations occur, they tend to avoid complete re-scheduling as much as possible, preferring to modify the present schedule or execution plan, because incremental modifications generally require fewer mental resources and less work to implement than generating and executing an entirely new schedule.

Most systems do not catastrophically fail if the management applied to them is less than perfect. Instead, they exhibit a range of degraded operation and inefficiency which increases as the management becomes worse. Others, such as nuclear power plant systems or tactical air traffic control, have failure modes which may be sudden and disastrous. Some systems are so large or complex that it is almost impossible to manage them properly when they are operated at peak through-put. These systems are conservatively operated at lower than maximal levels so that "slack" is introduced to avoid accidents.

Military command and control (C^2) is another example of an operations management problem. A commanding officer has intelligence and reconnaissance information which define the current strategic situation. This knowledge may include goals desired by enemy forces or enough evidence to allow a reasonable determination of them. Unlike the examples described above, C^2 normally has a final goal (to win) and a final state (to be the victor or the vanquished). While operations are in progress, however, this domain shares much with the previous examples. The commanding officer often has at least a mental model of the actions that both sides will take, and what parameters will effect the outcome of those actions. Projection leads to a picture of the future based on the current schedule plan (or lack of one), and the commander must create or update a

schedule plan to attain the goal.

Operations management is more than planning or scheduling for a single goal or set of goals. As time passes and new information becomes available, the goals desired by a manager often must *change*. Part of the task that the manager faces is to set the proper goals for a given set of circumstances, and change the goals as changing circumstances require.

Generally, then, operations management involves determining what goals should be undertaken, and then constructing a schedule of actions which will reach them. Considerable research has been done in recent years to attempt to automate the task of *planning*. Because of the added complexity of time, less has been accomplished with regard to *scheduling*. This thesis describes an attempt to create an automatic operations scheduling system.

1.2 Planning and Scheduling

A *plan* is "a detailed scheme, program, or method, worked out before hand for the accomplishment of an objective" [AHD]. A plan is composed of a list of steps or actions which, when executed, will attain the objective. A plan is created within a *domain* which constitutes the relevant attributes of the environment in which the plan will be executed.

The creation of a plan is a distinctly cognitive process which can be fraught with difficulties. The difficulties arise when the planner must create a complex scheme with many sequential and parallel steps, when the goal is to accomplish more than one objective, or when knowledge of the objective or future states of the domain are not known with certainty.

A plan which includes decisions constraining the times when the steps or actions are to be executed or completed is a *schedule*. Complex scheduling problems are ubiquitous to modern life. They are common in manufacturing, transportation, communications, and medicine; all large industries, in which thousands of man-hours and millions of dollars

are spent each year trying to create better schedules.

Usually, the solutions of scheduling problems prepared by human reasoners are *feasible* rather than optimal, because the number of disfunctional solutions is always much larger than the number of those that will actually work. Even the number of feasible solutions to real-world scheduling problems is astronomical. And, of course, out of that large number of feasible possibilities, only one or a few are optimal.

1.3 Historical Background

Philosophers and mathematicians have been studying the process of human cognition for eons [Bocheński 70]. Analysis of theorem proving techniques [DeLong 70] in the nineteenth century formalized the idea of automating at least some part of the reasoning process. The techniques of Operations Research, starting during World War II, formalized the idea of *optimality*, and developed methods to algorithmically search for solutions to some planning and scheduling problems.

The invention of the computer provided a vehicle to bring these ideas out of the purely theoretical realm of mathematics and apply them. As early as the late nineteen fifties, computer scientists such as John McCarthy [McCarthy 65] were beginning to study the possibilities of using computers to do symbol manipulation. Newell and Simon's [Newell 63] "General Problem Solver" (GPS) gave the first sign that perhaps a computer might actually create a plan *by itself* and initiated the field of *Artificial Intelligence*(AI). The descendants of GPS have divided into two major camps in the AI community: Expert Systems and Planning. In the expert systems arena, the major successes include the MYCIN [Shortliffe 76] medical consultant, the PROSPECTOR [Duda 79, Gaschnig 80] mineral exploration consultant, the XCON [McDermott 80] computer hardware configuration expert, the ACE [Vesonder 83] telephone cable maintenance adviser, and many, many more.

Research in planning has centered around trying to discover and understand the fun-

damental processes of planning in general, independent of the domain of application. It has resulted in a comprehensive mechanism, called *Domain Independent Planning*. Domain specific issues, such as solution justification, knowledge acquisition and representation, and human interface are usually added on top of a domain independent core.

1.3.1 Domain Independent Planning

Fikes and Nilsson [Fikes 71] actually created the first program that composed plans, though it did so in an unrealistically restricted domain. A few years later, Sussman [Sussman 75] established the "blocks world" as the standard domain in which planners are tested. It is an imaginary world populated by a robot and a set of blocks. "Blocks world" is a restricted domain, as well. There is only *one* robot, and it is constrained to take only one action at a time (i.e., the plan reduces to a sequence of operations). This is known as the "single actor assumption". In such a restricted world, nothing happens unless the robot acts to make it happen. Otherwise, the world is static. Thus the planner need only consider its goal and the current state of the world when creating a plan. Any plan that theoretically reaches the goal is guaranteed to succeed, because nothing can go wrong.

Expanding the capabilities of domain independent planners so that real-world problems can be attacked with these techniques has been very difficult. In great part, this difficulty arises from the fact that it is very hard to accurately predict the future, because for almost any rational prediction, there are infinitely many exceptions that may invalidate it. This is known as the "frame problem" (see, for example [McCarthy 69] and [Shoham 88, pages 16–19]).

The basic paradigm of domain independent planning may be summed up as follows: The planner is aware of the initial state of the domain and the desired final state (the goal). These states are characterized by a vector of propositions indicating the values of state variables or descriptors. The planner has a database of *operators* representing

actions. Each operator has a set of *preconditions* defining the circumstances under which the operator should be applicable, a *delete list* describing the propositions that the action of the operator will remove from the state vector which exists when the operator is applied, and an *add list* describing propositions which the action of the operator will append to the state vector. At any point during plan production, the database is searched to find all those operators which have preconditions matched by the current state. Each of the resulting subset of operators is applied to the current state, and each produces a different new state. If any of the new states is the goal state, the sequence of all the operators leading to it is the plan. Otherwise, each of the new states is treated in turn as a new current state, and the search and operator application process is used again. In this way, a tree or graph of potential plans is created until one of them reaches the goal. Like all graph traversing problems, there is no *apriori* mechanism to determine the best way to move through the tree.

Close scrutiny of this paradigm shows that it is not really as "domain independent" as it claims to be. The domain knowledge is hidden in the operator structures. If the preconditions of the operators are highly specific, so that very few operators are applicable to any given state, then the tree of possible plans will be narrow (have few members) and planning may terminate quickly. If the preconditions are less exclusive, then the number of potential plans may be enormous and the planning process may not terminate for a long time as a search is made through all these possibilities.

1.3.2 An Example of Conjunctive Goal Planning

The planners described above use the state of the domain at given times to represent the plan, and actions to represent changes to the state. The representations of actions used by almost all of them are variations of the STRIPS schemata introduced originally by Fikes and Nilsson [Fikes 72a]. Figure 1.2 shows the basic structure of some actions. A *goal* is represented by a statement which is asserted in the add-list of an action. For ex-

ample, "NEAR CEILING" is a goal which is satisfied by the "CLIMB LADDER" action. Complex domains are difficult to depict using this kind of representation [Lansky 87].

A better representation for planning was introduced in [Sacerdoti 75]. Rather than use individual domain states to represent plans, he used *partially elaborated plans*, or partial solutions. These are sets of actions which are pre-ordered and have a more general set of preconditions and effects. Sacerdoti was also the first to use a *network* to represent complex plans. The planning paradigm changes to finding a solution in terms of partial solutions and then expanding and ordering *or elaborating* them as more information constraining the order becomes available.

Such plans do not need to specify every effect that will occur when the plan is executed, and elaboration is deferred as long as possible. This is helpful, since the steps included and the effects of a plan depend heavily on the domain state that exists at the time of execution. Since the purpose of planning is to make decisions ahead of time, it is unusual that the planner knows with precision what the domain state will be at the time the plan will be executed. This representation directly describes the relationships between actions, and planners which use it are variously called action-ordering planners or, more commonly, *non-linear planners*. Almost all recent planning programs use this approach.

The primary problems with planner programs have been implementing efficient search algorithms and handling conjunctive goals. The search problem is common to many Artificial Intelligence and Operations Research problems. In the case of non-linear planners, the search starts initially for an action or a set of actions which asserts the current goal. Subsequent searches look for applicable reductions that assert the sub-goals of the initial action, and so on until the plan consists of nothing but primitive actions which, presumably, can be executed.

Handling conjunctive goals has been a much more difficult problem. The kernel of the problem is that actions required by a plan to accomplish one of the goals in the conjunct

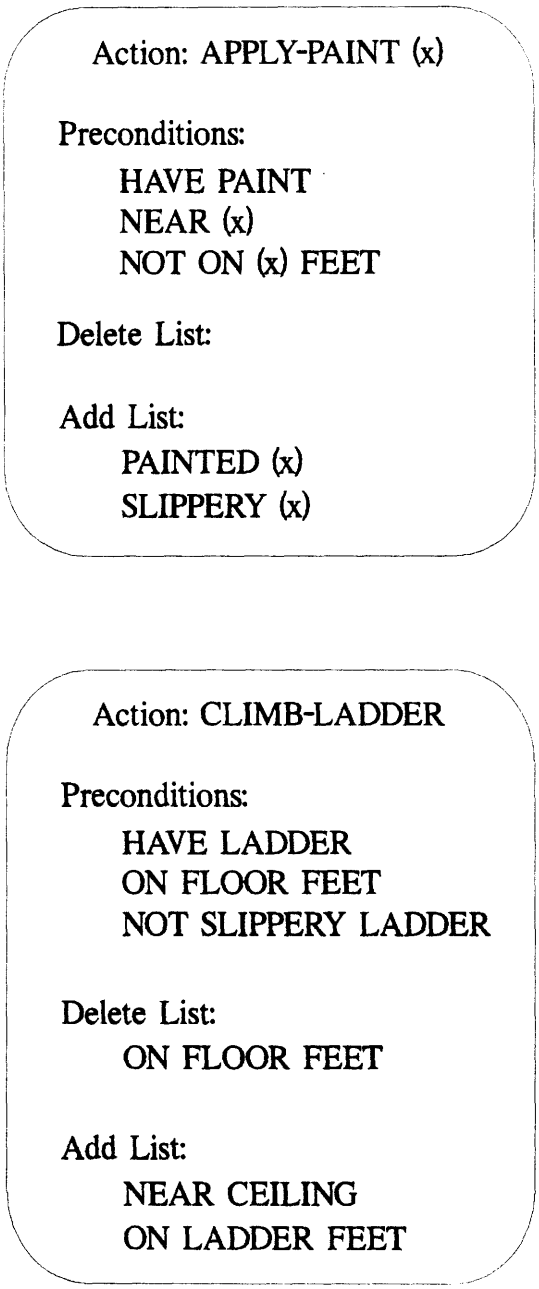


Figure 1.2: Classical Representation of Actions

may prevent the success of the plan to accomplish the other goal in the conjunct.

The standard example given for an interacting conjunctive goal is a robot problem: given the necessary tools and materials, such as paint, brushes, a ladder, and so on, the robot must plan how to accomplish the dual goal of painting the ceiling of a room and painting the ladder it must use to paint the ceiling. The problem is, of course, that the robot must decide to paint the ceiling *first*, and then paint the ladder.

Elementary plans are provided to the planner by the knowledge engineer as chains of actions and goals. Figure 1.3 exhibits two elementary plans. Any of the goals in such plans may be satisfied by either an action or another plan, depending on the granularity of the system. Further, there may be more than one possible action or plan which will satisfy a given goal.

The network representation of a plan requires special nodes for *splitting* and *joining* activities. To construct a plan to accomplish the conjunctive goal "PAINTED CEILING *and* PAINTED LADDER", the initial conjunct is *split* into separate goals (Figure 1.4 and 1.5). The procedural database is then searched for known plans which will satisfy these goals separately (Figure 1.6). The two sub-plans are expanded (elaborated) to form the network in Figure 1.7. The two paths through the net are examined to find redundant actions, and collapsed by moving the *split* node forward so that redundant actions are done only once (Figure 1.8). Next, an attempt is made to order (in a single path) the procedures remaining in parallel paths. A trial ordering is considered. If the result of a preceding procedure negates a precondition of the procedure following it, then the order of execution is changed so that the preconditions are no longer negated. In the example, if the initial ordering is described in Figure 1.9, it is found that a precondition for "CLIMB LADDER", "NOT SLIPPERY LADDER" is negated by the result of the action "APPLY-PAINT LADDER" which asserts "SLIPPERY LADDER". The order of execution is reversed and the analysis is repeated (Figure 1.10), but this time no contentions are found. If a contention were found after re-ordering the procedures, the

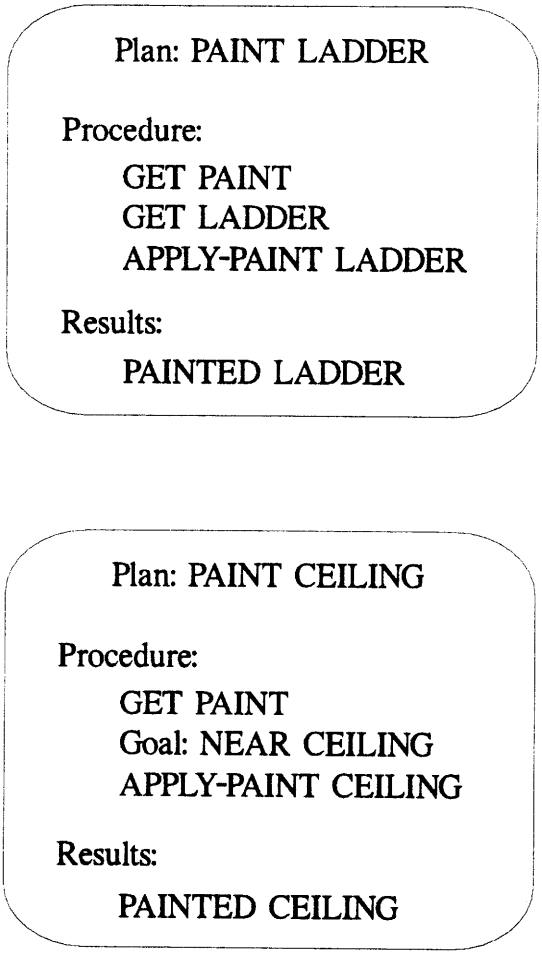


Figure 1.3: Elementary Plans

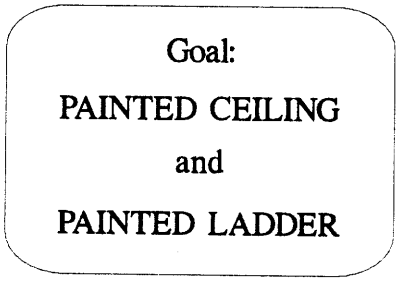


Figure 1.4: Conjunctive Goals

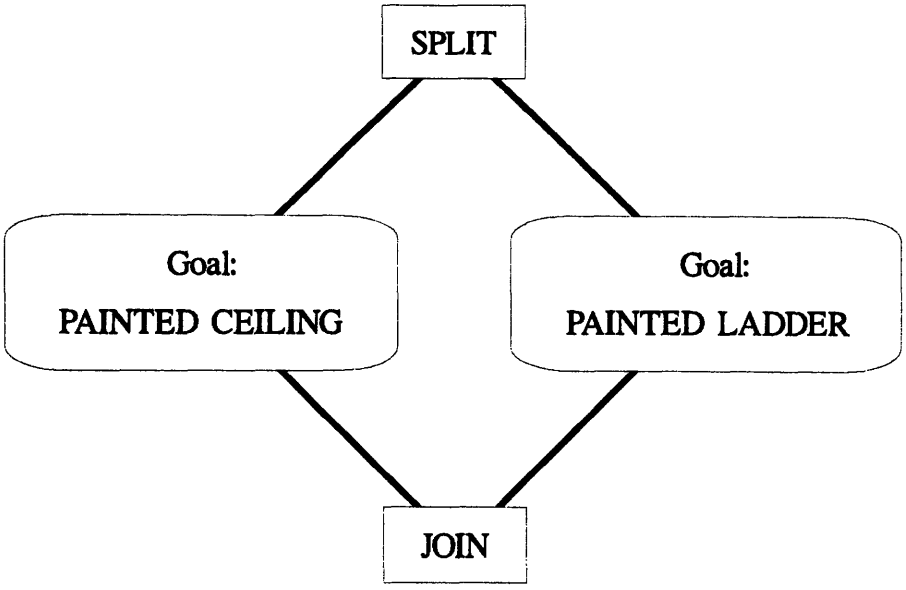


Figure 1.5: Split Conjunct into Network of Parallel Goals

system would be deadlocked. This situation is called a "double cross", and requires the use of special domain dependent rules for resolution.

"Double crosses" for which the knowledge engineer has not provided such special rules will either produce "loops" which never terminate or cause the planner to crash. David Chapman [Chapman 87a] has proven that planning for conjunctive goals requires exponential time in some cases, and that the creation of plans with conditional effects is undecidable (i.e., non-terminating). Most of the work in domain independent planning has been focused on generating feasible plans. Recently, Gupta and Nau [Gupta 90] have shown that finding an *optimal* plan using domain independent planning is NP-hard even for the restricted "blocks world" domain.

This is not to say that domain independent planning has been without success. In recent years, Wilkins [Wilkins 83] has built a system that plans aircraft carrier mission profiles. Vere [Vere 83] made a system for NASA which planned Voyager spacecraft mission sequencing. Tate's [Tate 77] original NONLIN electric generator turbine overhaul planner was expanded and improved by Tate and Whiter [Tate 84], and applied to

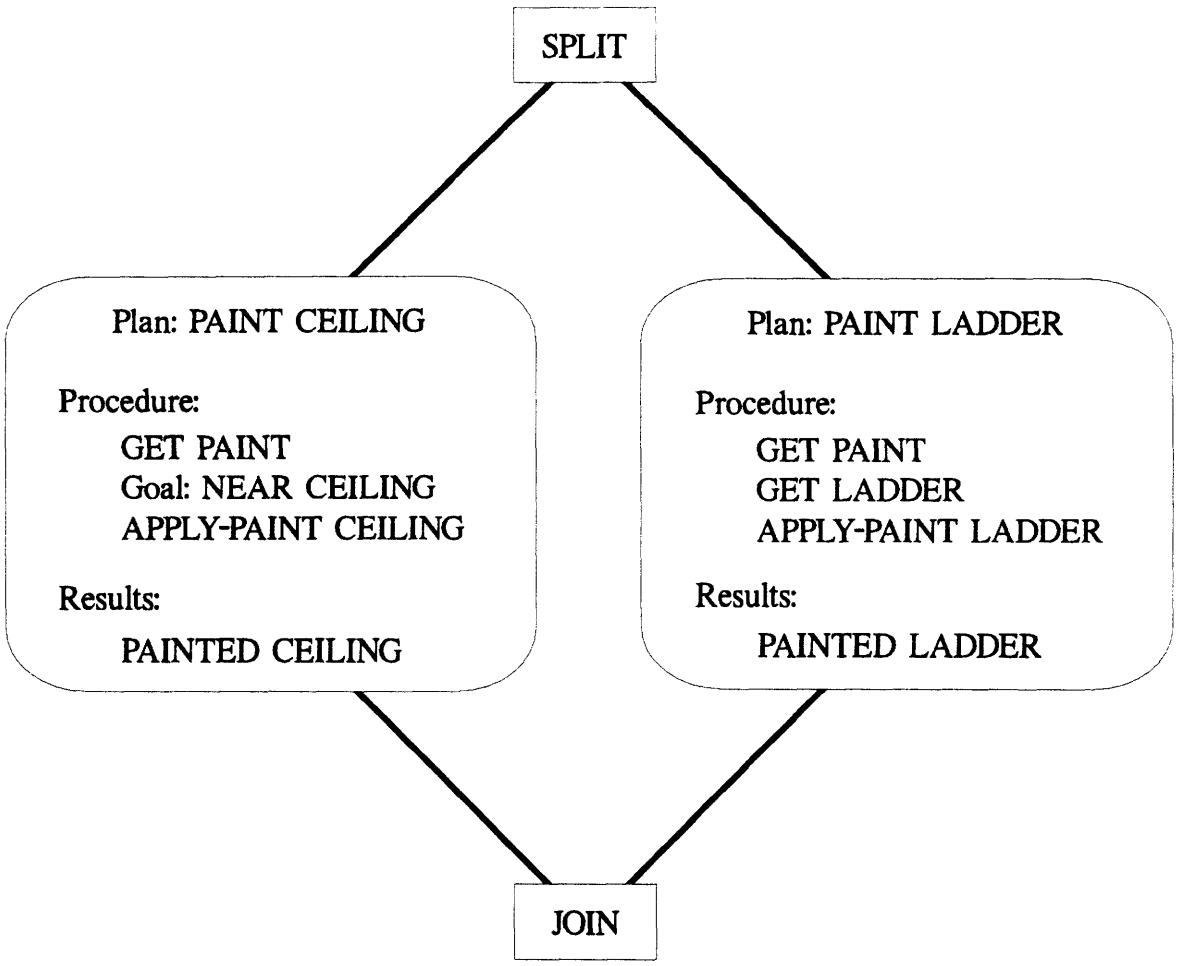


Figure 1.6: Use Known Plans

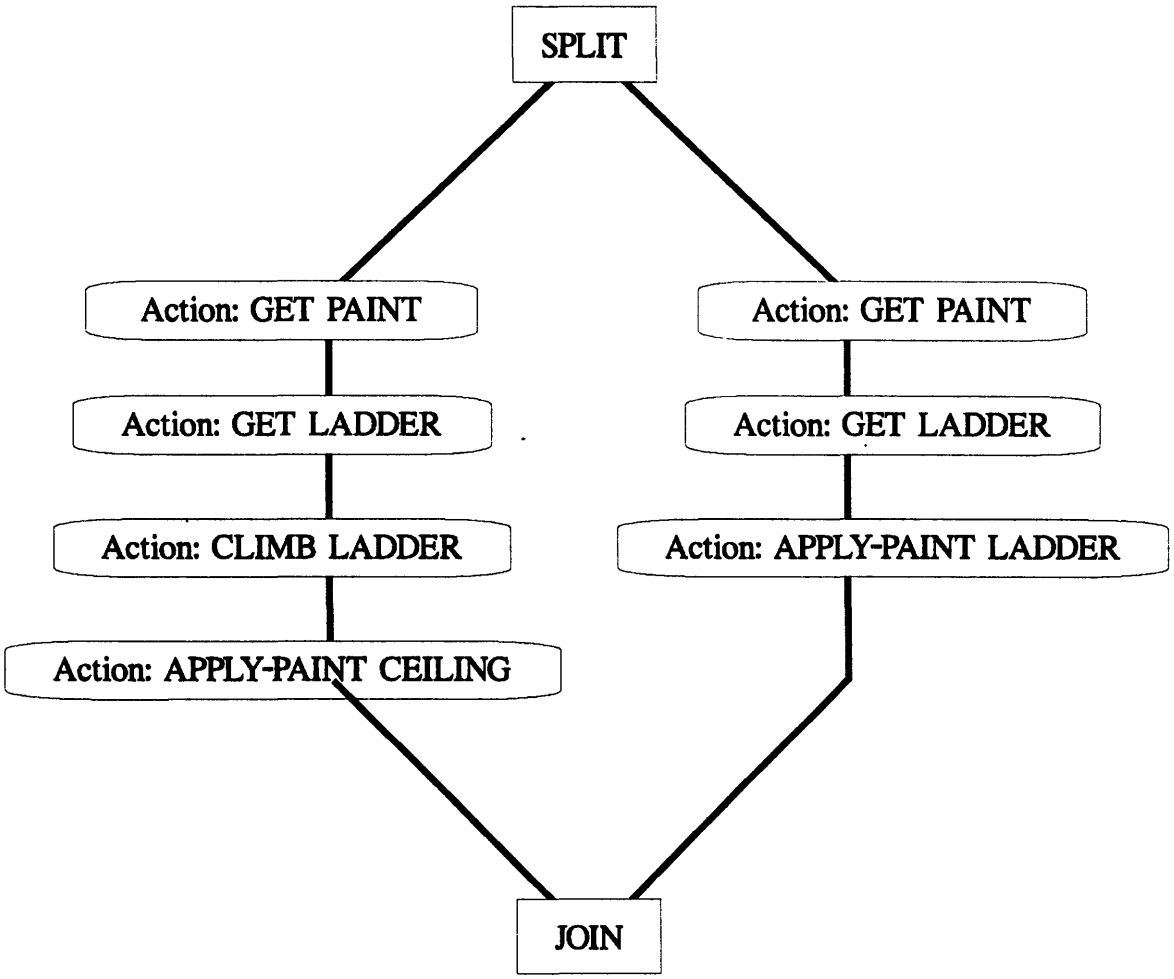


Figure 1.7: Elaborated

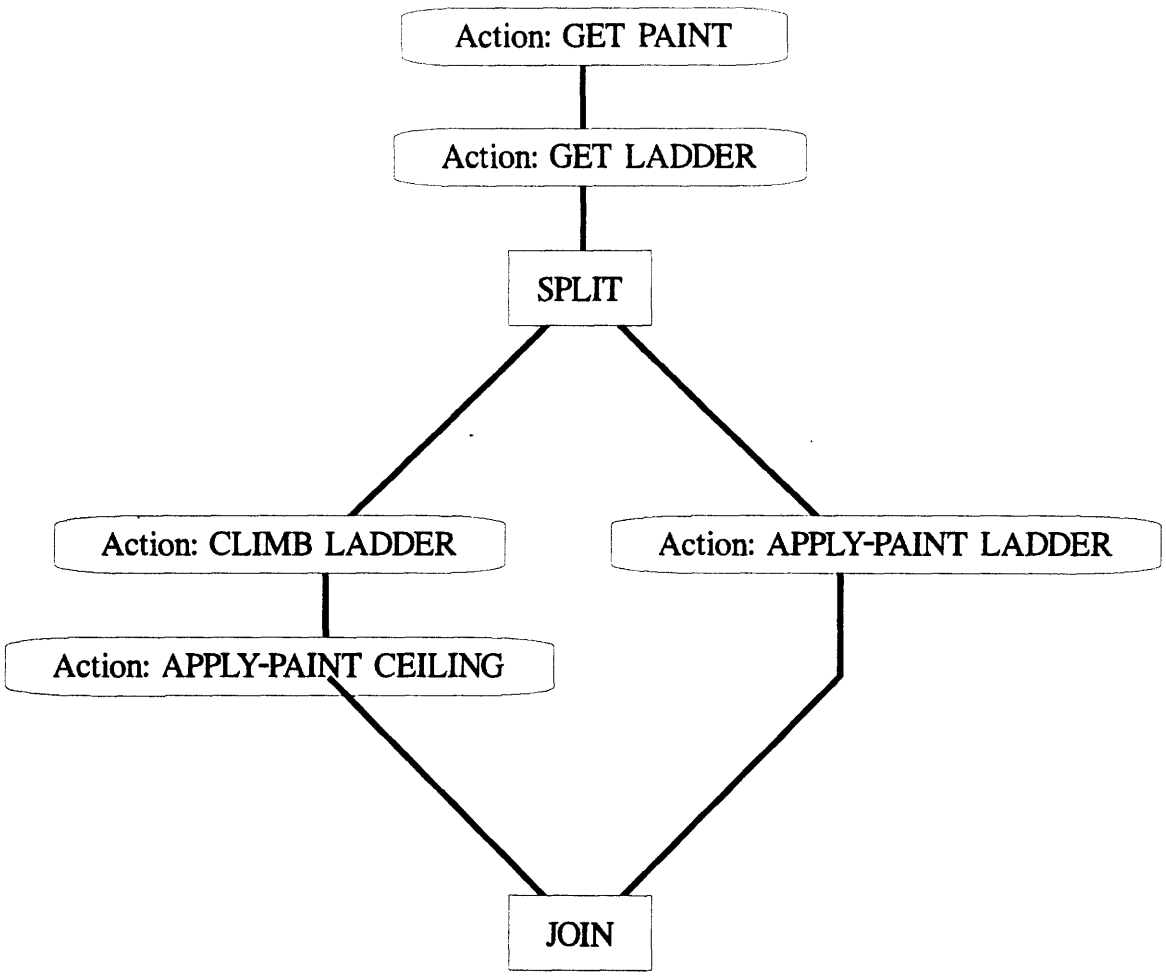


Figure 1.8: Redundancies Removed

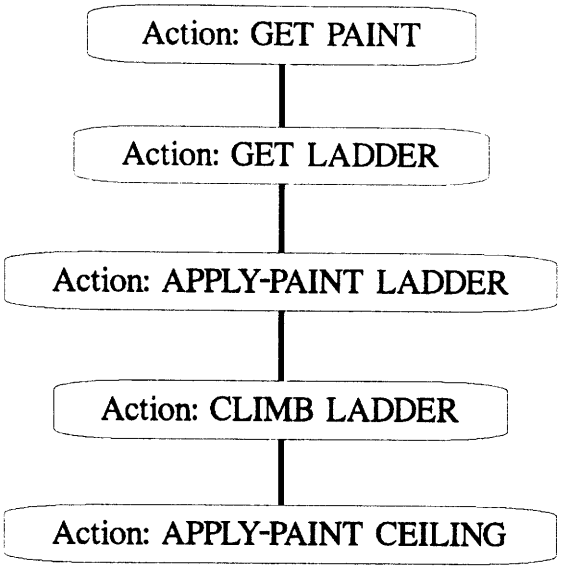


Figure 1.9: Possible, but Incorrect Ordering

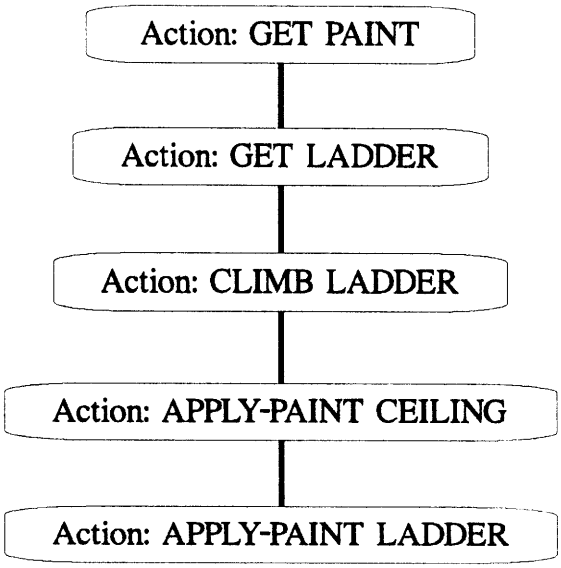


Figure 1.10: Correct Ordering

planning naval logistics. Miller, Firby, and Dean [Miller 85] constructed a robot activity planner for a factory which takes into account transit times required as the robot moves about.

However, all of these planners succeeded because they were designed to operate in worlds which are severely restricted. Charniak and McDermott state the restrictions well [Charniak 85, page 499]:

- Nothing happens unless the execution of the plan makes it happen.
- The effects of the actions involved with plan execution are predictable and instantaneous, and can be modeled in terms of "add-lists and delete-lists".
- Only *one* primitive action can happen at a time.

The reason for employing these restrictions is the so called "frame problem". This problem was first brought to light by John McCarthy and Patrick Hayes [McCarthy 69]. Put in its simplest terms, the frame problem arises from the fact that one cannot generally predict all of the effects of an action, because of lack of universal knowledge. Pulling the trigger of a gun, to use a now famous example, does not necessarily result in a loud noise. Even though it *usually* does, the gun's firing pin may have been removed, the bullets may have been removed, the gun's trigger mechanism may have been super-glued in such a way that pulling the trigger does not cause the gun to fire, et cetera. In order to avoid this kind of uncertainty, the operation of the world is restricted by the rules just described. [Shoham 88] presents a superb exposition on the frame problem and its causes and implications.

The effect of these restrictions is that the planner can model *time* as a single sequence of actions. Otherwise, the planner can completely ignore the passage of time in the real-world sense. Even in the case of Tate's NONLIN, the times required for actions are *predetermined* and related to specific actions.

In the example described in Figure 1.6, the sub-goal "NEAR CEILING" could have been alternatively accomplished by a "CLIMB TABLE" action. If the planner had chosen this mechanism, rather than climbing the ladder, then the order in which the ladder and ceiling are painted is no longer of consequence. More importantly, in a multiple actor domain, with more than one robot, the two activities could be carried out in parallel (i.e., at the *same time*).

For the most part, the planners created to date assume that actions taken as part of a plan take zero time to execute. This means that the multiple effects analysis described by Sacerdoti need only look at the resulting assertions and denials of any actions to determine if they interact in some destructive way. Even those systems, such as Tate's, which do include time use explicit specification rather than letting the system determine it.

A system which relaxes the restriction of instantaneous action requires a more complex representation for actions. Even though the *results* of two actions may not interact destructively, the interactions of activities *while* they are being done may create a problem.

So far, all domain independent planners have required unrealistic restrictions to the temporal complexity of the application domains. These restrictions have prevented their use in solving operations management problems. The handling of *time* in planning systems is critical if they are to schedule activities in which the planner must determine not only the sequence of activities and their starting times, but the length of time that an activity will be maintained or to which it will be constrained.

1.4 Requirements

Strategic operations management of complex systems and processes requires capabilities beyond those found in the planning systems described in the previous section. Such management systems must be able to reason and plan with information which changes

with the passage of time, and with actions that take varying amounts of time to execute depending on the situations in which they are performed. Management systems are required to make plans which involve multiple parallel activities carried out in a specific order and at specified times. Any useful management system must construct its own goals and then create plans to accomplish them. Lastly, an advisor in an operations management domain must dynamically fit its style and level of advice to the level of the operator it assists.

Most of the planning systems previously described have made the single actor assumption. While this is perhaps a necessity in light of the frame problem, the domains addressed by the current research all involve multiple actors. Events often happen in these domains in parallel. A runway may be plowed while its approach light system is repaired. More than one engine may be torn down and inspected on an aircraft at the same time by multiple crews. In each of these examples, there are multiple actors operating in the domain at the same time. Plans made in such a complex domain may fail due to lack of knowledge (i.e., the frame problem), but that should not discourage us from trying. Instead, the system must be designed to recover from such faults.

Planning systems of the past have not been required to deal with time. The single actor assumption allows the planning system to model time as a sequence of events, without regard to the size of the time interval between the events. Thus, in a robotic control domain [Sussman 75, Sacerdoti 77], a period of time might be modeled by a sequence of completed actions:

Move to the location of block B.

Clear the top of block B.

Move to the location of block A.

Pick up block A.

Move to the location of block B.

Put block A on top of block B.

Notice that there is no mention of how long it takes to do any of these activities. The fact that it might take much longer to move from the location of block B to that of block A than it does to pick up block A is of no consequence at all. However, in a multiple actor world, where parallel activities are common and desired, the times required for each activity become *very* important because some activities must never be executed at the same time (like plowing a runway and landing aircraft on the same runway at the same time!); while it is desirable to schedule other activities concurrently (such as clearing snow from a runway and repairing that runway's approach light system). In the first case, the actions must be ordered to avoid the conflict (as in [Sacerdoti 77]). The second case requires a completely different kind of reasoning. In the domains addressed by this research, the execution times of activities cannot be ignored.

In those planning systems that have dealt with time, such as [Tate 84], actions are described as taking fixed periods of time. In the domains considered here, actions may take different amounts of time depending on the particular situation during execution. For example, the time necessary to remove snow from a runway also depends on whether the snow accumulation will allow the runway to be brushed or requires plowing. Brushing is much faster than plowing. But in either case, the time necessary to clear the runway depends on the number of vehicles used and the length of the runway. The time also depends on how "heavy" the snow is: wet snow takes longer to clear than fluffy, "dry" snow. The same activity requires different intervals of execution under different conditions.

The necessity to reason about parallel activities of varying duration requires an extended logic (see [Kahn 77, McDermott 82, Allen 83a, Shoham 88]). Traditional propositional logic and first order predicate logic, which form the basis for reasoning in most expert systems and planners, have no representation of time. Extending the logic into the time domain demands a more powerful representation of information, and different methods to store and retrieve that information.

Complex expert systems and planners are virtually required to generate explanations of the reasoning behind the decisions concluded by the system. This powerful feature aids in debugging the rules and may act as a training mechanism in applications. The frame problem adds a new level of complexity to the generation of explanations.

The main ramification of the frame problem is that plans may not succeed because the future does not always unfold as predicted. If a prediction changes before plan execution begins, simple replanning is all that is necessary. Once begun, plans take time to execute. If new information indicates that a prediction was incorrect *after* the initiation of plan execution, then activities already completed (or in progress, but which cannot be undone) must be included in the replanning process. This leads to the possibility that an action in progress or completed may no longer be justified by the new situation. Explanation of the reasoning leading to such an action must refer to information no longer thought to be valid, but which was valid at the time the decision to execute the action was made.

For example, at noon the weather prediction during a winter snow storm indicates that at 2:30 p.m. the wind will require the use of Runway 33. The planner recommends plowing Runway 33 at 1:45 p.m. in order to have it cleared and ready for use at 2:30 p.m. Plowing commences as scheduled. At 2:00 p.m., a weather update shows that the storm is not moving as predicted, and the new forecasted wind will require the continued used of the current runway. When asked why plowing Runway 33 was recommended, the planner should reply that at the time the decision was made, it was justified by the predicted wind. Creating explanations of this kind requires retaining historical information.

Nearly all of the traditional planning systems described earlier produce plans based on *externally provided goals*.[2] In an operations management system, the overall goal is to keep the system operating in a stable and efficient manner. The system must be able to analyze the predicted future and create its own goals, and then produce plans to

[2]Some exceptions are Sacerdoti's NOAH [Sacerdoti 77], which was part of a larger system called the "Computer Based Consultant" [Hart 75], and Tate's NONLIN [Tate 84]. However, these systems did not do temporal projection to determine the goals for the planner. Instead, they were *reactive*.

accomplish those goals. As the future unfolds, it must update its plans as necessary, and deal with the unexpected.

The particular domain of runway configuration management has an additional degree of freedom because each airport has a different geometry in a different location surrounded by a different environment. Thus, each airport requires a unique set of rules in addition to the standard rules of operation. The system must be easily customizable for different airports.

Finally, it must be stressed that this runway configuation scheduling system is designed to be an *advisor* to one or more human supervisors in an operations management arena. The supervisors have the legal responsibility and authority and are not bound to any computer generated plan. This fact leads to a further requirement because of the possibility of different management styles of the supervisors charged with the task. In most cases, the same supervisor is not on duty at all times. Instead, an supervisor serves an eight hour shift and then is replaced by another supervisor. The controllers and operators available to execute management plans are likewise replaced in shifts, and vary in their training levels and capabilities. The diversity of style and level of expertise displayed by the different supervisors of the system, as well as the operating crews they command, requires that as the supervisors change, so must the rules and plans employed by the advisor system. For example, the strategic supervisor of a highly experienced controller crew might choose a more complex runway configuration than if the controllers were less practiced. Each supervisor using the system may wish to modify the advisory system to his or her particular style during the period he or she is responsible for management of the process. Such a change must disrupt the overall process being managed as little as possible.

As an advisor, the system is required to report its findings and suggestions to the supervisor in a timely and useful manner. The desired level and character of such reporting may change from supervisor to supervisor, or even for a single supervisor in varying cir-

cumstances. As with any process monitoring and analysis system, some information is of crucial importance while much of the detailed information used in formulating plans will be of little interest to the supervisor, except during unusual conditions. Certain kinds of important information should always be reported. However, reporting of mundane details should be suppressed unless specifically requested by the supervisor. The system must have a facility for specifying what types of optional information are to be reported.

These requirements were not well defined when this research was started. Rather, they were discovered as development proceeded.

1.4.1 Development History

When the present system was conceived, none of the technology for dealing with temporal systems existed. Much of the present system has been produced in parallel with work done by other researchers. The principal research about planning that has occurred during this development may be found in [Allen 83b], [Ambros 83], [Chapman 87a], [Charniak 85], [Dean 90], [Dean 89a], [Dean 88], [Dean 87a], [Doyle 86], [Drummond 88], [Firby 87], [Georgeff 87a], [Hanks 90], [Kautz 86], [Pelevin 87], [Schoppers 87], [Shoham 88], [Tate 84], [Vere 83], and [Wellman 90].

Several different attempts to create a time based planning system were made during the earlier stages of the research. Initially, a much more traditional, dynamic programming method was employed. This implementation, written in LISP, was able to find single point solutions for runway choices; but it lacked any time sense, had no explanation facility, was difficult to debug, and even more difficult to change. In addition, it didn't really *plan*. Instead, given the conditions at some moment, it reacted. This is a particularly bad strategy for the runway configuration management problem, since a given configuration generally interacts with both of the configurations preceding and following it. This kind of interaction is typical in many other operations management domains.

A second attempt was made with a purely back-chaining system written in PROLOG. In this system the rules were easier to debug, and it had a primitive explanation system, but it still had no time sense. In addition, building and using it pointed out that certain aspects of the cognitive process of strategic management are characteristically forward-chaining. Those parts of the PROLOG program were horribly complicated and very difficult to debug and maintain. Further, the program ran too slowly to be of any use. The PROLOG effort was abandoned.

The experience gained from these endeavors led to the conclusion that a successful system would have to consist of three interacting sub-systems.

First, the representation of *facts*, for which traditional planners and expert systems have employed property lists or associative triples, would have to be expanded into temporal histories describing the evolution of the property values over periods of time. This idea had been discussed before, notably by James Allen [Allen 83a] and Drew McDermott [McDermott 82]. Allen chose to represent time strictly as intervals, while McDermott created his intervals from more primitive temporal points. [Shoham 88] discusses both of these approaches in detail. None of these presentations discuss a detailed internal representation of histories or a precise mechanism to efficiently maintain such a database. An efficient representation of histories of facts had to be designed. A temporal database manager had to be constructed to maintain the resulting structure as beliefs changed due to the arrival of new information. The upkeep of this database had to include a temporally restricted form of truth maintenance.

Second, a rule-based forward-chaining expert system to produce and analyze projections of the domain had to be synthesized. This system had to be able to reason about facts with temporal extent provided to it by the temporal database manager, and in turn provide conclusions with temporal extent back to the temporal database manager. A logic dealing with propositions of varying periods of veracity, yielding conclusions with default "life times", had to be built into the system. The normal content and syntax of

assertions for this system needed to be extended to handle inferrences with "life times" which defied the norm in a number of possible ways.

Third, a planner, using an extended form of Sacerdoti's [Sacerdoti 77] network representation, had to be produced. The representation of actions in that work needed to be modified in order to support actions which take time to execute. Further, a scheme to compute the time necessary to accomplish an action under differing circumstances had to be devised and implemented. Goals for the planner would be supplied from the projection and analysis system in terms of patterns to be satisfied (if possible) by specified times. Plans generated would then be provided back to the projection system in the form of a set of assertions with specified time tags. These assertions would represent actions to be executed.

Chapter Two discusses an extension to the logic of time to include the ability to reason about self evolving processes and offer better explanation ability in a temporal environment. Chapters Three, Four, and Five present each of the sub-systems described above (see Figure 1.11). Chapter Six discusses an application of the resulting system to the Runway Configuration Management Problem, specifically with respect to snow removal. Chapter Seven concludes the thesis.

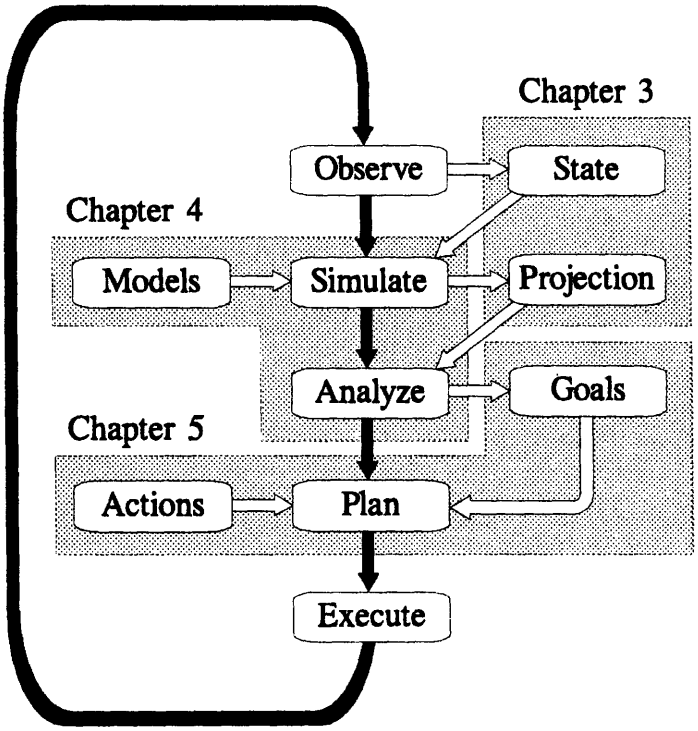


Figure 1.11: Scope of Chapters

$$\oslash p(\tau_1)_{(t_1, t_2)}$$

Chapter 2

Temporal Logic

Managing (i.e., planning and scheduling) runway configurations is a purely cognitive process. The *Tower Chief* effort was initiated to attempt to acquire and automate that cognitive process in the form of a rule based "expert system."

A rule based "expert system" is a computer program which employs some form of "reasoning" using a set of "rules" to infer new information about some domain of knowledge. The creators of such programs generally attempt to make the program closely mimic the reasoning process of a human.

Human reasoners engage in diverse and powerful forms of cognition. The formal mathematical representation of reasoning is called logic and includes induction and deduction. Induction is the usual process of learning. The most celebrated and best understood process of thought is deduction, the method employed in mathematical proof and in many other forms of argument. Humans can also make plans and reason about events and causality in an environment which frequently changes and often provides incomplete or inaccurate information. Humans need not be certain about their facts, and can make assumptions when required information is unavailable. They may use intuition based on prior experiences. These kinds of cognition go beyond the usual scope of formal logic.

Classical Logic normally applies to domains which are not dependent on time. The restriction to exclude time simplifies the mathematical model and makes it much more comprehensible. The restriction also limits the range of problems to which logic may be successfully applied.

Clearly, human reasoners often contemplate the future and create rational plans about it. During the last century, logicians have begun to extend Classical Logic in order to better model this kind of cognition. *Modal logics* have been invented to describe systems in which propositions "may" be true or false. *Defeasible logics* and *model theory* are used for systems which involve change. And *temporal logics* have been devised to reason about time. The purpose of this chapter is to describe the temporal logic used in the reasoning scheme employed in the operations management system.

2.1 Propositions

The elementary operands of logic are propositions. A proposition is a declarative statement such as, "Socrates was a man", or "There is ice on Runway 33." Logic consists of:

1. A set of abstract operations for combining propositions in ways that preserve the veracity of the resulting compound proposition;

2. A mechanism for generating new propositions from (i) knowledge already known and (ii) general statements called *rules*. This process is called *deduction*.

Symbolically, a proposition which is believed to be true may be represented as p. The belief that the same proposition is not true (false) is represented as $\sim p$. The opposite of a proposition being false is that the proposition is true, or $\sim (\sim p) \equiv p$. A situation in which p and $\sim p$ are claimed to be true is a *contradiction*. In Classical Logic, contradictions indicate an error in the logical system in which they occur.[1]

2.1.1 Compound Propositions

Propositions may be combined via "conjunction" (i.e., logical AND) or "disjunction" (i.e., logical OR). The symbolic representation of the conjunction of two propositions is:

$$p \wedge q$$

[1]Generally caused by an erroneous *rule*. Read on.

and that of the disjunction[2] is:

$$p \vee q$$

2.2 Rules

Deductive inference is accomplished through the use of *rules.*

> "A *Rule* is a hypothetical proposition composed of an antecedent and consequent by means of a conditional connective or one expressing reason which signifies that if they, viz. the antecedent and consequent are formed simultaneously, it is impossible that the antecedent be true and the consequent false."[3]

Translated into more modern terms, a rule is a conditional statement, consisting of an antecedent (the set of conditions to be met) and a consequent (the set of inferrences to be implied if the conditions required in the antecedent are satisfied).

While the laws of logic are *domain independent*, rules are based on semantic information. Thus,

"If it is raining, then the runways are wet." (2.1)

is a rule. Such a rule is written formally as

$$p \supset q \tag{2.2}$$

where p represents the antecedent ("it is raining") and q represents the consequent ("the runways are wet"). The hypothetical nature of the statement is embodied in the symbol $\supset$.

Generally, rules have more than one antecedent and may have more than one consequent. Rules may combine antecedents purely by conjunction, purely by disjunction, or

[2]The use of the symbol $\vee$ for disjunction is from the Latin word *vel*, meaning "inclusive OR". Unlike English, Latin possesses a separate word for "exclusive OR", *aut.*

[3]This definition is from a translation of the fourteenth century logician Pseudo-Scotus (John of Cornubia) which appears in [Bocheński 70].

in combination. As an alternative, a disjunctive rule can be split into several simple or conjunctive rules. While this is less efficient from a notational point of view, it avoids disjunction altogether. This is valuable when creating a computer program to do logic, because the program need only perform conjunction.

There are six classical forms in which deductive rules of inference may appear. For the purposes of this discussion, the description of two will suffice.

Modus Ponens This form, properly known as modus ponendo ponens, or the method of affirmation leading to affirmation[4], comes to the conclusion of the consequent if the hypothetical antecedent is declared to be true. For the rule above, the information that it is raining leads to the inference that the runways are, indeed, wet. Formally

$$\begin{aligned} & p \supset q \\ & p \\ & \quad \therefore \ q \end{aligned} \tag{2.3}$$

Modus Tollens The correct name is modus tollendo tollens, or the method of denial leading to denial.[5] In this form, the negation of the antecedent is concluded if the hypothetical consequent is declared to be **not** true. The formal definition is

$$\begin{aligned} & p \supset q \\ & \sim q \\ & \quad \therefore \ \sim p \end{aligned} \tag{2.4}$$

A *rule* may be thought of as a generalization that can be applied to a domain of specific situations. In effect, rules are the logical analogs of algebraic equations. This generalization is accomplished, as in algebra, through the use of *variables.*

[4]From the Latin *ponere*, "to affirm".
[5]From the Latin *tollere*, "to deny".

Note that the rule as stated in (2.1) is not really complete. Obviously, if the rain is falling in Boise, there is little effect on the runways at LaGuardia. The rule as stated is not precise enough. A more precise statement of the intended meaning of rule (2.1) is

"If it is raining *at an airport*, then the runways *at that airport* are wet." (2.5)

If the airport is represented by the variable x, that it is raining at x by Rx, and that the runways are wet at x by Wx, then a precise formal statement of rule (2.5) corresponding to (2.2) is

$$Rx \supset Wx \tag{2.6}$$

When a variable is given a specific temporary value, it is said to be *bound* to the value. A variable may have only one binding at a time; that is, no variable may have different values for two appearances of it in the same rule during any single evaluation of the rule.

2.2.1 Quantification

Rules may refer to *sets*. For example, if S is a set and $x \in S$ means x is a member of the set S, then one can make rules of the form:

$$\forall x, x \in S \wedge \mathrm{P}x \supset x \in S' \tag{2.7}$$

which states[6] that the set S' is made up of all the members of the set S which satisfy the requirement that $\mathrm{P}x$ is true. For example, the set of available runways at an airport is all the runways at the airport that are not closed. The form $\forall x, x \in S$ implies that the rule must be iterated over the members of the set S as x is bound to each member. Such iteration is called *enumeration* of the set, and the variable x is said to be *universally quantified*.

[6]The actual statement of equation 2.7 is, "For all x such that x is a member of the set S and satisfies the *predicate* P, x is a member of the set S'." A *predicate* is a test proposition which is either passed (TRUE) or failed (FALSE).

It is often the case that the requirements for solution to a problem are satisfied by more than one member of a set. If there is a rule specifying a method to compare the members of the satisfactory subset, a preferred member may be determined. Such a "*preference rule*" must be "seeded" with an initial choice for comparison, but *any* member of the satisfactory subset will do since enumeration is exhaustive. If the comparison choice is bound to the variable p then:

$$\forall x, x \in S \wedge x \sqsupset p \supset p \leftarrow x \tag{2.8}$$

states that, "For all x which are members of S, if x is *preferred* over p, then the preferred member p should be x rather than what it was prior to this evaluation." That is, when enumeration of the set S is completed, p will be the most preferred member of the set.

If there is no mechanism to decide a preference in the set, but a single member is sought for the solution to a problem, then any member will suffice. A rule to generate an individual member y of a set S satisfying some predicate P is:

$$\exists x, x \in S \wedge \mathrm{P}x \supset y \leftarrow x \tag{2.9}$$

or, "If there exists an x such that x is a member of S and x satisfies the predicate P, then that x is an *acceptable* candidate for y under P." This form enumerates the set S *until* an x satisfying the requirements is encountered, and then stops. The variable x in such a form is said to be *existentially quantified*.

Further information on Classical Logic can be found in [Copi 72].

2.3 Default Logic

In chapter 4 of Aristotle's Metaphysics [Bocheński 70], there is a section regarding the application of logic to predicting the future. In that treatise, Aristotle describes what has now become known as the axiom of Excluded Middle. In essence, he states that in his logic, a proposition can be either TRUE or FALSE, exclusively. No other values for the veracity of a proposition are possible. There is no middle ground. Because the

future is indeterminate, one cannot assign absolute knowledge of forthcoming events and therefore logic cannot be applied to reasoning about them. This is a broad statement which has far reaching implications. As pointed out by Bertrand Russell in the early nineteen hundreds, it is not a trifling thing to toss aside easily.

On the other hand, perhaps the interpretation of the axiom of excluded middle has been too broad. Aristotle was seeking truth in a very absolute way. He was quite aware of the works of Zeno and Pythagoras, and to a certain extent his logic was developed from his knowledge of their process of mathematical proof.

Classical Logic treats knowledge in a very restricted way. There is a tacit assumption that the reasoner knows all that is necessary in order to proceed. While this may be true in mathematical proof, it is certainly *not* true in general. Human reasoners are often faced with *incomplete knowledge.*

The standard interpretation of FALSE as NOT TRUE leads to an easily demonstrated ambiguity caused by incomplete knowledge. Suppose that p represents the statement, "It is raining." One interpretation of $\sim p$ is the statement, "It is *not* raining." Another interpretation is, "No information is available as to whether it is raining or not." That is, the reasoner's knowledge about precipitation is incomplete. In *either* case, it cannot be said that p is TRUE. For the purpose of mathematical proof, either meaning of NOT is sufficient to ensure that statement p cannot be made.

The domain of human reasoning is not as confined as that of mathematical logic. The difference between **knowing** something is FALSE or TRUE and **not knowing** may be crucial. The meaning of $\sim$ must be defined more precisely, and a separate, new symbol must be used to indicate incomplete knowledge if we are to capture more of the essence of human reason.

There is implicit to the statement that some proposition is true, the further statement that it is *known* to be true. When it is stated that, "It is raining", the actual *meaning* is that, "It is *certain or guaranteed* that it is raining". Similarly, when some proposition is

reported false, the meaning should be that it is *certainly* NOT true.

Let us look more closely at the meaning of negation. There are statements such as, "Day is *not* night", in which the negation is a property of the domain. That is, in the case that the meaning of negation is the opposite state of the proposition, the veracity of the logical connection is semantically derived from a rule.

Alternatively, denial of possession of knowledge concerning the truth of a proposition has nothing to do with the domain. It is a purely *logical* matter, having only to do with form. Having or not having knowledge about a proposition has nothing to do with the semantic content of the proposition.

The importance of this distinction may be exhibited by reference to a rule used in the Modus Tollens progression (refer to (2.4) above). If $\sim q$ means that q is *known* to be false, then the result of the progression is that p is *known* to be false as well. If, instead, $\sim q$ indicates that no knowledge is available about whether q is true (or false), then the result is that p is unknown, too. While the results look formally the same, the meaning is very different. In the case that the state of a proposition is known, we may not logically make any assumption concerning it. But when we have no direct knowledge about a proposition, it is often useful (or indeed, necessary) to make *assumptions.* For example, suppose that it is *known* that the temperature outdoors is 40°F, the dew point is just one degree less, and the humidity high. While it cannot be stated with certainty that there is fog, there is evidence to support the *assumption* that there is fog. Of course, if, in addition to the above, there is specific information that no fog is observable, an assumption about fog should not be made, but in the absence of such information the logic of planning may procede on the basis of the assumption.

This ability to make assumptions is the utility of supporting an "excluded middle". As a prerequisite, the two ambiguous meanings of **not** must be formally distinguished. For the remainder of this discussion, "$\sim$" will be used to denote that the semantic opposite of the proposition following it is true, while "$\oslash$ " will signify the lack of verifiable or

trustworthy knowledge of the proposition which follows it.

A few rules concerning assumptions are necessary:

- Observation must always take precedence over assumption. This means that if an assumption has been inferred about some proposition, and contradictive information is subsequently observed directly concerning the same proposition, the assumption must be replaced by the observation.

- All propositions which are concluded from rules in which one or more of the antecedent propositions are assumptions, are themselves assumptions.

A new class of rules called *default* rules may be defined. An default rule is different from a normal rule because one (and only one) of the propositions in the antecedent claims lack of knowledge about something. The consequent of a default rule is, by definition, an assumption. In the case of the "fog" example above, let $\mathrm{F}x$ indicate the proposition that there is fog at airport x. Let $\mathrm{Q}x$ indicate that the dew point is within two degrees of the outside air temperature at airport x. The default rule may be written

$$(\circledcirc\ \mathrm{F}x) \wedge \mathrm{Q}x \supset \mathrm{F}x^* \tag{2.10}$$

to mean, "If it is not known that there is fog at an airport, and it is known that the dew point is close to the air temperature at that airport, it can be assumed that there is fog at that airport". The superscript asterisk appended to the conclusion is a reminder that this proposition is an assumption.

This particular choice of mechanism for making assumptions is a special case of Reiter's "Default Logic"[Reiter 80].

The existence of assumptions extends the meaning of contradiction. To appreciate this, it is necessary to understand the different classes of information which can be present in a defeasible logic[7] with default rules:

[7] A *defeasible* logic allows the denial of assumptions which cause contradictions.

Observations are propositions that are obtained from outside of the logical system, i.e., given facts.

Inferred Facts are the consequent propositions resulting from rules in which all antecedent propositions are either observations or inferred facts.

Fundamental Assumptions are the consequent propositions arising directly from default rules.

Inferred Assumptions are consequent propositions which derive from rules in which one or more antecedent propositions are assumptions (of either variety).

The meaning of contradiction depends on the classes of the propositions involved in the inconsistency:

- Observation vs. Observation: One of the observations must be in error. Given enough domain information (in the form of rules), it may be possible to ascertain which proposition to believe, but this is generally difficult even for humans.

- Observation vs. Inferred Fact: This situation almost always indicates an erroneous rule. Some rule in the deductive chain leading to the inferred fact must be responsible; with luck, there might only be one.

- Inferred Fact vs. Inferred Fact: A contradiction of this kind is also indicative of an error in a rule, but the logical system cannot determine which rule is mistaken.

- Observation *or* Inferred Fact vs. Fundamental Assumption: The default rule that asserted the fundamental assumption does not apply to the situation. This might be due to insufficient specificity in the antecedent, but sometimes is unavoidable.

- Observation *or* Inferred Fact vs. Inferred Assumption: This is an interesting case. If only one of the propositions in the antecedent of the rule which inferred the assumption is an assumption, then the inferred assumption must be denied, and

the assumption which appeared in the antecedent must be denied, and so on back to the fundamental assumption which started that chain of reasoning. This process is called "dependency directed backtracking". If it should happen that there were more than one assumption in the antecedent of any of the rules in the chain that led to the discrepancy, then a choice must be made: Which assumption should be retracted? There are various approaches that might be taken to answer this question:

  - One could retract the chronologically latest assumption and search for an alternative.
  - One could withdraw the chronologically earliest and search for an alternative.
  - Alternative assumptions could be found for each of the candidates, trying each one until a choice is found which does not cause the contradiction. This may be very time consuming; or, in fact, undecidable.

- Fundamental Assumption vs. Fundamental Assumption: One or both of the default rules does not apply to the present situation.

In essence, all of these possibilities reduce to two major situations.

1. Contradictions among facts, which indicate rule errors of some kind.
2. Contradictions among assumptions, which require the replacement of one of the propositions with another assumption. Assuming the rules are correct, the difficulty lies in deciding which assumption to replace, and with what.

The essence of the solution to the problem of replacing an assumption involved in a contradiction is the employment of a class of domain dependent preference rules. These rules have a general form of "If there is a contradiction involving two assumptions regarding X, then Y is the preferred assumption to retain (or to retract)."

The idea of preference rules may also be used to decide which of two (or more) conflicting observations to keep. If each observation is tagged with a description of its origin, then preference rules stating that one origin is more "believable" or more "important" may be used.

2.4 Defeasible Logics and Truth Maintenance

The main difficulty encountered when attempting to create a logic involving time is *change*. The kernal of the problem is that if a proposition which was previously believed to be true is later thought to be false, then all propositions which were inferred using the changed proposition must be re-evaluated and either verified or denied. Logics which support change in this way are said to be "nonmonotonic" or "defeasible". In a classic paper in 1979, Jon Doyle [Doyle 79] made a first attempt to create a computer program employing an extended Classical Logic for use with dynamic domains. Doyle's *Truth Maintenance* is based on the idea that reasoned inferences are supported by evidence. The evidence supporting an inference is composed of the propositions that were used as the antecedents of the rule that resulted in the inference. If one or more of the evidenciary propositions changes, then the inferred consequent must be examined to verify that it is still true. If an inference is no longer supported by any evidence, then the inference must be denied.

More formally, Doyle, in creating Truth Maintenance Systems, restricted the application domain to those systems in which "all the propositions which satisfy modus ponens *also* satisfy modus tollens", i.e.

$$\begin{aligned} & p \supset q \\ & \sim p \\ & \therefore \ \sim q \end{aligned} \tag{2.11}$$

Propositions which meet this criterion are said to be logically equivalent. An example of logical equivalence is the state of a switch and the state of the voltage on a line controlled by the switch. When the switch is on, the voltage on the line is on; and while the switch is off, the voltage is off. While there are many examples of natural phenomena which exhibit this kind of behavior, there are many more which do not.

In Doyle's version of Truth Maintenance, evidence is kept in a "support list". In essence, a support list associated with an inference contains those propositions and a reference to the rule that was used to conclude it. Consider two rules which conclude the same proposition

$$R_1 : p \wedge q \supset c \tag{2.12}$$

and

$$R_2 : a \wedge b \supset c \tag{2.13}$$

These two rules are equivalent to a disjunction which embodies both rules

$$R_3 : (p \wedge q) \vee (a \wedge b) \supset c \tag{2.14}$$

If c is declared true, then its support list will contain one or the other (or both) of the conjunctions in the antecedent. If, at some future time, one of the propositions in the support list changes to false, the conjunction it appears within is removed from the support list. If the support list is empty, then by (2.11) c must be denied in Doyle's system.

This scheme works well for the restricted set of domains which satisfy (2.11). Unfortunately, most real-world processes **do not** fall into this set, and Truth Maintenance in this form cannot be used successfully to reason about their dynamics.[8] Further, there is no explicit mention of time in the Truth Maintenance mechanism, so the length of time that a proposition might be true cannot be easily specified.

[8] In Classical Logic, the form described in (2.11) is considered to be an error, and is known as "denying the premise".

2.5 Temporally Dependent Propositions

Propositions in mathematics are *universal* in their temporal extent. One never hears a geometer state that two lines are parallel from two until four this afternoon. The lines are simply parallel or they are not. The geometer's proof makes no reference to time, and, in like manner, neither does Aristotle's logic. On the other hand, human reasoning certainly encompasses planning and scheduling future events.

Time dependence can be formally introduced [Shoham 88, page 41] into logic by defining:

$$p\,(\tau)$$

to represent that proposition p is true during the time interval τ. The interval τ is a pair of numbers (such as Universal Times) such that the first member of the pair precedes or is equal to (i.e., is *before* or *at the same time of*) the second. Formally,

$$\tau = (t_1, t_2),\ \ t_1 \preceq t_2$$

The sequence of propostions

$$p_1\,(\tau_1), p_2\,(\tau_2), p_3\,(\tau_3), \ldots p_n\,(\tau_n),$$

describing the evolution of some aspect of the world is called the *history* of p. The collection of all of the histories known at any given time is called the reasoner's *time map*. The time map describes the reasoner's beliefs about the dynamics of the domain.

Having introduced this notion of the interval of veracity or the *activity interval* of a proposition, its effect upon all of the axioms of logic introduced in the previous section must be explored. It will suffice to examine only Conjunction, Disjunction, and the activity interval of the consequent of a rule.

As suggested in [Charniak 85] and others, in *Classical Temporal Logic* the activity interval of a conjunct will be defined as the *intersection* of the activity intervals of the operands:

$$p\,(\tau_1) \wedge q\,(\tau_2) = p \wedge q\,(\tau_1 \cap \tau_2) \tag{2.15}$$

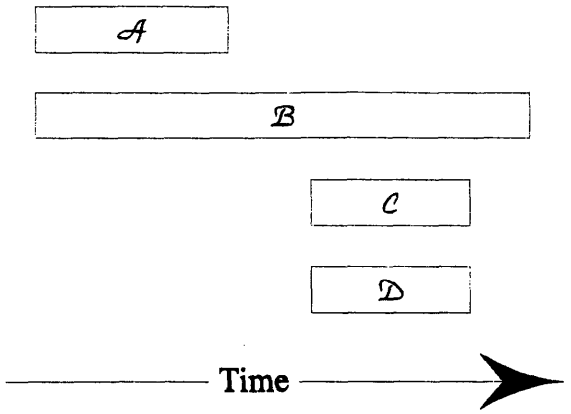


Figure 2.1: Perceived Deferred Causality

That this is a reasonable definition can be seen in the following example: If I am in room A during the time interval from two until four this afternoon ($p\ (\tau_1)$), and you are in room A during the interval from three until five this afternoon ($q\ (\tau_2)$), then *we* are both in room A from three until four this afternoon ($p \wedge q\ (\tau_1 \cap \tau_2)$).

Nota Bene: This definition of conjunction effectively states that two propositions can interact if and only if they have overlapping time intervals. This may seem overly restrictive at first glance, especially considering the human penchant for describing many interacting activities as following one another and being causally linked. However, closer examination reveals that the restriction is completely correct. Temporally disjoint activities which *seem* to interact are invariably connected by some "persistent" activity, produced as an effect of the first (causing) activity, which remains in effect at least until its time interval overlaps that of the second (caused) activity (see figure 2.1). Activity A causes activity B which persists after A has ceased to be true. Activity B and some other activity C cause activity D. Activity (C) is a *triggering* activity, and could become true due to some change in the environment or simply due to the passage of a specified period of time. The perceived causing activity (A in the figure) only indirectly causes the perceived second activity (D) through the agency of an unperceived (or possibly ignored) persistent activity (B) and some other activity (C). More will be said about this topic in

section 2.5.1 below.

The preceding motivates the definition of the activity interval of a disjunctive pair as the *union* of the activity intervals of the operands:

$$p\,(\tau_1) \vee q\,(\tau_2) = p \vee q\,(\tau_1 \cup \tau_2)$$

Again, using the room occupancy example: If I am in room A during the time interval from two until four this afternoon ($p\,(\tau_1)$), and you are in room A during the interval from three until five this afternoon ($q\,(\tau_2)$), then one or the other of us is in room A from two until five this afternoon ($p \vee q\,(\tau_1 \cup \tau_2)$).

2.5.1 Evanescence and Persistence

The rules described in the previous section for generating the activity interval of a logical combination of temporally constrained propositions allow the computation of the activity interval of the antecedent of a rule. However, in the real world, the activity interval of the consequent of a rule is not necessarily the same as that of its antecedent. Processes and physical things whose activity intervals are shorter than the activity intervals of their antecedents are called *evanescent*. An example of an evanescent process is the firing of a gun. When the hammer drops the gun fires. The fact that the hammer remains down does not make the "bang" last longer. Other things and processes are *persistent*, lasting well after the events which created them have ceased to exist. For example, if the temperature is below freezing on the ground, and it is raining, ice will form on the ground. When the rain stops, the ice does not simply disappear. Thus, the determination of the activity interval of the antecedent of a rule does not provide enough information *by itself* to determine the activity interval of the consequent propositions of the rule. This is a problem that must be overcome by a system which must model and "understand" real world events and processes.

The root of this problem is the domain dependence of the activity interval of a physical process. While it is true that no fact can exist without some form of causal precedent, once

formed a fact may have an independent existence of its own. In the case of persistence, quite often the only way to "undo" something which has been "done" is to do something else specifically designed to destroy it.

For practical purposes, there are only two classes of consequent

- Inferred propositions with activity intervals which are the same as the computed activity intervals of their antecedents. In this case, no further information is required in the consequent.

- Inferred propositions with independent activity intervals. Causality requires that the beginning of the consequent activity interval be the same as the start of the computed antecedent interval, but the domain dependent information to compute the end of the consequent activity interval must be supplied in a rule.

In the next chapter, this idea of different temporal types of consequents is further developed.

2.5.2 The Effects of Time on Defeasibility

Classical temporal logic is insufficient to describe the domain and events which occur in the ATC environment characterized previously. In particular, the information available about the future state of some value may change. In this sense, *any* proposition in a temporally dynamic domain is defeasible.

For example, there may be a weather prediction at 09:00 that claims that passage of a front, with an associated shift in wind direction will occur between noon and one o'clock. A later prognostication, perhaps at 11:00, might change the time of the frontal passage or some parameter associated with it. Such a change may require a modification of a planned configuration shift which may already be in progress. Because of the infeasibility of certain transitions, the modification of one configuration choice may affect those which precede and follow it, and so on.

Preparation for the use of a specific configuration may demand the allocation of resources in advance. In winter, for example, one or more of the runways that are to be used in a future configuration may require snow or ice removal or treatment to prevent ice accumulation prior to being put into service.

In standard expositions on temporal logic [McDermott 82, Shoham 88], the processes that the system is designed to model usually involve the evolution of some physical quantity such as the position of a ball or the temperature of an object. The rules for this kind of modelling generally look like

$$p(t_1, t_2) \wedge q(t_3, t_4) \supset r(t_5, t_6)$$

where $(t_5, t_6) = [(t_1, t_2) \cap (t_3, t_4)]$ because of Equation 2.15. If r is a new state that was described previously by p, then $t_2 \preceq t_5$[9], $\sim p(t_5, t_6)$, and propositions p and r describe a "self evolving process" in which a later state of the process depends on some earlier state. For example, "If some water is in the liquid state (p) during (t_1, t_2), and is brought in contact with a thermally massive object with a temperature greater than the boiling point of water (q) during (t_3, t_4), then the water will be in the vapor phase (r) during (t_5, t_6)." (Of course, this isn't quite right because there is a period, no matter how short, during which the water is in contact with the thermal source and heating, but not yet boiled away. For simplicity, this period is being ignored.) Clearly, if $p(t_1, t_2)$ is later found to have been a mistaken belief and withdrawn, its dependent $r(t_5, t_6)$ must be withdrawn as well. This is straight forward truth maintenance as described in Section 2.4.

Consider instead the reasoning involved in the allocation of some resource. Let $\mathcal{W}r(t_1, t_2)$ indicate that use of resource r is desired during the time interval (t_1, t_2). Similarly, let $\mathcal{A}r(t_1, t_2)$ indicate that resource r is available during the time interval, and finally let $\mathcal{H}r(t_1, t_2)$ represent that the resource has been allocated for the period. Then we might write

$$\mathcal{W}r(t_1, t_2) \wedge \mathcal{A}r(t_1, t_2) \supset \mathcal{H}r(t_1, t_2) \wedge \sim \mathcal{A}r(t_1, t_2)$$

[9] In other words, a thing cannot be in two disjoint states at the same time.

to describe the rule for allocation: "If resource r is required during (t_1, t_2), and the resource is available during the period, then r is allocated for the interval, and is no longer available for allocation during that time." There is an apparent paradox in this formulation, since $\mathcal{A}r(t_1, t_2)$ and $\sim \mathcal{A}r(t_1, t_2)$ appear at the same time. The reason that this paradox appears is that the reasoner's *belief* about the availability of the resource for other use during the specified interval changed as a consequence of the reasoning process itself. This is an example of a *planned change.*

This example demonstrates that reasoning about the future may involve a different kind of non-monotonic logic. Propositions previously believed to be true cannot necessarily be retracted. They may be in the support lists of later propositions which involve actions or the reasoning process itself. A human reasoner in the resource allocation example would explain, "Of course, I believed the resource was available *before* I allocated its use. Now, however, I have 'committed' to a plan of using it during the time period, and it is no longer available for other use during that period." The reasoner is aware of the temporal order of the events of the reasoning process in addition to the projected time of execution of the plan.

There are *two* kinds of time periods associated with reasoning about the future. The first kind is the period in the future during which the activity is to occur (i.e., the interval during which the use of the resource is desired, in this case). The second kind is the time period during which the reasoner believes the *apriori* future scenario before its decision to change that future (i.e., the time interval that it *believes* that the resource will be available during the activity period).

Another example of the second kind of time interval is the time during which the planner believes the *aposteriori* future prospect after the decision to change the future (i.e., the time interval during which the planner believes that it has allocated the resource for the activity period and the resource will no longer be available for other use during that interval). Time intervals associated with the reasoning process, rather than the

scheduled execution time, are called "belief intervals". If the belief intervals are put into the logical statement of the resource allocation rule as subscripts to the propositions, we obtain

$$\mathcal{W}r_{(\tau_1,\tau_2)}(t_1,t_2) \wedge \mathcal{A}r_{(\tau_3,\tau_4)}(t_1,t_2) \supset \mathcal{H}r_{(\tau_5,\tau_6)}(t_1,t_2) \wedge \sim \mathcal{A}r_{(\tau_5,\tau_6)}(t_1,t_2)$$

and the paradox is resolved. The reasoner can now refer to what it believed prior to making its decision as well as its opinion after the act of making the decision.

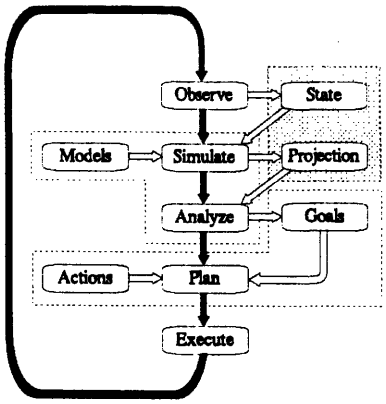
A plan is by definition intended for future execution. Plans are based on what the planner *believes* is going to happen. Because the future is not fixed, a plan may have to be modified or even abandoned before or during its execution. Actions which have been taken cannot be withdrawn when a plan is abandoned. That such actions were based on beliefs which turned out to be false does not change the fact that they were executed. If the planner is to be able to explain the reasons behind its actions, it must recall its prior beliefs even if they were later proven to be wrong. In fact, the planner will change what it believes about the future as a result of its intention to carry out some plan. After all, the causal precedent of a plan is that the reasoner wants to *change the future*!

Since ATC supervisor's duties include allocation of men and equipment to a variety of tasks, and a manager must take *actions* based on the current knowledge about the domain, the kind of reasoning described above is a requirement for accomplishing the cognitive task of planning runway configurations.

2.6 Summary and Preview

In this chapter, a theoretical basis for a reasoning system capable of inference in a temporally dynamic domain based on extensions of logic was presented. While a theoretical foundation is absolutely necessary, it may be useless without a careful implementation. As is usual, there is little indication of what may be a good implementation in the theory. Perhaps the most critical part of the implementation of a reasoning system is the structure used for storing data representing facts. This becomes even more important in

the system being described here: An efficient mechanism must be used to represent the evolution of information in time.



Chapter 3

The Time Map Manager

Any reasoning process requires knowledge. Traditionally, the information required by an expert system is stored in a "knowledge base", a database of structures, representing propositions, describing the facts known about the domain. In many cases, each piece of information is stored in an *associative triple*, a three part object consisting of a *thing*, an *attribute name*, and a *value*. For example, "(ball color red)" is an associative triple. A *thing* can be anything, such as a ball, a runway, or an airport. An *attribute name* (henceforth, simply, an *attribute*) is the name of some property of the *thing*, such as the "color" of the ball, the "length" of the runway, or the "wind direction" at the airport. A *value* is the actual value of the attribute of the thing, as in "red", or "10005 feet", or "270 degrees". Facts, represented by associative triples, are the operational propositions of the reasoning system.

Like the traditional logic it emulates, this model is static, having no mechanism for the description of time. Of course, even simple real world objects *do* have attributes which change in time. For example, the color of a stop light is not constant. Sometimes it is red, sometimes yellow, and at other times green. As described in the previous chapter, the static nature of the values appearing in traditional associative triples must be replaced with histories describing the temporal evolution of the values taken on by the attribute.

The sequence of values which make up the history of some thing's attribute has been referred to as a "time map" [Charniak 85].

Changing the value of a traditional associative triple is a simple matter: the value is replaced. Changing a value in a time map is not so simple. First, the temporal position of the change in the time map must be determined. Second, the inclusion of new information may change the temporal extent of previous information in the time map. It is necessary to have a special facility to maintain and access information in time maps: a *time map manager*.

3.1 A Structure for History

The introduction of the activity interval (section 2.5) allows facts of the form, "Logan's ceiling is 1200 from 18:00 until midnight." The term "is" in this statement may be replaced with any simple declarative tense of the verb "to be", such as "was" or "will be". A time ordered set of statements like:

> Logan's ceiling will be 3500 from 07:00 until 10:00.
>
> Logan's ceiling will be 3000 from 10:00 until 11:00.
>
> Logan's ceiling will be 2500 from 11:00 until 11:30.
>
> Logan's ceiling will be 2000 from 11:30 until 12:00.
>
> Logan's ceiling will be 1200 from 12:00 until 13:00.
>
> Logan's ceiling will be 800 from 13:00 until 13:30.
>
> Logan's ceiling will be 600 from 13:30 until 15:00.
>
> Logan's ceiling will be 2500 from midnight until 06:00 tomorrow.

is called a *history* of Logan's ceiling. Note that it is perfectly reasonable to have gaps in a history during which knowledge about the value of the attribute is not known.

It would be much more efficient to write the history of Logan's ceiling as follows:

Logan's ceiling will be 3500 from 07:00 until 10:00.

3000 from 10:00 until 11:00.

2500 from 11:00 until 11:30.

2000 from 11:30 until 12:00.

1200 from 12:00 until 13:00.

800 from 13:00 until 13:30.

600 from 13:30 until 15:00.

2500 from midnight until 06:00 tomorrow.

Each entry in this structure is called a *history cell*, and consists of a value and an associated activity interval during which the value remains constant. This extended fact may now be defined as a *thing*, an *attribute*, and a *history* made up of a time ordered set of history cells describing the temporal evolution of the attribute's value.

Although there is added complexity due to implementation requirements of the program, the internal structure of propositions in the program is essentially based on this structure. Most importantly, all new information presented to the system is supplied as history cells; that is, as a *thing*, the name of an *attribute* of the thing, a *time interval*, and a *value* the attribute is believed to have during the interval. References to the value of a fact during an interval result in one or more history cells describing the history of the attribute during the specified period.

Of course, this picture of the future may change as further information about it arrives. This indefinite property relegates such "facts" to the realm of "beliefs". Operationally, this has little effect since, whenever a plan is made for some future action it is based upon beliefs about what the future will bring.

Charniak and McDermott [Charniak 85, page 422] suggest that the inference engine of a system like this one be called a *projection manager*. As they also suggest, the current program consists of two major modules (see Figure 3.1): a *temporal-system analyzer*, a system for creating, manipulating, and using rule systems and rules called "*rulesys*" (this is where the logic is), and a *time-map manager* or temporal database manager named

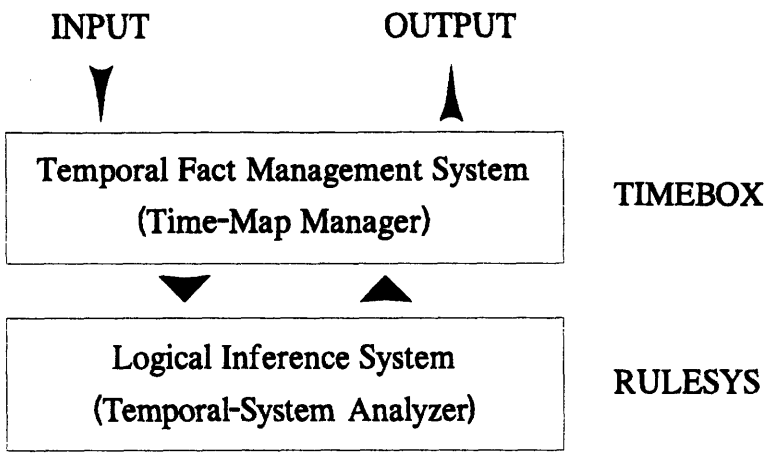


Figure 3.1: Projection Manager

"*timebox*". These two modules are not independent; there are functions in each which refer to structures defined and manipulated by the other. But to the extent possible, they are isolated systems with separate purposes.

3.2 TIMEBOX

Virtually all of the structures and manipulating functions concerning the creation and maintenance of facts are contained in the *TIMEBOX* module. *TIMEBOX* refers to functions and structures defined in two other modules, *timan* and *timetags*. The *timan* module contains all the definitions and functions necessary to use time intervals, while in the *timetags* module, time instants and the lowest level time functions are defined.

In order to make the presentation as clear as possible, the architecture will be presented in a "bottom-up" fashion, starting with the simplest and most basic structures and building more complex objects from them.

Values are *timeless*. That is, values by themselves are not bounded or constrained in time. Values are further devoid of any meaning other than their value. Values have no pre-defined structure.

Each value has at least one *implicator* (see section 2.4). An implicator is either

a rule and the information that was used to imply the value, or the special implicator "USER-INPUT", indicating that the information was supplied as observation. In a complex domain having more than one possible observer, the name of the observer may be used as an implicator of information provided by that observer.

An *event* as used here means something which happens at a single time instant. This may be either the beginning or termination of some time interval, or the time of occurrence of something which may be taken to happen over a time interval so short as to be viewed as instantaneous. The time at which an event happens is described by a *time point*. For the present implementation, a time point is represented by an extended UTC integer. "UTC" is "Universal Coordinated Time", the method used by astronomers to define the time of an observation. For a given time, it is defined as the number of seconds since midnight, January first, 1900, at the Royal Observatory in Greenwich, England. That is, it is the elapsed time in seconds from the end of the last century. Because events in a computer program may happen very fast, a special "extended" UTC is used. This AUTC[1] is hardware dependent, defined in terms of the particular machine's basic clock tick. This is accomplished by defining a machine-dependent constant which is the number of clock ticks per second. The AUTC is the integer expressing the number of clock ticks since midnight, January first, 1900. All internal real times are expressed in this unit.

3.2.1 Temporal Relationships for Information

A number of useful functions have been included in the *timetags* module for operations concerning event times. In particular, there is a function, `print-tod`, which will print, in human-readable format, an AUTC. Another function, `cvt-day-time`, will convert a human readable expression of a time into an AUTC. The function `now`, which has no arguments, returns the AUTC corresponding to the instant when it is called. A whole list of functions for defining convenient times is provided. These include:

[1] For *Accurate* UTC. This is its internal name in the code.

from-now *hours &optional minutes seconds*: produces an AUTC offset in the future by the arguments.

before-now *hours &optional minutes seconds*: produces an AUTC offset in the past by the arguments.

last-midnight: returns the AUTC of the last midnight which occurred before the current time.

next-midnight: returns the AUTC of the next midnight which will occur after the current time.

noon-today: returns the AUTC of today at 12:00.

tomorrow-at-this-time: returns the AUTC twenty four hours in the future from the time when it is called.

tomorrow-at *time*: produces the AUTC corresponding to the time of day specified, but on the following day from today.

how-long-until *AUTC*: will return the time remaining until the specified AUTC.

A time of day acceptable to **cvt-day-time** may be stated as either a two digit integer indicating an hour of the day (e.g., 14 means 14:00), or as a four digit integer indicating the hour and minute (e.g., 1423 means 14:23). Note that a twenty four hour clock is used throughout.

The above functions, while very basic to the system, are also very useful to rule writers. Examples of their use in writing rules will be given later.

Time intervals are defined and manipulated by programs in the *timan* module. A time *interval* is specified by a starting time and an ending time, both of which may be AUTC's. However, either or both of these times may be specified by the special symbol **FOREVER**. A start time of **FOREVER** indicates a time forever in the past, while an end time

of **FOREVER** signifies a time forever in the future. The interval starting at **FOREVER** and ending at **FOREVER** is the special interval **ALWAYS** which contains all other intervals by definition.

The function **print-time-interval** is provided to print time intervals in a convenient format.

The lowest level user function for making activity intervals is

create-interval *start &optional end*: The start and end times may be a time acceptable to **cvt-day-time**, or any of the event time producing functions mentioned above. In addition, the start time may be specified as **'until**, which will result in a time interval starting forever in the past and ending at the specified end time. This is really syntatic sugar, since one could use **'forever** as well. Using **'until** without an end time is an error. Specifying only a start time will result in a time interval starting at that time and extending forever into the future. Finally, the special interval **'always** may be used by itself.

In addition to this basic function, a number of more advanced functions are provided to make the user specification of activity intervals relatively easy. These comprise:

today: which returns the interval from **last-midnight** until **next-midnight**.

this-morning: the interval from **last-midnight** until **noon-today**.

this-afternoon: the interval from **noon-today** until 18:00 today.

tonight: the interval from 18:00 today until 06:00 tomorrow.

The job of the temporal database manager (*TIMEBOX*) is to keep the information about each fact's history up to date and in time order. This ordering is accomplished by comparing intervals and using an information updating scheme.

Allen [Allen 83a] has shown that, for arbitrary intervals, there are only thirteen possible relations that may result from a comparison of two intervals (See Figure 3.2). The

inclusion of open intervals indicating forever described above does not change this, though it does make the comparison program somewhat more complicated. Open intervals are necessary so that the system can express unbounded intervals such as **ALWAYS** and open ended persistence.

All information presented to the system is required to be accompanied by an activity interval. The arrival of some new information describing the value of some attribute of some thing during an activity interval may be *new*. That is, there may not have been any previous value or values for that particular thing's attribute during the specified time interval. In that case, updating the history is easy: just insert the new history cell into the proper place in the fact's history. However, it is also possible that there were previously believed values during the period defined by the interval. In this case, the new value is assumed to be *replacement* information unless specific instructions are received indicating that it is *additional* to what was already known. In either case, the final result is an updated fact, and the new information depicting the changed state of the domain is represented by the updated fact and the activity interval.

3.2.2 The Replacement of Old Information with New

The programs which access and maintain the temporal knowledge base are in the *TIMEBOX* module. Data which is replaced may not simply be over-written and forgotten. This requirement emerges from the property of reality that actions taken due to a belief generally may not be retracted if the actor later finds out that the belief was incorrect. It might be argued that, while the *action* may not be retractable, the reasons for the action certainly are. But that view would make explanation of the reasoning behind the action impossible. If the reasoning system is required to give explanations, then asking it why it did something and getting the answer, "I don't remember." is not acceptable. For example, if the reasoner believes that a forthcoming change in wind direction will require changing to a runway that is currently unusable due to being covered

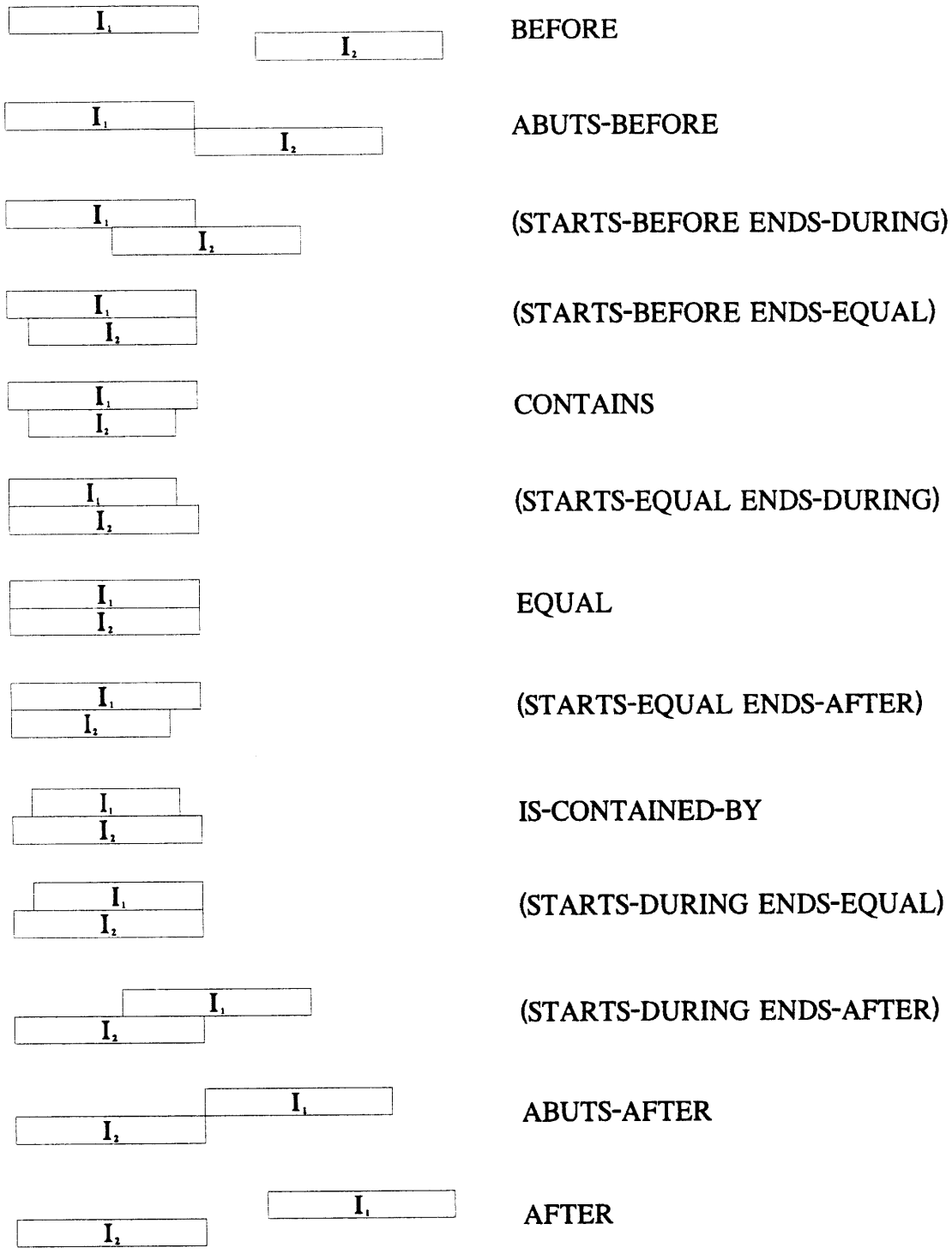


Figure 3.2: Temporal Relations

with snow, it should order snow removal equipment to clear the runway prior to the time of the required change. If it later turns out that the storm stalls and the wind change is delayed, a different plan will have to be produced. If the reasoner is then asked why the specified runway has been plowed, it should be able to answer that it *thought* (note the past tense) that the wind was going to change at the time the decision to plow the runway was made.

Changed information may have been used to infer other information. If the change has an activity interval in the future, it is usually the case that the inferences based upon it should be withdrawn. This process is the equivalent of "truth maintenance" (See [Doyle 79]), but the introduction of time brings a new twist to the process. If the start time of an inference is after "now", it might no longer be deniable. For example, if a situation has inferred that a runway should be plowed beginning at some specified time, and later information indicates that the wind will not change as quickly as was previously expected, a new inference may choose to move the time at which plowing should commence. If the original time interval of the plowing activity is "close" to "now", then the reasoner must ask if the plowing has begun. If it has not, then perhaps the commencement time can be moved. If it has started (or has been committed and cannot be stopped), then a whole different chain of reasoning must be considered.

The requirement of remembering what was believed in the past complicates the structure of facts described above. Instead of each fact having a single history, it must have a "history stack". The top most member of this stack is the current belief of what the history of the fact looks like. The next member is the most recent past belief of what the history looked like, and so on back into the past. It may seem at first glance that this would demand vast amounts of memory, but judicious choice of the structure and updating algorithm greatly reduces the storage required.

If there is no known information about *thing's attribute* during the specified time interval, then the new information can simply be added to the current history. In all

other cases, truth maintenance must be attempted. Old values implied by the changed information must be denied, and their inferences denied exhaustively before a new value may be asserted over the period. The process is made more complicated because the time interval of the new information may incompletely cover the time interval of an existing history cell. In this case, implied value cells *during the covering interval* must be removed from the history cells containing them. This may require splitting a history cell into two history cells, one with and one without the old value.

One can visualize (Figure 3.3) a fact's history as tape stretching from the past into the future. On the tape are lines indicating the beginning and ending of each "history cell". A history cell is a structure indicating an interval of time during which the value (also specified in the cell) of a specific attribute of some thing remains constant. Within each history cell, the value or values valid during that period are written. Figure 3.3(a) shows a part of a history of sky conditions and precipitation over a period of twelve hours. From 07:00 until 11:00 the sky is partly cloudy (PC). From 11:00 until 14:00, it is expected to be overcast (OVC). From 14:00 until 17:00, the sky will be low overcast (LOVC), and between 17:00 and 19:00 forecasters are predicting thundershowers indicated by R.

In our analogy, a new replacement history cell can be made by cutting a piece of opaque tape of the correct length, writing the new value(s) on it, and then taping it in the proper place on top of the tape representing the history. This operation will cover or hide some of the old history tape. Clearly, if the starting time of the new history cell is before the ending time of the history cell which directly precedes it in the history, then that preceding history cell ends prior to when it was thought to do so before the introduction of the new history cell. In fact, it must now end when the new history cell begins. In similar fashion, if the ending time of the new history cell is after the starting time of the history cell which immediately follows it, then that following history cell's starting time must be changed to the ending time of the new history cell. Any cells which started after the starting time of the new history cell and ended before the ending

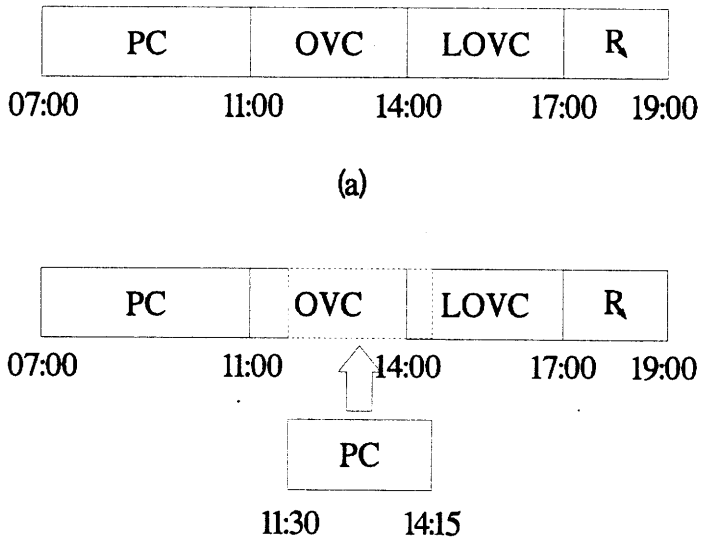


Figure 3.3: Replacement

time of the new history cell will be completely covered by the new cell. Figure 3.3(b) illustrates the situation when the forecast *changes* to partly cloudy between 11:30 and 14:15. The new "partly cloudy" interval begins during the "overcast" interval and ends during the "low overcast" interval. The paradigm described above indicates that the new information will be represented by a piece of tape lasting from 11:30 to 14:15, glued over the appropriate interval on a copy of the original tape. The old tape (i.e., the old history) is pushed one level deeper in the history stack, and the amended tape (Figure 3.3(c)) becomes the currently believed history.

Each history also has a belief interval (see section 2.5.2) associated with it. The start of the belief interval of the history at the top of the belief stack is the time that the last update was made, and the end of it is forever. The start of the previously believed history is the time when the update that created it was made, and the end is the same as the start of the next history later in the history stack. Figure 3.4 shows a schematic of the structure of a single fact.

At the top-most level, the fact consists of the name of a thing, the name of some attribute of the thing, and the history stack which describes the evolution of the reasoner's beliefs about the temporal development of thing's attribute. Each entry in the history stack is a history depicting the temporal variations of the attribute as it was believed to be over some belief interval. The history is made up of a list containing one or more history cells. The history cells are time ordered, and each one delineates a period of time (the "activity interval") during which the attribute was believed to be unchanging. The value or values describing the attribute are stored in a list of "value cells". Each value cell contains one or more values, a "sense" which may indicate the negative (i.e., **not**), a flag indicating whether the values are the result of a default rule, and dependency information for use by the truth maintenance system (not shown in the figure). There are "backward" references from the history cells to the history containing them, and from the histories to the fact to which they belong.

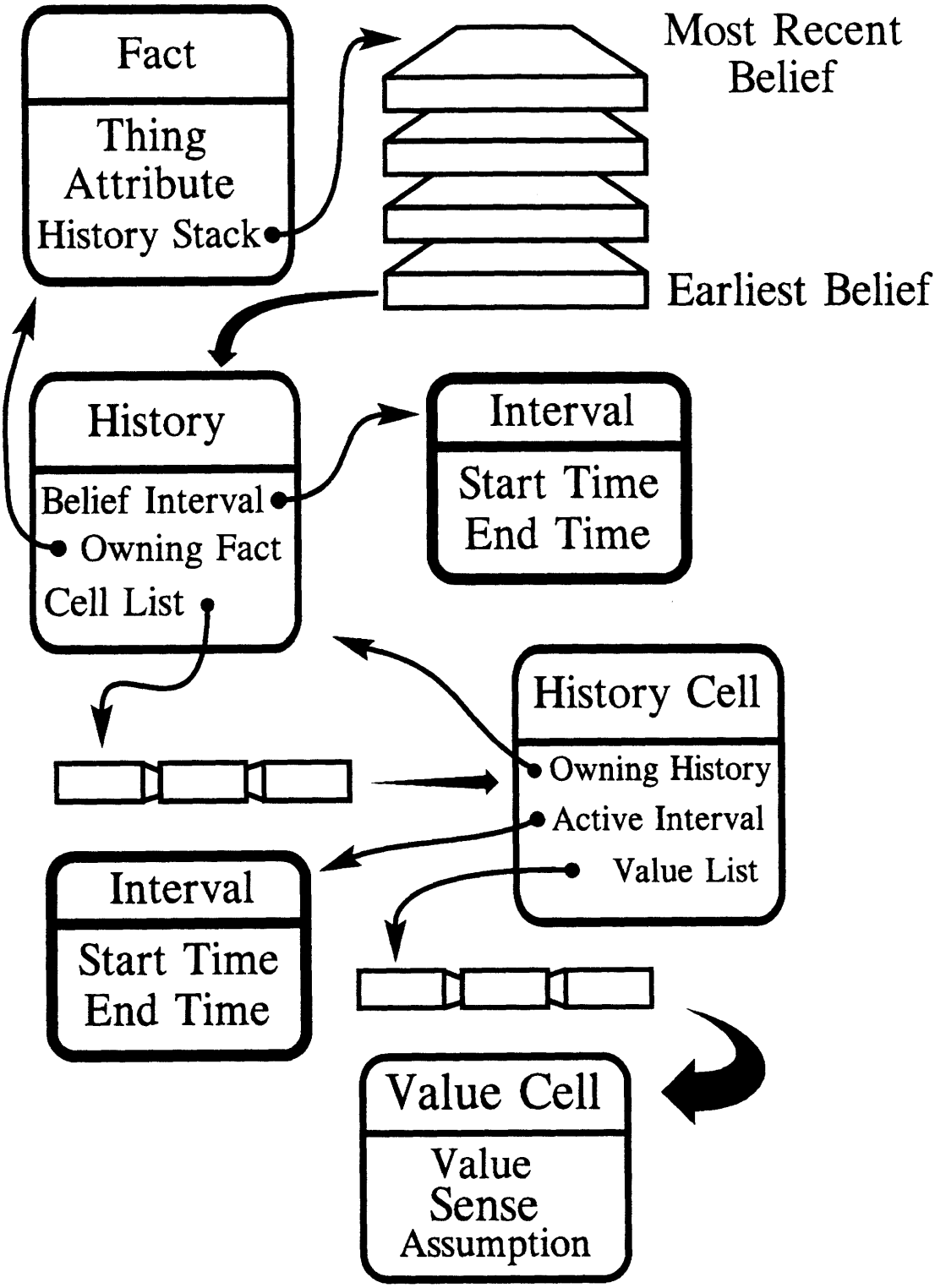


Figure 3.4: Internal Fact Structure

The reader is warned that the figure and explanation above are simplified for the sake of explanation. The actual structure is considerably more complex. The additional complexity is required in order to minimize the storage requirement for the structure and to accelerate update processing.

3.2.3 Addition of New Information

The value of an attribute of some thing may be a *set*. This is a most useful idea, in that it allows rules to be filters on sets which create sub-sets.[2] Any history cell containing more than one value cell may be interpreted as being a set. The membership of a set may, of course, be time dependent. For example, the set of usable runways at an airport changes with the weather, the time of day, and other parameters. The existence of sets requires the ability to add members to a set. The reasoner must be able to properly deal with statements of the form: "In addition to what you already know about the membership of this set during some interval, χ is also a member." This kind of update is called addition.

The updating process for additional information can be viewed in much the same way that was presented for replacement, but in the case of addition the tape for the new history cell must be transparent rather than opaque so that the previous values can still be seen (See Figure 3.5). Any partly overlapped existing history cells must be split into two history cells at the starting (or ending, as appropriate) time of the new history cell, and the new information added, as a new value cell, to each history cell covered by the new time interval. Of course, the result becomes a new history that is then pushed onto the top of the fact's history stack.

3.2.4 Functions for Updates

Both "replacement" and "addition" described in 3.2.2 and 3.2.3 are accomplished by making assertions to a rule system. There are two functions (tell and rs-assert) that

[2]More about this later.

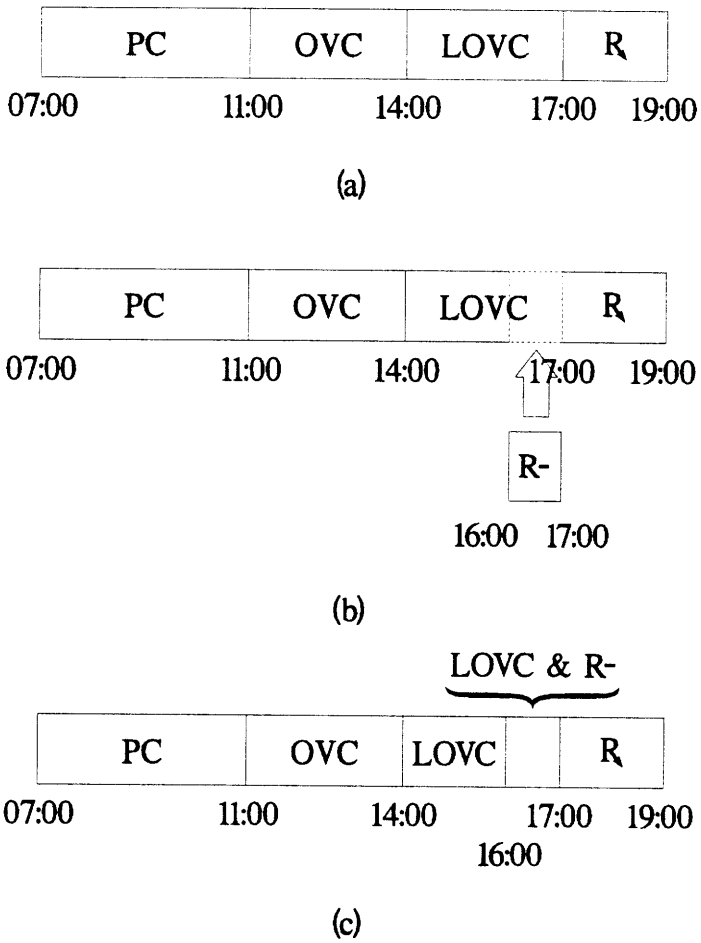


Figure 3.5: Addition

allow assertions to be made to the fact database, one for direct input by the program's user, the other from within the consequent of a rule.[3] All of the function definitions are in standard LISP notation.

User Updates

When the user wishes to assert some new replacement information about a fact to a rule system, the command is:

```
(tell <rule-system> <thing> <attribute> <time-interval>
<value>)
```

where

<rule-system> is the name of a rule system,

<thing> is the name of a thing,

<attribute> is the name of a state variable of **<thing>**,

<time-interval> is a time list or interval defining function,

<value> is a value.

If the information is in addition to that already known, then the statement must have the sequence "**:additional t**" added to it, as in:

```
(tell <rule-system> <thing> <attribute> <time-interval>
<value> :additional t)
```

The single symbol "-" may be used for the **<time-interval>** in a tell form to represent the interval **always**. This is particularly useful in describing initial conditions and physical relationships of a system. For example,

[3]See section 4.4.

```
(tell tc bos-4L runway-direction - 35)
```

states that the direction of Runway 4 Left at Boston's Logan airport is always 35 degrees.

The **tell** form is an attempt to simplify the user interface of the system. It will not accept variable names in its input, does not require the use of the LISP **quote** form, and requires the user to provide an activity time interval. The form used in rules is less restricted.

Rule Updates

Rules have the activity intervals of their consequents determined by the projection manager, so activity intervals are not included in assertions within rules.[4] The form used to assert some new information in the consequent of a rule is:

```
(rs-assert <thing> <attribute> <value> <rule-system>)
```

where

<thing> is either a variable determined in the antecedent of the rule or the literal name of a thing preceded by an apostrophe, as in **'logan**.

<attribute> is the name of an attribute preceded by an apostrophe, as in **'ceiling**.

<value> is either a variable determined in the antecedent, a literal non-numeric value preceded by an apostrophe, like **'obscured**, or any lisp function called with any or all of the variables determined in the antecedent (see section 4.3) as parameters.

In the consequent of a rule, the rule system will determine the appropriate activity interval, so a consequent assertion never mentions a time interval.

To make **rs-assert** add the new information rather than replace it, the sequence "**:additional t**" must be added to the end, just as with **tell**:

[4]Information about the *extent* of the time interval of a conclusion may be included. See section 3.2.6.

```
(rs-assert <thing> <attribute> <value> <rule-system>
:additional t)
```

The fact that some thing's state is *not* a specified value is just as important as a statement concerning what it is. For either **tell** or **rs-assert**, the addition of **:sense nil** at the end changes the assertion to the negative. For example, the form:

```
(tell tc logan precip ((this-evening)) rain)
```

means that it is expected that precipitation at Logan this evening will be in the form of rain. The negative of that statement, that it is expected that precipitation at Logan this evening will **not** be rain (of course, it might be sleet or snow) would be:

```
(tell tc logan precip ((this-evening)) rain :sense nil)
```

3.2.5 Denial of Information

Note that neither of the above is the same as, nor has the same effect of, a statement that some thing's state is *unknown* during an interval. Such a statement can be made by using the form:

```
(deny <thing> <attribute> <interval> <rule-system>)
```

which indicates that the value of *thing's attribute* is unknown during the specified interval. If a value is included in the **deny** form, as in:

```
(deny <thing> <attribute> <interval> <value> <rule-system>)
```

the action of the system depends on the nature of *thing's attribute* at the time of the denial. If the attribute is single valued (i.e., not a set), then for all sub-intervals of **<interval>** during which *thing's attribute* is the specified *value*, *thing's attribute* becomes unknown. If the attribute is a set, then the specified **<value>** is removed from the set during the **<interval>** if it is a member of the set. That is, during the specified **<interval>**, **<value>** will no longer be a member of the set. Denial may, like replacement,

trigger truth maintenance on information inferred using the denied history cell. The same restrictions also apply.

There are equivalent forms to **deny** for use in the consequent of a rule, where the time interval is inferred by the system. They are:

```
(rs-deny <thing> <attribute> <rule-system>)
```

and

```
(rs-deny <thing> <attribute> <value> <rule-system>)
```

The effects of these forms are identical to the corresponding user forms described above.

3.2.6 Temporal Extent of Inferred Information

Thus far in the discussion, only one temporal type of consequent has been mentioned. There are several different types of consequents, distinguished by the temporal form and extent of the assertions within them.

a) **Bounded** assertions are the default. The activity intervals for these kinds of consequents are defined as described formally in section 2.5 and less formally in [Charniak 85, page 421]. As the search to match the patterns in the antecedent is carried out, the activity intervals of the matching history cells are *intersected* to form a final "search interval". This interval can be no larger than the activity interval of the *causer* which triggered the rule. By default, the final search interval becomes the activity interval of all assertions made in the consequent of the rule (Figure 3.6).

b) **Persistent** assertions are very important, because they are the usual connective mechanism of causality. The laws of temporal logic developed in the previous chapter (and the implementation of them described here) only allow logical interaction between pieces of information with overlapping activity intervals. That limitation raises the question of how such a system can have causal relationships in which an action during one activity interval can cause another action during a later (and non-overlapping) activity interval.

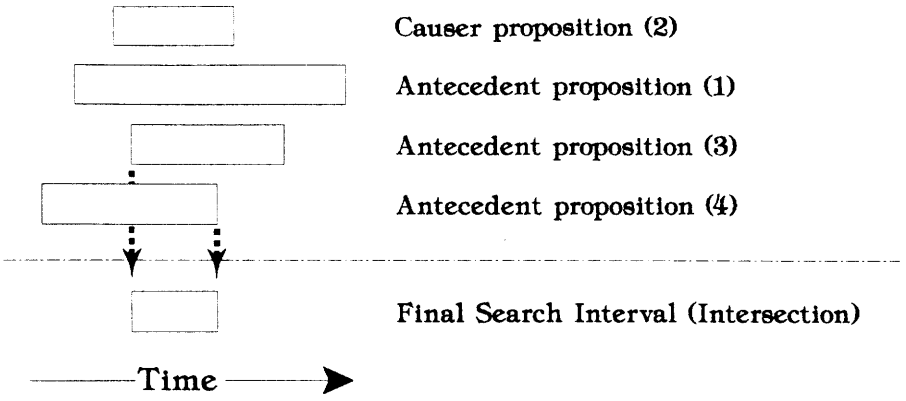


Figure 3.6: Determination of Intersection Interval

Close scrutiny of such relationships reveals that, invariably, the first action creates some information which "*persists*": that is, the activity interval of the new information endures after the termination of the activity interval of the final search interval determined in the antecedent. This persistent information interacts with a later piece of information to create the second action. The information with which the persistent information interacts may be some real action or may simply be the passage of a specified period of time.

The following example illustrates both of these possibilities. Suppose a rule system is desired which will predict the outcome of the following story:

> "It rained hard from 11:00 until noon. The precipitation quit completely by 12:30. At 12:45 the driver of a car traveling at high speed violently engaged the brakes."

The conclusion should be that the car skid out of control and crashed. In terms of the representation which might be used by a hypothetical "motoring-advisor" system, the following rules would predict the outcome of the story:[5]

```
(defrule wet-roads motoring-advisor
```

[5]Words appearing with a prefixed question mark (?) indicate variables.

```
((?road precip-state rain))
(rs-assert ?road 'surface-state 'wet motoring-advisor
:type 'persistent))
```

The second line of this rule is a search *pattern.* Patterns will be discussed in section 4.3. This rule states that, "If it rains on a road, the road gets wet and stays wet for sometime after the rain stops." The next rule:

```
(defrule skid motoring-advisor
((?auto traveling-on ?road)
(?road surface-state wet)
(?auto speed ?v)
(?auto brakes hard))
((> ?v 10))
(rs-assert ?auto 'stability-state 'skidding motoring-advisor))
```

states, "If a car is traveling on a wet road with speed greater than ten and the driver brakes hard, then the car begins to skid." Notice that skidding is not persistent. This is an implicit statement that the skid only lasts as long as the driver continues to brake hard. Of course, in reality this may not be true, but it is satisfactory for the present example.

Then there is a final rule:

```
(defrule crash motoring-advisor
((?auto stability-state skidding)
(?auto speed ?v))
((> ?v 30))
((rs-assert ?auto 'speed 0 motoring-advisor
:type 'persistent)
(rs-assert ?auto 'condition 'destroyed motoring-advisor
```

```
:type 'persistent)))
```

This final rule indicates, "If a car is skidding at a speed greater than thirty, then its speed becomes zero and the car is destroyed." Presumably, it hits something and crashes, though to keep the example short this has been omitted.

Notice that the skid is only indirectly caused by the rain. The rain causes the road to be wet, and the wet road persists after the rain stops.

The reader may detect that there is something not quite right about the last rule in this group. The problem is that, in fact, the auto probably does not crash the instant that the skid begins, but only if the skid persists *for more than a specified period of time*. In order to specify that the consequent of a rule is true only if the antecedent of the rule remains true for a specified interval, a special test form, called **persists**, may be included in the tests performed after the matching stage is complete. The syntax of **persists** is very simple:

(persists <time-offset>)

or

(persists <time-offset> <unit-name>)

where **<time-offset>** is a number and **<unit-name>** is the name of a time unit, such as **second, minute, hour, day**, or **year**. The default unit name is **minute**. A second version of this form is:

(persists-until <time-point>)

where **<time-point>** is any event time described in section **5.2.1** above. For example:

```
(persists-until (noon-today))
```

These two tests examine the final search interval determined in the matching process. In the first case, if the interval is as long or longer than the **<time-offset>**, then the conse-

quent is evaluated. In the second, if the final search interval *contains* the <time-point>, then the consequent is evaluated.

The effect of persistence requirements on the starting time of the consequent assertions is situation dependent. In the skidding example described above, the crash should not begin until *after* the skidding has persisted for some period. For example, suppose that the skidding rule should only indicate that the car will crash if it continues to skid for a period of five seconds. Then the activity periods of the assertions about the crash should not begin until five seconds of skid have occurred. This is indicated by including :ap in the assertions, indicating "after persistence".

Now the last rule about the possibly skidding car can be properly written:

```
(defrule crash motoring-advisor
((?auto stability-state skidding)
(?auto speed ?v))
((> ?v 30)
(persists 5 'seconds))
((rs-assert ?auto 'speed 0 motoring-advisor
:type 'persistent :ap)
(rs-assert ?auto 'condition 'destroyed motoring-advisor
:type 'persistent :ap)))
```

so that the crash only happens if the speed stays above thirty and the skidding persists for five seconds or more, and that the activity interval of the assertions start five seconds after the onset of the skid.

A planner may use the persistence of an activity in another way. For example, a runway configuration planner must have rules which limit the number of configuration changes which occur within a given interval. The reason for this restriction is that configuration changes do not happen instantaneously. Suppose, for example, the wind is forecast to change direction rapidly by almost one hundred eighty degrees, as when

frontal passage occurs close to a low pressure center. There will be a period, as the wind changes direction, when the wind may favor use of an intervening runway configuration between the original and final wind directions as the front passes. If the period during which the intervening runway configuration will be usable is not long enough, then the configuration should not be put into use at all. If the configuration will be usable over a long enough time, then a change to it should be made at the *beginning* of the usable period.

If the "`:ap`" is not included, then the activity interval of the consequent assertions is the final search interval determined in the antecedent, as described before. A rule to require a configuration to be "valid" for at least half an hour in order to be usable might be written:

```
(defrule declare-config tc
((?airport arrival-runway-1 ?arway)
(?airport departure-runway-1 ?depway)
(?airport arrival-demand ?demand)
(?config is-a configuration nt)
(?airport owns ?config nt)
(?config arrival-runway-1 ?arway nt)
(?config departure-runway-1 ?depway nt)
(?config arrival-capacity ?capacity nt))
((persists 30 'minutes)
(> ?capacity ?demand))
(rs-assert ?airport 'configuration ?config))
```

Translating this rule into English yields, "If new arrival and departure runways have been established for an airport, and the period over which these runways will be usable is at least thirty minutes, and there is a configuration that uses these two runways which has a capacity which exceeds the expected demand for the period, then assert that the

configuration should be used during the period."

c) **Decaying** assertions are related to persistent assertions. A decaying assertion is a persistent assertion which has an initial value, an associated decay function, and a decay limit. The **value** normally associated with any assertion becomes the initial value of the decay cell. The specification of the other two parameters is made in a list following the keyword **:decay-spec**. The list must contain the name of the decay function followed by the decay limit. The initial value must be a number or a variable determined in the antecedent which evaluates to a number. The function name must be the name of a previously defined LISP function or expression. The decay limit must also be a number or a variable that evaluates to a number. *Note*: Decay starts at the *end* of the final search interval determined in the match process of the antecedent, unless it is over-ridden by a specific persistence test. The exact syntax is:

```
(rs-assert <thing> <attribute> <value> <rule-system>
:type 'decay :decay-spec '(<decay-function> <decay-limit>)
```

The decay function must be a function of three numerical arguments; the first is the initial value of the decaying attribute, the second is the start time of the activity interval, and the third is the current time. Both of these times and the initial value will be supplied by the system during binding of a variable when the history cell is matched. Times will be provided as UTC's in *seconds*, in *real* (i.e., float) internal format. The decay function must return a *real* number. An example of a decay function is:

$$w(t) = w_0 e^{-t/1200} \tag{3.1}$$

The LISP function to compute this is:

```
(defun wet-road-evaporation (init-val start-time end-time)
(* init-val (exp (/ (- end-time start-time) 1200))))
```

Once the decay function has been defined, it may be used in an assertion. For example, the following rule restates the wet road condition described above in a more realistic way:

```
(defrule wet-roads motoring-advisor
((?road precip-state rain))
(rs-assert ?road 'surface-wetness 1.0 motoring-advisor
:type 'decay :decay-spec '(wet-road-evaporation 0.2)))
```

This rule declares that a road becomes wet when it rains, and stays wet for about thirty minutes after the rain stops. That is, the computed persistence of the activity interval of the wet road would extend about thirty minutes after the end of the activity interval of the rain.

Direct use of the *value* computed by the decay function at the time a pattern is being matched allows knowledge engineers to design rules which refer to quantitative comparisons in the test sections of rules. For example, the skid rule defined in the discussion on persistence above could be written:

```
(defrule skid motoring-advisor
((?auto traveling-on ?road)
(?road surface-wetness ?how-wet)
(?auto speed ?v)
(?auto brakes hard))
((> ?v 10)
(> ?how-wet 0.3))
(rs-assert ?auto 'stability-state 'skidding motoring-advisor))
```

to indicate what level of road "wetness" will cause a skid.

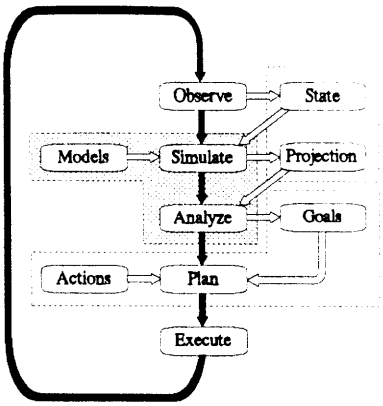
3.3 Summary

The temporal database manager described in this chapter is not a replacement for traditional database management systems used to organize information for improved performance. Rather, it is a requirement brought on by the addition of temporal variability

to information. A traditional database management system could be placed on top of the system described here.

A new, powerful and efficient structure has been introduced for the representation of interval based time dependent information. The structure allows the use of closed- and open-ended intervals and has the capability to efficiently keep old beliefs for explanation and causal reasoning. A system to access and maintain information stored in this representation was described.

The purpose of the time map manager sub-system is to create and maintain an internal representation of the temporal evolution of domain state. Information in rules is used to extend and analyze this temporal picture of the domain in order to more completely model its future and effect changes. The sub-system which does this rule-based modelling is the Temporal System Analyzer.



Chapter 4

The Temporal System Analyzer

The *temporal system analyzer* [Charniak 85] used in this scheduling system is called *RULESYS.* It is a rule-based forward-chaining expert system that uses information supplied by the time map manager to predict and analyze the future. All new information arriving in the system passes through the temporal system analyzer (*TSA*). Most of the rules are used in a modelling or simulation mode to predict future domain states. Other rules in the TSA have an analytical or decision making purpose. The conclusions of these analytical rules contain requests for the planner to devise a schedule of actions which will result in a change of a future state.

The programs in the core of this expert system refer to a data structure called a "rule system". A rule system is an object, in the sense of "object oriented programming." That is, there can be multiple rule systems extant at any time. Each "rule system" contains specific rules and information about some domain. That is, each "rule system" has a set of regular rules, a (possibly empty) set of default rules, and a set of facts concerning some domain of application. Facts are entered into a rule system by "asserting" them to it. The consequent of a rule normally asserts some newly derived facts to a rule system. That the recipient rule system may not necessarily be the same rule system which made the assertion implies communication *between* rule systems. This is a distributed "committee

of experts" which could be implemented on separate computers.

4.1 Composition of Rule System Objects

A *rule system* is a complex structured data object which contains the rules, factual information (in the form of histories), and state information describing current knowledge about some domain. The RULESYS programs operate on the rule system database in order to maintain and extend the knowledge it contains. The components of the rule system structure employed here are:

name The name of the rule system. This identifier is used to direct assertions to a specific rule system, as well as other house keeping commands.

trigger-list A directory of the rules defined in the rule system, organized by the names of the attributes which trigger their use.

default-trigger-list A directory of default rules, organized by the patterns which, if no known facts match them, cause these rules to be used.

rules The actual repository of rules belonging to the rule system.

default-rules The storehouse of default rules belonging to the rule system.

agenda The queue of situations needing attention by the rule system.

pattern-list The unique patterns used in the antecedents of the rules of the rule system.

facts The factual information known to the rule system about its domain.

reportables The topics about which the user has asked the rule system to report whenever they change.

rules-to-monitor A sub-set (normally empty) of the **rules** which are to be traced when the user is debugging a rule system.

actions A network of primitive actions and plans making up the knowledge base for the scheduler (see Chapter 5).

It should be noted that there are *three* separate databases contained in this structure: the **facts** database which is the temporal database describing domain states; the **rules** database which contains the analytical rules for the temporal system analyzer; and the **actions** database which includes the primitive actions and plan templates used by the scheduler. The description here is only an introduction to the structure; further detail will be provided later in this chapter and the next.

The command to create a rule system is

```
(def-rule-system <name>)
```

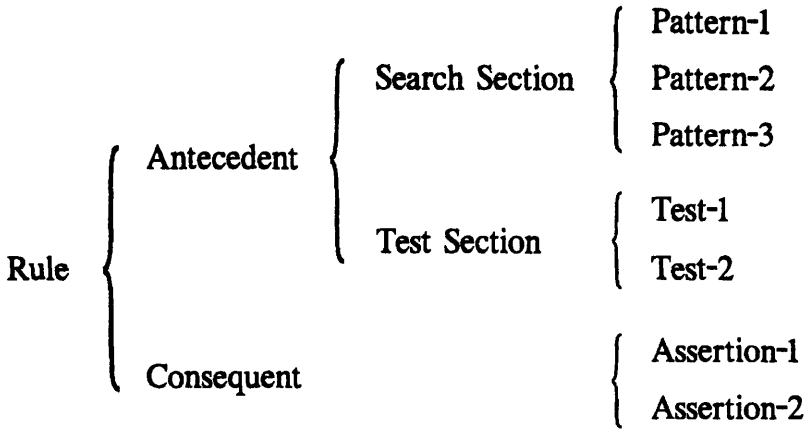
Where **<name>** is the name the user wants to give to the rule system. The name can be quite arbitrary,[1] though if more than one system is to be in existence at a time care must be taken not to name them identically.

4.2 Structure for Rules

The implementation of *rules* (section 2.2) requires three major activities of the TSA. First, it must *search* its knowledge-base for all facts pertinent to the antecedent of the rule. If the information is incomplete (i.e., if one or more of the propositions in the antecedent cannot be matched by known information), then use of the rule is rejected for the particular situation being explored. Second, any *tests* required by the antecedent must be carried out. Should any test fail, again the rule must be rejected. Third, if sufficient information was found and all tests passed, the rule must *assert* its conclusions.

[1] Actually, a rule system name should be restricted to the alphabetic characters and the character "-". The system ignores case.

Thus, the overall structure of a rule is:



4.3 Patterns with Variables

One of the central issues in any rule based reasoning system is, "Under what conditions should a specific rule be used?" There are two ways to answer this. First, a rule should be used when the consequent might assert something about the current state of the domain that the system (or user) desires to know. Second, a rule should be used when some new fact matching part of its antecedent becomes known by observation.

The first of these answers describes the paradigm used in "back chaining" systems such as PROLOG. The second describes the process used in "forward chaining" systems such as *Tower Chief*. There are also systems, such as ART[2] and KEE[3], which allow programmers to take advantage of both. *Tower Chief* uses back chaining to answer certain hypothetical queries for the user, but uses forward chaining to produce all of its inferences.

Both of the paradigms require "pattern matching" in order to choose rules to evaluate under appropriate conditions. In the *Tower Chief* system, patterns are represented by associative triples of the same sort already discussed. For example, the triple

```
(Logan ceiling 2200)
```

[2]ART is a trademark of Inference Corporation.
[3]KEE is a trademark of IntelliCorp.

is a pattern. A "match" to the specific pattern appearing above must be *identical* to it. If all patterns had to be identical to the information to which they matched, the resulting system would be all but useless because there would have to be a specific rule for each individual case of every situation in the domain of the system. Rules are supposed to be generalities that are applicable to many different situations in order to infer similar specific information about each situation. The real power in pattern matching is the use of *variables*. Variables are represented in this system by a word preceded with a question mark, such as `?airport`. When a pattern containing variables is matched against a fact, the variables are said to be *bound* to the corresponding element in the fact. The list of pairs of variables and corresponding bound elements is called the *binding list* or simply the *bindings* of the match. For example, the pattern

```
(?airport ceiling ?cloud-height)
```

would match

```
(Logan ceiling 2200)
```

or

```
(Worcester ceiling 800)
```

In the first case the bindings would be `((?airport Logan) (?cloud-height 2200))`, while in the second the bindings would be `((?airport Worcester) (?cloud-height 800))`. Once a variable is bound during the matching search of an antecedent, that binding takes effect for all instances of the variable appearing in the antecedent. That is, once a variable is bound, the binding cannot be changed by further search until the binding is removed.

Variables may be used in the assertions appearing in the consequent of a rule. The value of the corresponding binding will be substituted when the assertion is actually made. Thus if the consequent of a rule containing the pattern `(?airport ceiling ?cloud-height)` made an assertion:

```
(rs-assert ?airport 'operational-status 'IFR tc)
```

then in the first case above the assertion would state that `Logan` is operating in `IFR`, while in the second the airport to which the assertion referred would be `Worcester`.

4.4 RULESYS

As described in 2.2, rules have an antecedent (the "if" part) and a consequent (the "then" part). If the antecedent is not satisfied, then the consequent is not true. The meanings of "satisfying the antecedent" and "the consequent is true" are not very clear in the form given in the last chapter when applied to antecedents and consequents with variables. The purpose and content of both antecedent and consequent must be thoroughly defined if a computer program is to understand their meaning.

The overall purpose of a rule in an "expert system" is to find and combine known information in order to infer some new and previously unknown information. The "known information" is located in some form of database which the expert system can search. Part of the antecedent of a rule is a set of "patterns" which are used to direct the search for known information. Since the facts in this TSA are, at least to some degree, associative triples, the patterns are also associative triples. As described above, variables in the TSA are specified by words with a prefixed question mark,[4] as in `?x`. The pattern

```
(?airport ceiling ?x)
```

is said to match `(logan ceiling 1000)` if the variable `?airport` is "bound" to `logan` and the variable `?x` is bound to `1000`. The set of pairs consisting of a variable and that part of a fact to which it matches is called the *bindings* of the match. In the case above, the bindings are `((?airport logan) (?x 1000))`.

Any non-variable appearing in a pattern must be identical to the corresponding part of a fact if it successfully matches the fact. That is, if the *thing* part of a pattern is not

[4]More precisely, by any character sequence beginning with a question mark followed by a string of at least one character. The string must not contain any of the characters "'", ",", "", "#", ":", ";", or "@".

a variable, then knowledge about that thing must exist in the database. In the present form of RULESYS, the *attribute* part of a pattern may not ever be a variable. If the *value* part of a pattern is a non-variable, then that value must appear as one of the values of *thing's* attribute. If this is not the case, then the pattern does not match the current fact. For example, suppose the pattern

```
(?airport operational-status vfr)
```

were to be matched against information that

```
(Logan operational-status ifr3a)
```

In this case, the pattern does not match the fact.

The temporal aspects of facts requires additional complexity in the matching/binding process. Most of the time there is more than one pattern in the antecedent of a rule, and all the patterns much be matched conjunctively.

The evaluation of a rule is due to the arrival of new information which "triggers it." That is, the rule is evaluated because a pattern in its antecedent may match the new information. Recall that "information" in the sense of this system is embodied in a history cell, with an associated activity interval. This new history cell which triggers the evaluation of a rule is called the rule's *causer*. The causer's activity interval defines the widest temporal range that the conclusion of the rule can possibly effect. However, most rules have more than one pattern in their antecedent, and all the antecedents must be matched conjunctively if the consequent is to have any effect at all. In the previous chapter, the conjunct of two temporally constrained propositions was defined to be true only during the intersection of their activity periods. As each subsequent search is performed to find a match to a pattern in the antecedent, the search is constrained to the intersection of the activity intervals of all the history cells which have been previously matched. This interval is called the "search interval" of the evaluation, and can never become larger as each pattern is matched. It is certainly possible that the search will,

because of the existence of variables in the patterns, match more than one history cell of a fact during the constraining search interval. This will result in a list of matched history cells. In this case the evaluation of the rule separates into two or more subsequent evaluations, one for each of the matching history cells. The search interval of each succeeding search will be the intersection of the previous search interval and the activity interval of the particular history cell chosen from the list. For example, consider the set of patterns:

```
(?airport ceiling ?cl)
(?airport visibility ?vis)
...
```

and ceiling data given by:

```
(logan ceiling (11 1130) 2500)
(logan ceiling (1130 12) 2000)
(logan ceiling (12 13) 1200)
```

Suppose now that the user enters a fact:

```
(logan visibility (11 13) 2.5)
```

to indicate that Logan's visibility will be 2.5 miles from 11:00 to 13:00. The rule in which the set of patterns appears will be triggered, and the initial search interval will be from 11:00 until 13:00. When the **ceiling** pattern is matched over that period, there will be *three* matching history cells, with activity intervals from 11:00 until 11:30, from 11:30 until 12:00, and from 12:00 until 13:00. The resulting bindings for the search for matches to the next pattern would be

```
((?airport logan) (?ceiling 2500) (?visibility 2.5))
```

from 11:00 until 11:30

```
((?airport logan) (?ceiling 2000) (?visibility 2.5))
```

from 11:30 until 12:00

```
((?airport logan) (?ceiling 1200) (?visibility 2.5))
```

from 12:00 until 13:00

If there were no more patterns in the antecedent, then further evaluation of the rule would occur for each of the three bindings and time intervals.

If the TSA is unable to find facts which match all of the patterns in a rule's antecedent, then the context for that rule is incomplete. That is, the rule does not apply to the situation currently under consideration. If, on the other hand, RULESYS is able to find facts which produce matches (perhaps with bindings), then the rule bears further consideration. The pattern matching step in rule evaluation is essentially an existence test for the information which is necessary for the rule. If that information is incomplete, the rule does not apply. If it does, then further investigation may be necessary.

A rule may require certain relationships to exist about or between the variables which are bound in the pattern matching step. For example, a typical rule to determine if an airport is operating under Instrument Meteorological Conditions can be stated, "If the ceiling[5] at an airport is less than one thousand feet, then the airport is operating in Instrument Meteorological Conditions." In this case, not only must the ceiling at the airport be known, but it must be tested to determine if it meets the criterion stated. If it does not pass the test, then the consequent is not true. If it does, then the consequent is true. Of course, there may be several such tests to determine the propriety of the information found in the matching step.

So, perhaps a more profound statement of the purpose of the antecedent would be that it must produce a complete and meaningful context under which the consequent is true. In this regard, the antecedent must accomplish two tasks. First, it must find appropriate facts matching its patterns and assign values to all of the variables which appear within it. Second, it must verify that all relationships required about or between

[5]The *ceiling* at any place is the height above the ground of the first layer of cloud which obscures more than forty nine percent of the sky.

those assigned values are correct.

The new knowledge specified by the consequent if the antecedent is satisfied must be added to the system's store of information in its fact database. That is, the consequent of a rule consists of one or more assertions of new information. Now a very precise form of a rule can be written:

```
(defrule <rule-name> <owning-rule-system>
(<patterns-to-match>)
(<tests-to-perform>)
(<assertions>))
```

For example, the rule stated above, about Instrument Meteorological Conditions, can be written

```
(defrule ifr1 tower-chief
((?airport ceiling ?x))
((< ?x 1000))
(rs-assert ?airport 'operational-status 'imc tower-chief))
```

4.4.1 Creation of Rules

A considerable amount of pre-processing is done by the system when a rule is defined. First, the rule system structure is checked to see if a rule with the same name already exists. If it does, then the user is warned about the re-definition of the rule and the old rule is replaced by the new one. If no rule with the name of the new rule is found, a new rule structure is created and put into the rule system fact repository. The patterns of the antecedent are then scanned to create the "*triggering attribute list*" and the "*variable list*" of the rule. The triggering attribute list is the set of attributes (state variable names) used to determine when the rule should be considered for use. A pointer to the rule is entered in the rule system's trigger list for each attribute appearing in the triggering attribute list. The variable list of a rule is a list of all the variables that appear in the

rule. It is used to form the parameter lists of the test and the consequent sections of the rule (see below).

It is often the case that while all of the patterns in an antecedent are required to be matched for the rule to assert its conclusion, not all of the patterns should trigger the use of the rule. For example, a single rule might be used to determine and assert both the lateral and longitudinal wind components of a runway. A rule to determine whether a runway is feasible under the resulting conditions would have patterns in its antecedent to bind values for each of these components:

```
(?runway headwind ?rhw)
(?runway crosswind ?rcw)
```

If both of these patterns cause situations to be enqueued on the agenda, the rule will be evaluated *twice* for each determination of the wind. In order to avoid this inefficiency, a pattern may be tagged as *non-triggering* by appending the term "nt" to the end of the affected pattern. If the author of the rule above did not want the rule to be triggered by assertions of headwind, the patterns would be written:

```
(?runway headwind ?rhw nt)
(?runway crosswind ?rcw)
```

The rule in which this fragment appeared would still match and bind both patterns, but it would only be triggered by crosswind assertions.

Users should be cautioned to use this feature with care. Incorrectly preventing a rule from being triggered by even one attribute can have broad and complex repercussions on overall system correctness. Making all patterns triggering (i.e., never making use of "nt") will at worst degrade the run time efficiency of the system. Rule developers are urged to add the use of "nt" only after all other debugging is complete, and then with considerable prudence.

Each pattern in the antecedent of a rule is tested to see if it appears in the pattern list[6] of the rule system structure, and added if it does not. Thus each pattern only appears in the pattern list once. The actual patterns in the internal representation of the rule are pointers to the proper patterns in the TSA's pattern list. The set of tests in the antecedent are translated into a lisp function using the keyed variable list as the lambda list of the resulting function form. This function is stored in a slot of the rule structure called the "computation". A similar translation is done to the consequent, but with two special variables added to the variable list as primary and secondary positional variables. These two variables are the context and the activity time interval of the function call. Explanation of the structure and formation of the values of these variables when the rule is being evaluated will be discussed below. The translated consequent is stored in the rule structure's consequent slot. The translations of these two parts of the rule allow them to be compiled by the lisp compiler, resulting in a significant increase in system performance.

Rules are normally entered to the system from a file previously constructed with an editor. However, rules can be defined or re-defined at any time by the user. There is a slot in each rule's structure called its "implications." Each time a rule's consequent makes an assertion, a pointer to that assertion is added to the rule's implications. If a rule is denied (i.e., removed or re-defined), the TSA recursively denies all deniable inferences which resulted from it. Re-definition of a rule pushes the old version of the rule on to a kind of history stack for the rule, and the new version then replaces it as the current version. Whenever a new rule is added to a rule system structure, it is immediately run with an empty *causer* and an initial activity interval of **always**. This has the effect that anytime a new rule (or new version of an existing rule) is defined, all possible inferences producible from currently known information are created and added to the knowledge base. Deniable information which changes due to the change in the rule

[6]See section 4.1.

will be disavowed. Note that this process affects only deniable information. Inferences produced by the previous version of the rule which are either in the past or are within their execution period with respect to the current time are kept.

4.4.2 Rule Evaluation

RULESYS is forward chaining. That is, whenever a new piece of information is asserted to a rule system, the rule system searches its trigger list for an entry with the name of the attribute of the new assertion. If an entry is found, then the rules associated with it *may* apply to the new information. The rules are keyed in the trigger list by the *attributes* appearing in their antecedents. The rules selected at this stage each have at least one pattern in their antecedent which mentions the *attribute* of the *causer* information, but the complete pattern mentioning the attribute may not match the *causer*. For example, the attribute of the pattern:

```
(?airport operational-status vfr)
```

has the same attribute as the fact

```
(logan operational-status ifr)
```

but the pattern does not match the fact. A "situation" is constructed, consisting of the new information (represented by the changed fact and the activity interval over which the change occurred) and the set of rules that may apply. The situation is enqueued on the rule system's agenda. The agenda is a first-in-first-out queue. If the agenda was empty prior to enqueuing the new situation, then the rule system is told to process the agenda. This is accomplished through a call to the function **process-agenda**, which is provided with a pointer to the rule system structure as its only parameter. If the agenda was not empty, then the rule system is already processing the agenda and does not need to be told to do so again.

The agenda is processed one situation at a time. The fact, activity interval and one of the rules from the situation are initially sent to a function called **bind-variables**. This

function compares the antecedents of the rule to the values in the history cells of the fact during the activity interval. When an antecedent matching the fact is found, bindings are made appropriately. The program then executes the **run-rule** function on the rule and bindings, but with the matched pattern removed from the antecedent. This function first checks to see if there are any more patterns to match. If there are not, then any testing of relationships about or among the bound variables is carried out. Should those tests be passed (or if they are non-existent), then the consequent is executed. If there are more patterns in the antecedent, then **run-rule** chooses another one, substitutes into it any values already bound to any of its variables, and begins to search for a matching fact.

The search begins by finding all facts in the database which have the same *attribute* as the pattern in question. If there are none, then the default rules are searched to see if one of them can infer a default value for the missing fact. If there are any facts that have the required attribute, then **bind-variables** is called again, iteratively, over the list of candidate facts, with each new candidate, the remaining bindings, and the activity interval.

In **bind-variables**, if the *thing* part of the candidate fact does not match the corresponding part of the pattern, **bind-variables** returns immediately and another candidate (if there are any left) from the list is tested. If it does match, then **bind-variables** obtains the segment of the candidate's history which occurs during the activity interval. As this history segment may contain more than one history cell, the program must iterate over the resulting list of history cells. Each history cell in the list must have a unique activity interval which is a sub-interval of the time interval provided by **run-rule**. For each history cell in the list, appropriate testing or binding is accomplished and **run-rule** is called with the new bindings, the time sub-interval of the history cell and, again, the antecedent list sans the matched pattern.

It is possible that there might be more than one **value-cell** in any of the history cells found by **bind-variables**. If this is the case, then the values are treated as a set, and they are each bound consecutively to the pattern's *value* variable. **Run-rule** is called again with each new binding added to the bindings list and, of course, one less pattern in the antecedent. That is, the normal action of rule evaluation when a variable is bound to a set is *universal quantification* and *enumeration* as described in section 2.2.1. If the pattern's *value* is not a variable, then the set is searched to verify that the specified value is a member. If it is, then **run-rule** is called again, without the matched antecedent but with the same bindings as were passed in the call to **bind-variables**.

If no corresponding fact can be found at all, even through the use of a default rule, then RULESYS backs up to the last choice of a fact matching a previously matched pattern, and uses the next fact in the list having the proper *attribute*.

The successful execution of a rule's consequent will result in one or more assertions being made. These assertions are treated just like those made by the user. That is, each assertion may result in the enqueuing of another situation on the agenda. After the assertions have been made, control is returned to the last recursion of **bind-variables** which has values or sub-intervals remaining to examine. If those sets are empty, control reverts to the last recursion of **run-rule** which has remaining facts to attempt to match, and the process continues to exhaustion. In this way, RULESYS guarantees to produce every possible result that can be inferred from the available data and rules. If all of the facts in the initial call to **run-rule** have been tested, then control reverts to **process-agenda**, which takes the next rule out of the current situation and starts the whole process over again. If all the rules in the present situation have been evaluated, then the next situation is taken from the agenda and processing continues. If the agenda is emptied, RULESYS shuts down until new information arrives in the form of an assertion from the user or from another rule system.

4.4.3 Universal Quantification, Set Forming Rules, and Preference Rules

The use of `:additional t`, described on page 76 creates *sets*. For example, the rule:

```
(defrule ry1 tc
((?airport is-a airport nt)
(?airport op-status vfr)
(?runway owned-by ?airport nt)
(?runway headwind ?hw nt)
(?runway crosswind ?cw))
((and (> ?hw -10.0) (< ?cw 22.0)))
(rs-assert ?airport vfr-feasible-runways ?runway tc
:additional t))
```

will produce the *set* of feasible runways satisfying the criteria that the airport is operating under Visual Flight Rules, that the maximum tail wind for a member runway is less than ten knots, and that the maximum crosswind component for a member runway is twenty two knots for each final search interval resulting from the evaluation of the rule. The set has an activity interval, like every other piece of information; that is, the membership of a set may vary with time.

If the search for a match to a pattern results in a set, and the value part of the pattern is not a variable, the pattern is considered as matched if the pattern value is a member of the set. If the pattern value is a variable, on the other hand, the set is *enumerated*. That is, the rule is evaluated with *each* of the members of the set bound to the variable.

Rules which enumerate over sets act as *filters* on the sets. If these rules make assertions using the `:additional t` option, they will produce sub-sets or co-sets of the sets they reference (see equation 2.7). If they do not use `:additional` assertions, they act as preference rules, resulting in a single member of a set (see equation 2.8). This result is a preferred member, based on the preference criteria in the antecedent of the rule. The

preference criteria may indicate the "best" or "worst" or any other preferred member of the result set. The preference rule below chooses a member of the set of feasible runways at an airport to be that airport's primary arrival runway. The resulting arrival runway is the one with the least crosswind and most headwind. The consequent asserts that the specified runway is the primary arrival runway during the activity interval, that its use is allocated during the period, and that the comparison alternative runway is free for other use (for example, as a departure or secondary arrival runway).

```
(defrule choose-arrival tc
((?airport feasible-runways ?runway)
(?runway allocation available nt)
(?runway headwind ?nhw nt)
(?runway crosswind ?ncw nt)
(?airport arrival-runway-1 ?arway nt)
(?arway headwind ?ohw nt)
(?arway crosswind ?ocw nt))
((and (< ?ncw ?ocw) (> ?nhw ?ohw)))
((rs-assert ?airport arrival-runway-1 ?runway tc)
(rs-assert ?runway allocation used tc)
(rs-assert ?arway 'allocation 'available tc)))
```

The rule is triggered by any *addition* to the feasible runways at an airport. A separate rule is needed to re-infer what the arrival runway should be if the chosen runway is later found to be infeasible for some reason.

4.4.4 Existential Quantification, Cut, and Default Rules

Existential quantification (see equation 2.9 on page 44) requires the termination of the evaluation of a rule enumerating a set upon establishing a single true conclusion, even if there are other true conclusions which could be inferred by the rule. This requirement is

satisfied through the establishment of a non-local exit node[7] so that rule evaluation will be terminated upon a single execution of the consequent. The name of the mechanism to accomplish such a non-local exit (borrowed from PROLOG, which has a similar feature) is **cut**. The syntax used for **cut** is simply to wrap the last assertion of the consequent in a cut form, thus:

```
(cut (rs-assert ?runway 'localizer 'operational tc))
```

A *default rule* is a special rule which is used to determine a default value for some information which does not exist in the rule system's database but is desired in order to "run" some rule. Often this is used to initialize a fact for a preference rule. Default rules *always* **cut** after an acceptable value has been determined, so using **cut** in a default rule is superfluous and unnecessary (and will produce an error). Default rules are defined using the **def-default** form:

```
(def-default <rule-name> <owning-rule-system>
(<patterns-to-match>)
(<tests-to-perform>)
(<assertions>))
```

The first pattern in the **<patterns-to-match>** is treated differently in a default rule than in a regular rule. This pattern is used only to *find* the default rule. It is a pattern describing what is *not* known, and is never used for binding purposes. Note that default rules are only invoked during the evaluation of regular rules. The last assertion in the **<assertions>** must establish the default information in the database. This is important because the implied **cut** returns the default value directly to the regular rule requiring it. This saves time, as the regular rule does not have to search for the new information. The setting of a default is otherwise treated the same as any other assertion. That is,

[7] A non-local exit node is a specific address on the execution stack. Once such a node is established, *any* program seeking to terminate *the entire sub-process* below the node may do so by transfer to the node address. The node will return the value supplied to it by the transfer point that executed the non-local exit.

the rule system trigger list will be scanned for applicative rules for the assertion of the default, and if any are found a situation will be enqueued on the rule system's agenda.

4.4.5 Assumptions and Paradoxes

It is possible, in a complex system of rules, for a situation to occur in which two different rules conclude different values for the same fact over the same activity interval. This situation is the practical equivalent of a paradox, and it must be resolved for RULESYS to operate in a logical manner. As pointed out by [Doyle 79], the problem may not be as simple as it first appears. This is because the two conflicting results may be the end points of chains of reasoning involving several rules and many facts. Somewhere in the chain of reasoning of one of them there must be either a mistaken observation (input by the system's user) or a rule which concluded an incorrect result.

Of course, a default rule, acting on lack of knowledge, could conclude an incorrect result. If this is the case, the entire chain of facts leading from it must be withdrawn by the truth maintenance system. The problem, of course, is to ascertain *which* fact and associated chain of reasoning is incorrect. To make this somewhat easier, facts resulting from default rules are marked as assumptions, and this marker is inherited by all succeeding facts resulting from any rules which use the assumption in their antecedents. Thus, if a paradox should appear and one of the conflicting facts carries the assumption marker, RULESYS *assumes* that the chain associated with the marked fact is the one to withdraw. In addition, the existence of the conflict temporarily invalidates the default rule which was responsible for the problem. The default rule is marked as unusable for the interval in which the paradox occured, and the *other* (i.e., non-marked) fact is indicated as the reason why it is not applicable. If that fact should be denied later, the disabled default rule will become usable again during the period, and may be re-invoked.

The above scenario is the simplest of such situations; the assumption is an obvious candidate for backtracking. In more complicated circumstances, there may be more than

one candidate assumption and no easy way (perhaps no way at all) to determine which one should be retracted. For this reason, default rules are rarely employed accept for providing a "seed" for universal quantification (see section 4.4.3).

4.4.6 Disjunction and Run-rule

Two different rules may have identical consequents. That is, two rules may assert the same things for *different reasons.* This is the method by which a knowledge engineer would express a rule with a causal *OR.* It is not unusual that there may be a third rule which should be evaluated if one of the disjunctive pair succeeds but not if the alternative succeeds. This could be accomplished by having the third rule re-match some or all of the antecedents of the specific rule it is to follow, as well as being triggered by the effect of the consequent of the *OR.* However, that would be very inefficient, since at the time of the expression of the consequent it is known whether the third rule should be evaluated or not. In order to take advantage of this knowledge, the **rs-run-rule** form may be employed to enqueue a situation with a specific rule and an initial set of bindings onto the agenda. The syntax of this form is as follows:

```
(rs-run-rule <rule-name> <shared-bindings> <rule-system>)
```

where **<rule-name>** is the name of the rule to be enqueued, **<shared-bindings>** is a list of bindings in the current rule that should be used in the target rule, and **<rule-system>** is, as usual, the name of the rule system in which the target rule is to be found. The target rule need not be triggered by *any* of its antecedents. It should be noted that this mechanism adds nothing logically to RULESYS. Its use only serves to increase the run-time efficiency of the rule system. Experience has shown that the major amount of time consumed by RULESYS in evaluating rules is spent in searching the knowledge base for pattern matches, so the increase in system responsiveness due to this strategem can be considerable.

4.4.7 Requests to the Planner/Scheduler

The consequent of a rule may contain a request for the scheduler[8] to create a plan and schedule its execution time in order to accomplish a desired result. The form of such a request is

```
(schedule <thing> <attribute> <value> <rule-system>)
```

where the activity interval will be determined in the same way that was used for **rs-assert**. Unlike **rs-assert**, the temporal extent may *not* be included. An example of a rule making such a request is:

```
(defrule sched-snow-removal tc
((?airport arrival-runway-1 ?runway)
(?runway snow-depth ?snodpth))
((> ?snodpth 1.0))
(schedule ?runway snow-depth 0 tc))
```

which states that if at some time the airport's desired primary arrival runway has more than one inch of snow accumulation, then a plan should be constructed and scheduled so that at the beginning of the activity period that the runway is required, it will be clear of all snow.

4.4.8 Order of Execution

Those familiar with rule based expert systems may see some similarity between RULESYS and the PROLOG system. These similarities are superficial, and the reader is warned not to rely on them.

In particular, PROLOG is a back-chaining system while RULESYS is forward chaining. Further, antecedent evaluation in PROLOG (and all other rule based systems with which the author is acquainted) is *ordered.* That is, pattern matching and evaluation

[8]See Chapter 5.

of each form in the antecedent of a PROLOG rule is done in a pre-specified order. In addition, the order in which the rules appear in the rule system determines the order in which they will be evaluated at run time.

RULESYS does *not* have any pre-determined order of evaluation of either rules or their antecedents. While it is true that the rules are entered into the rule system trigger list so that the last one entered will be the first to be evaluated in a situation, this order is completely arbitrary and nothing in RULESYS depends upon it.

The choice of segregating the pattern matching part of the antecedent from the testing part guarantees that all the bindings have been made before testing is permitted. The order in which the patterns are matched depends on what information caused the situation in the first place, and that often changes since a rule may be triggered by more than one pattern. The inspections performed in the test section of the antecedent are done in an implicit conjunction. Because this section is translated into a lisp function, the order will be exactly as the user specified. This is still quite arbitrary, because of the commutative property of conjunction.

4.5 Reports and Queries

There are a variety of reporting levels available to the user. Because of the exhaustive forward chaining reasoning paradigm employed and the nature of the rules describing the dynamics of a given domain, there are usually a large number of inferences that result from any change of domain scenario. Most of these inferences are intermediate results which, while of paramount importance to the internal reasoning process, are of little interest to the user. The rule system can be informed of what the user considers reportable events. Under normal operating conditions, RULESYS will report on those events without being queried. This is accomplished through the assertion of a special fact with the form:

```
(tell <rule-system> user interested-in <interval> <thing>
:additional t)
```

For example, the user might want changes in inferred traffic delays at Logan reported during the current evening. The user can inform the rule system of this with:

```
(tell tc user interested-in ((this-evening))
'(logan arrival-delay)
:additional t)
```

It should be noted that this fact is established no differently than all others. That is, it could be established, via the **rs-assert** form, in the consequent of a rule instead of directly by the user. Note also that outside of the indicated activity interval, arrival delays will not be automatically reported.

The user is free at any time to query the rule system for information. There are currently six different queries that the system will answer. They are:

(what-is <thing> <attribute> <time-interval> <rule-system>) will display the value(s) that *thing's attribute* is believed to have during the specified interval.

(what-about <thing> <rule-system>) displays everything that **<rule-system>** knows about the specified thing. This will produce a description of the entire history of every state variable associated with *thing* in the rule system. This can be a lengthy report.

(when-is <thing> <attribute> <interval> <value> <rule-system>) searches the history of *thing's attribute* during the specified *interval* for periods when **<value>** is valid, and produces the set of sub-intervals when it is. The symbol "-" can be used for the **<interval>** to indicate **always**. Also, if the **<value>** is **nil**, the list produced will be when *thing's attribute* is unknown.

(**why-is** <thing> <attribute> <interval> <value> <rule-system>) will display the context(s) that were used to produce the specified information, if it is true. This is very useful when debugging a large or complex set of rules.

(**why-isnt** <thing> <attribute> <interval> <value> <rule-system>) does a complex analysis. This query does a kind of back-chaining, scanning the *consequents* of all the rules in the rule system for rules which could assert the specified information. Whenever such a rule is found, the rule system fact database is searched (constrained by the <interval>) for a match to each pattern appearing in the antecedent. Every time that *no match* is found, RULESYS reports that pattern to the user. If all patterns of a rule are successfully matched by known information, then the test section of the rule must have failed when the rule was evaluated, and the user is informed of that.

(**what-if** <rule-system> <thing> <attribute> <interval> <value>) has the same syntax as **tell**. This request will assert the specified information to the named rule system, but the assertion will be internally marked as a speculation. Other than this special marking, the assertion is treated normally. That is, RULESYS will infer everything that it can from it, but any assertions resulting from the use of a speculation are themselves marked as speculations, too. When the resulting agenda is empty, RULESYS will ask the user whether the result should be kept or discarded. The user may make other queries (see above) or further speculation using **what-if**. However, the user cannot make further definite assertions, using **tell**, until action is taken concerning the outstanding speculation. Possible user actions are **keep** and **discard**. If the user chooses to **keep** the speculation, all the assertions associated with the **what-if** are re-marked as definite and the system returns to normal. If the user chooses **discard**, all information marked as speculation are removed from the rule system's fact database and the system returns to its pre-**what-if** state. The syntax of **keep** and **discard** are very simple, as they have no parameters:

(keep)

and

(discard)

In addition to the above, there are two mechanisms designed to help knowledge engineers debug rule systems. The first of these is the **GLOBAL** parameter ***adebug***. This parameter is normally set to the default value **nil**. When it has *any other value*, **all** rule systems will report every assertion, the name of the rule system in which the assertion is being made, and the assertion's author when it is asserted. An assertion's author is "**USER-INPUT**" if the user tells the system the information, and is otherwise the name of the rule resulting in the assertion.

The second mechanism allows specific rules to be monitored. The command

```
(monitor <rule-name> <rule-system>)
```

allows the user to inform a rule system that whenever the named rule is evaluated information concerning that process is to be provided. One can monitor as many rules at a time as one wishes, by using **monitor** for each rule name. Rules can also be removed from monitored status with

```
(un-monitor <rule-name> <rule-system>)
```

The **GLOBAL** parameter ***rdebug*** is used to control whether monitoring is on or off. The default value of ***rdebug*** is **nil**, which indicates that no monitor reports should be generated. If ***rdebug*** is set to *any other value*, then monitor reports will be generated by *all* rule systems which have been told to monitor rules.

The lisp primitive **setq** may be used to set the values of either ***adebug*** or ***rdebug***. For example,

```
(setq *adebug* t)
```

would set the value of ***adebug*** to "T", resulting in global assertion reporting. Such a form to set or reset a debug switch can be part of the consequent of a rule, as well, so there can be rules that turn on or off these global reporting switches.

4.5.1 When Things Get Really Bad

It is certainly possible to get a rule system into a state in which restarting it may be easier than fixing it. For example, if RULESYS detects a pair of paradoxical rules, it will stop processing and display an error message. The user may want to restart RULESYS after flushing the processing of the current rule if the problem is thought to have little effect on future decisions, or quit out of the rule system altogether and fix the rule(s) before restarting. Simply continuing processing after an error message and halt due to paradox discovery will cause RULESYS to essentially execute a "cut" which will stop evaluating the offending rule and continue processing the rest of the current situation and agenda.

Should RULESYS actually "hang", that is, stop processing without an understandable message, the user may return to the top level LISP listener. The command to do this varies with specific implementations of LISP, so the method to accomplish it will have to be determined for each particular version of LISP. In any event, the form

```
(process-agenda <rule-system-name>)
```

will restart the processing of situations on the agenda of the named rule system. An alternative would be to use

```
(rs-soft-boot <rule-system-name>)
```

which will reset the named rule system's agenda to empty and thereby stop the processing of the last new information it received. Finally, the command

```
(rs-reset <rule-system-name>)
```

will completely destroy the named rule system so that it can be recreated from scratch. This is not something that one would want to do often. However, knowing how to do it may come in handy someday.

4.6 System Resources

4.6.1 Managed Resources

There are a number of internal structures which RULESYS generates and uses as processing of information progresses, including patterns, situations, and contexts. If these structures were simply created, used, and then discarded, the system would generate garbage at a high rate and would spend time collecting and processing that garbage to the detriment of performance. With this in mind, a resource management program was designed early on to manage and re-cycle specified structures. While the running of the resource manager is completely hidden from the user, it can be queried to determine the status of the resources that it oversees. This is not something that the normal user will probably want to do, but it may be useful for a knowledge engineer. The command is:

```
(<resource-name> :report)
```

where **<resource-name>** is the name of the resource, such as "**pattern-resource**" or "**situation-resource**". The "**pattern-resource**" is the resource manager object for all of the patterns defined in all of the rules of all of the rule systems in a lisp process. Requesting the "**pattern-resource**" object to report will tell the user the total number of patterns that have been defined for all the rule systems running on the computer. The "**situation-resource**" manages the situation structures used by the rule system agendas.

Both **rs-soft-boot** and **rs-reset** reset all appropriate resources.

4.6.2 Long and Short Term Memory

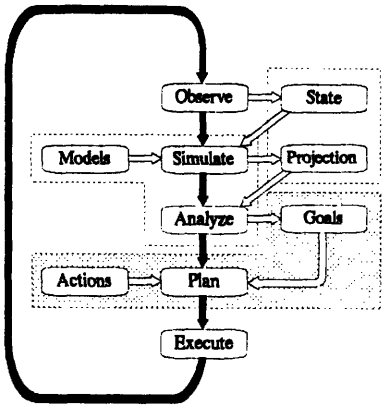
Rules, plans and actions (see Chapter 5) constitute the long term memory of the rule system, while facts and their history cells are the short term memory. For all practical purposes, the long term memory is fixed because rules and actions are rarely added or deleted. Short term memory, however, grows continuously as more history cells arrive from new observations and forecasts. The computer will "overload" with all this information if it is not allowed to forget what has happened in the past. The problem is that history cells describing the present and even the future may have been inferred from history cells whose activity intervals are in the past (i.e., the ending time of their activity intervals are less than now). And those history cells may have been inferred from yet older information. If the computer program is expected to be able to answer arbitrarily deep questions about the reasoning behind its decisions, then *all* facts which have been used in any inferences must be remembered indefinitely. If this information is not saved, then it may happen that the program will answer certain questions with, "I don't remember."

The current implementation of the program "remembers" everything that it is told. The problems it has been given to solve have been small enough, and change slowly enough, that the limitation of history cell space has not yet become a concern. This bridge has been left to cross another day.

4.7 Summary

The RULESYS temporal system analyzer is the "simulator" which produces and analyzes the projected future of the domain. The dynamics of the domain are defined by a set of rules. The use of a powerful set of temporal extent specifiers in the assertions appearing in the consequents of the rules allows the expression of a broad (and extensible) range of domain dynamics. A further capability to request the generation and scheduling of planned activity to change the expected behavior of the domain is also included. The

next chapter describes the functioning of the plan generator and scheduler.



Chapter 5

The Scheduler

The scheduler program has the responsibility of generating and scheduling an ordered set of steps which will bring about the accomplishment of a specified goal. A *goal* is a desired state of the domain at a specified time, which will not occur naturally. The goal is supplied to the scheduler by the Temporal System Analyzer in the form of a history cell describing a desired state of the world. For example, the scheduler might receive a goal history cell stating:

(bos-33l snow-depth 0 18)

indicating that it is desired that all snow be removed from Boston runway 33L at 18:00. The meaning is that the runway should be treated in the appropriate manner so that by 6:00 p.m. the runway should be clear of all snow. Notice that the request does not specify *how* the job is to be done, but only the result. It is up to the scheduler to determine how it should be accomplished, if it can be done at all, and report its findings back to the temporal system analyzer. The report takes the form of a *plan*, and is stored in the temporal database in a "fact" called *user plans*.

Plans are made up of *actions*. An action is an activity, carried out by one or more actors, which results in a change in the state of the world. Note that the passage of

time alone is a change in the state of the world, so doing nothing at all (i.e., *waiting*) is an action. An action is an activity over which the scheduler has control. For example, changes in the weather are not actions, even though they result in changes in the state of the world. Clearing snow from a runway is an action, since it is something which the operator can choose to do.

The representation of actions presented here is an extension of the standard which has been developed over the past twenty years (see [Fikes 72a, Fikes 72b, Sacerdoti 75, Charniak 85, Hendler 90]).

5.1 Historical Background

A planner's knowledge is contained in a database of plans and actions which are provided to it by a knowledge engineer. Plans and actions generally contain variables, in the same way that rules do. The planner creates specific plans for the accomplishment of a goal by combining and editing generalized plans and actions and binding the variables. This process is called *instanciation* of the plan.

The simplest description of an action is a two part structure consisting of a set of preconditions and a set of results. The results characterize the effects on the domain if the action were to be executed, while the preconditions represent the states of the domain under which the action may be executed.

The basic paradigm used by all automatic planners is based on the idea that a goal may be satisfied by some action $\mathcal{A}_1$. If the preconditions of $\mathcal{A}_1$ are met by the domain, then the "plan" to accomplish the goal consists simply of executing action $\mathcal{A}_1$. If the preconditions of $\mathcal{A}_1$ do *not* already exist in the domain, then they, in turn, become goals and the planner must search for appropriate actions to bring them about. The actions to satisfy the preconditions must be carried out prior to the execution of $\mathcal{A}_1$ so that their results are available to satisfy the preconditions. Each of the actions found to satisfy these precondition goals may in turn have preconditions which do not exist in the domain, and

the system must find actions to satisfy them, and so on. This plan generation process goes on until all of the preconditions of the earliest actions in the plan exist in the domain state.

Usually there is more than one action which satisfies a specified goal, and the planner must choose among them. The choice of which action to use in a given situation may be critical. Most actions have several effects on the domain, and it is not unusual that an action $\mathcal{A}_3$, which satisfies a precondition goal of a later action $\mathcal{A}_2$, causes a precondition of a yet later action $\mathcal{A}_1$ *not* to be satisfied. In such a case, action $\mathcal{A}_3$ is said to "clobber" a precondition of action $\mathcal{A}_1$. When this happens, it becomes necessary to choose an alternative to $\mathcal{A}_3$ which will still satisfy the precondition of $\mathcal{A}_2$ but will not "clobber" action $\mathcal{A}_1$. If this problem is not discovered until after generation of a plan to satisfy $\mathcal{A}_3$'s preconditions, the entire structure from $\mathcal{A}_3$ back must be abandoned. This is called "backtracking", and can be very computationally expensive. Unfortunately, it is unavoidable in the general case of planning. Even human planners are often forced to backtrack and reconsider some portion of a plan.

A plan can have various unrelated parts. If the backtracking system is based on saving the domain state at each decision point, no information is kept about what parts of the plan depend on the decision and the entire tree before the errant choice must be discarded, including those parts that are not effected by the choice. Tate [Tate 75] developed the idea of "dependency directed backtracking", in which the dependencies between decisions and alternatives to those decisions are saved. Backtracking is accomplished by recursively undoing only the parts of the plan that are dependent on the particular choice.

Sacerdoti's [Sacerdoti 77] *procedural nets* attempts to avoid backtracking altogether. Essentially, Sacerdoti noticed that the particular choice of actions to create a plan is a kind of knowledge. The knowledge base of his planner contains both actions and plans. When his system is required to develop a plan to satisfy a single goal, it can, for the most part, simply find an already perfected plan in the database, and return it. The

real power in his planner is displayed when it creates plans for conjunctive goals. The planner represents the overall plan to accomplish the conjunct as a network containing the individual plans to accomplish each goal separately (see Figures1.4 and following in Chapter 1). The planner then employs several transformations to combine and order the individual steps in the separate plans into a final plan satisfying the conjunct. Since the pre-stored plans are, by definition, workable, and the transformations for ordering steps for conjuncts check for difficulties *before* the ordering takes place, backtracking never occurs.

5.2 A Scheduler

A *scheduler* is a planner that determines the execution times of the actions, thereby determining the order of execution, i.e., a temporal planner. The new scheduler described here extends the idea of an action by including a time interval which is required for the action to be performed. It also utilizes pre-stored plans and transformations for combining and ordering combinations of plans. The inclusion of time in both the domain and the actions requires changes and extensions to the mechanisms previously applied to achieve the transformations.

The steps of a plan may be specific actions or goals, or even other plans. In fact, a plan may be considered as an action at some level of abstraction. With this in mind, a broader definition of an action, to include that of a plan, is desirable. In the remainder of this discussion, the terms action and plan will be used interchangably.

Because the system being characterized is an advisor, primitive actions may be described by a request to the user to perform some activity, and the result of carrying out that activity on the domain. Requests are stored in the procedure slot of the action. When the action is to be executed, the request is made to the user. The general assumption is that the action has been carried out unless evidence to the contrary is presented to the TSA. It should be remembered that information describing present and forecasted

domain states, in the form of history cells, is constantly being provided to the TSA. These new history cells may replace one or more history cells already in the temporal database, thus triggering replanning if necessary. In essence, this yields an implicit form of "plan execution monitoring" which gives the overall system a reactive nature or feedback mechanism. The schedule and content of the plans derived by the scheduler react to changing environmental circumstances.

The temporal database manager supports both assertions and denials, so the "add-list" and "delete-list" of the STRIPS representation can be combined into a single slot called the action's *results*. In order to support compound actions (i.e., *plans*), a *procedure* slot is added to the representation of an action. The procedure of a plan describes the steps and order of execution required to carry out the plan. It is important to understand the difference between the preconditions and procedure of a plan. The preconditions of a plan are goals which must be met (i.e., must be true) before the plan may be considered as a way of attaining a goal. They are not ordered temporally or any other way. The procedure, on the other hand, is a *time ordered* list of actions and goals.

When the scheduler receives a goal pattern, its initial reaction is to see if the goal is already represented by a history cell in the temporal database. Of course, if it is, no further processing to achieve the goal is necessary and no schedule to accomplish it need be generated or scheduled. If such a history cell does not exist, the schedule generation process begins by pattern matching the goal pattern against the result assertions appearing in the plans and actions stored in the scheduler's knowledge base.

This first matching step and the temporal information provided with the goal make up the initial context for schedule generation. As with the evaluation of rules described in the previous chapter, this context is represented by a set of variable bindings. The preconditions and/or procedure of the candidate action may refer to variables which do not appear in the result assertions, and therefore will not be bound by the initial match step. The additional patterns required for determination of these unbound variables are

located in the "context" slot of the action.

The same pattern matching subroutines are used for matching both the antecedent patterns of rules and the context patterns of actions. The resulting variable bindings may then be used by substitution in evaluating the tests, preconditions, and procedure of the action.

Once all variables are bound, their values can be tested to determine the suitability of the candidate action. These tests are contained in the action's "test" slot. If a test fails, the action is not appropriate to the current domain situation and is discarded. The next candidate action is considered. If there are no more candidate actions, the scheduler must report that the goal is unattainable.

Just because all the tests succeed does not necessarily mean that the action is appropriate to the situation. All of its preconditions must either be true or attainable before the scheduled time of execution of the procedure of the action. Thus, the time when the preconditions must be satisfied is the time the goal of the action is to be accomplished minus the time required to execute the procedure, called the action's *initiation time*. First, the time map manager is queried to determine if each condition already holds. If it does, no further plan generation or scheduling for it is required. If a precondition does *not* hold, then the precondition is treated as a goal due at the initiation time of the current action. The eventual result of this recursion is a scheduled network of activities, i.e., a *possible schedule* to accomplish the goal which was requested by the TSA.

Goals may appear in the procedure of a plan. They express the possibility that any one of several possible actions may be used to accomplish a step in the plan. Another way of looking at this is that these unsatisfied goals allow the scheduler some freedom of choice about the plan. Before execution, all of the goals of a plan must be satisfied by an executable plan or action. Of course, this additional freedom has a price: backtracking becomes a possibility. More on this in Section 5.3.

The temporal aspect of a plan or action is the time that will be necessary to execute it.

Knowledge of this interval is of supreme importance to the scheduler, because this time is subtracted from the desired time of completion to schedule the beginning of the activity. For primitive actions such as turning on a runway lighting system, the execution time may be considered a constant. However, the execution times required for most actions are highly dependent upon the variable bindings resulting from instanciation. For example, the time necessary to climb a ladder depends on how long the ladder is, and the time necessary to plow the snow off of a runway depends on how long the runway is, how heavy the snow, and how many plows can be employed at the time. In the case of plans, the situation is even more complicated. The time necessary to carry out a plan is the sum of the times necessary to carry out each of its steps. But since any of the steps in a plan could be a goal, and the particular method used to accomplish a goal will be decided during (or after) the instanciation of the plan, it is generally impossible to know *a priori* how long the execution of the plan will take. In these cases, an initial "worst case" interval for completion of the activity must be provided by the knowledge engineer so that an initial schedule time can be produced for the activity.

The primary goal supplied by the TSA may generate many secondary goals, each of which may require searching the action knowledge base to obtain candidate actions. However, this search *need not be done each time the scheduler considers how to accomplish an unsatisfied goal.* In fact, it need only be done once.

Like rules (see Chapter 4), plans and actions are pre-scanned at the time they are defined by the system knowledge engineers. Each time a goal is detected, the plan knowledge base is searched for actions which have a match for the goal in their results slots. A list of references (pointers) to these actions is permanently associated with the goal so that anytime in the future the scheduler may immediately consider its choices for accomplishing the goal. In addition, whenever a new plan or action is added to the database, all of the procedures of prior entries are searched for goals that may be satisfied by the new action, and a reference to the new action is added where appropriate. Finally,

a table of all plans and actions, referenced by the patterns asserted and denied in their results slots, is created to improve the search efficiency when a new goal is received from the temporal system analyzer.

The content of the results slot of an action describes the changes to the domain that will take effect *after* the action has been executed. Because actions are no longer considered to take place instantaneously, another slot is required in the representation to describe the changes to the domain *while* the action is being executed. For example, the result of plowing a runway is that the snow depth on that runway is reduced to zero. However, *while* the runway is being plowed, there are vehicles (plows) on the runway and the vehicles are allocated to that task and are unavailable for other duty. This slot in the action structure is called the *active-effects*. Both the results and active-effects are a collection of assertions and denials of the same kind, and with the same kinds of temporal extent, as those appearing in the consequents of rules (see Chapter 4).

The domain described in earlier chapters supports multiple actors and parallel activity. As far as the scheduler is concerned, planned activities can take place at the same time whenever such parallelism is not specifically banned. Two actions considered for overlapping activity intervals must be subject to the same kind of analysis that Sacerdoti and others (see section 1.3.1) have used to determine ordering in single actor systems. In a way, the analysis is somewhat simplified, since any effect of an action whose activity interval ends prior to the beginning of the activity interval of a requirement for a precondition of another action may be ignored.

There is an additional twist, though. When planners of the past have considered conjunctive goals, the goals are presented to the planner at the same time, and a single plan is produced which will satisfy the goals. In the current system, data (in the form of facts) is continuously arriving and the temporal system analyzer's picture of the future is constantly being extended and (more importantly) *changed*. This fact brings up the very real possibility that the scheduler may be given a goal to achieve at a certain time and

successfully build a schedule to perform it, and then later the *TSA* may obtain new information indicating another goal to be satisfied. The scheduler *can not* simply regard the second request independently of the first if the schedule activity periods overlap. At the same time, the scheduler can not ascertain whether the activity periods of the schedules overlap without creating an initial schedule to satisfy the second goal independently of the first schedule. The only way out of this predicament is to create the second schedule independent of the first, and then transform the pair of schedules together.[1] As a result, there exists *only one "schedule" at any given moment.* That is, either there is no schedule indicated for a period of time, or there is one schedule, which may involve a number of parallel activities to accomplish several goals. There may be different schedules for different, non-intersecting intervals, but if any activity interval of one schedule intersects an activity interval of a following schedule, the two schedules must be combined into one contiguous schedule.

5.3 Handling Uncertainty with Plan Refinement

The future is, of course, uncertain. In Chapter 1 it was noted that the single actor assumption obviates the necessity to consider this aspect of reality. However, the real world application domains targeted by the current effort require that some mechanism be used to address this issue.

Sacerdoti pointed out in [Sacerdoti 77] that deferring the step ordering decisions of a plan would reduce backtracking and thus improve efficiency. This idea, known as *least commitment planning*, has been generalized to include deferring the binding of variables [Stefik 81b] and choices for sub-plans [Currie 85] until the system is forced to do so by constraints.

This technique is particularly applicable to the issue of uncertainty in a temporal

[1]This process can get very complex, as it may require not only re-ordering the steps of the schedules, but backtracking to *change* one or more steps so that the two schedules are not mutually exclusive. This kind of problem is not limited to computer schedulers; it is a ubiquitous and unavoidable difficulty common to to all schedulers including humans.

environment. Initial schedules can be made in a coarse and fuzzy manner, without committing to details unless necessary. As more information arrives, as a result of observation and the receipt of more forecast data, further commitment to a specific course of action can be made.

As a simple example, consider the removal of snow from a runway at some future time. The forecast calls for snow and predicts that the accumulation will be from one to three inches. If the snow accumulation at the time chosen to clear the runway is less than two inches, the runway may be brushed rather than plowed. Brushing a runway takes only about one third the time necessary to plow it, so brushing would be preferable. If it were necessary to specify the method as part of the schedule from the beginning, the scheduler would have to select plowing since that is the worst case. The scheduler can, instead, set aside enough time for the worst case, but leave the option of which method to use open. If further information regarding the accumulation depth becomes available, the scheduler may be forced to make a choice. In any case, if the time that the worst case option should be initiated arrives, the scheduler must choose one of the options.

As noted above, schedules are stored as facts (complete with activity intervals, of course) in the temporal database. Whenever new knowledge arrives (i.e., a new history cell is asserted to the temporal database manager), schedules with activity intervals intersecting the activity interval of the new history cell are re-submitted (by the temporal system analyzer) to the scheduler along with the new history cell. If the new history cell causes a change in a schedule, the changed schedule is reported back to the TSA and updated in the temporal database. This is the mechanism by which a coarse schedule is refined as additional information becomes available. In this way, a schedule may be re-submitted for refinement several times before it is actually executed.

This method will only work when the scheduler has no data with which to decide on a course of action. Once the necessary information arrives, it is forced to make a choice. Of course, as noted in previous chapters, all data is only belief, and is subject to change.

In the event that a fact used to make a decision *changes*, the scheduler has no choice but to backtrack and reformulate the affected schedule.

Even this problem may be rectified to some degree. Each action could be marked with a maximum time interval *from the present* beyond which elaboration would not be allowed. That is, each action would be supplied with a "temporal horizon" beyond which planning using it would not be permitted even if history cells for the period of interest exist in the temporal database. This mechanism would essentially model the falling "level of trust" of forecasts of events too far in the future. This mechanism has not been implemented in the current program. Instead, a single horizon can be defined for the whole system.

Recently some work has been done in the area of probabilistic reasoning applied to planning [Dean 88, Dean 89b, Kanazawa 89]. The ideas expressed in these works seem applicable to the kinds of problems addressed here, but no attempt has been made to incorparate them into this planning program at this time.

5.4 Operation of the Scheduler

Operation of the scheduler is primarily controlled by the Temporal System Analyzer, through the mechanism of "schedule" requests (described in Section 4.4.7) appearing in the consequents of certain "analytical" rules. Prior to execution time, schedules may be returned for possible modification to the scheduler by the TSA if new history cells are encountered which have activity intervals which intersect the scheduled activity interval of the schedules. As now reaches the start of the activity interval of each action, its requests are reported to the supervisor, and its active-effects are asserted. When now reaches the end of the activity interval of each action, the results are asserted. To the supervisor operating the program, all this appears to be automatic.

5.5 The Plan and Procedure Editors

While there are similarities between rules and plans, plans are considerably more complex structures (see Section 5.2). Because of this added complexity, a special editor program is provided to ease the job of entering actions and plans into the scheduler's knowledge base.[2] The program is actually *two* editors; an "outer" editor which handles references to all of the parts of an action except the procedure, and an "inner" editor which only deals with the contents of the procedure slot.

The internal representation of the scheduler's knowledge base is a complex network, as described earlier in this chapter (page 123). Each rule-system structure (see Section 4.1) contains one of these networks in the **action** slot. As the editor makes changes of any kind on the knowledge base, the entire network is updated. To start the editor, a rule-system structure must already exist. The command to create a rule-system object is also described in Section 4.1. The command to edit the actions and plans in a given rule-system is:

```
(edit-actions <rule-system>
```

On receipt of this command, the system will respond with the prompt:

```
Edit-Action>
```

and will await an editor command. The commands are:

? prints the list of all editor commands.

list-actions lists all of the primitive actions and plans in the rule-system.

show displays a user-readable representation of the current action or plan object.

[2] After completing this editor and putting it to use, it was decided to construct a similar editor to more easily maintain a rule base for the Temporal System Analyzer. It has not been completed and will have to await a later version of the program.

new sets the current object to a "new" (empty) action. Warns the knowledge engineer if the current object has not been saved prior to the issuance of the **new** command and allows the user to back out gracefully if desired.

quit exits the editor. Like **new**, **quit** will warn the user if the current action object has been changed since it was last saved, and allow the user to decline the exit if desired.

save installs the current action object in the knowledge base; determines all cross references and stores pointers as appropriate.

name allows the user to specify by name an action object to edit. Warns if the current object has been changed but not saved, and allows user abort.

rename permits the user to change the name of the current action object. **rename** will not permit renaming an object to the same name of another object in the knowledge base.

delete eliminates the current action object from the knowledge base. Warns user and specifies the name of the object that is about to be deleted, and allows the user to cancel if desired.

tests lets the user input test patterns which will be used to bind variables.

add-test prompts the user to input one test pattern.

del-test displays a numbered list of the test patterns belonging to the current action object, and prompts the user to identify which one to delete. User response is a number, and the corresponding precondition is deleted from the list. Responding with anything except one of the numbers in the list causes the delete to abort.

reqs allows the user to enter precondition patterns for the current action object.The list of patterns is terminated with an empty list, i.e., **()**. **NOTE:** This command

replaces the precondition list that was on the current object with the new list. It deletes the old precondition list.

add-req prompts the user for a single pattern which is added to the precondition patterns already present.

del-req displays a numbered list of the precondition patterns of the current action object, and asks the user which one to delete. User response is a number, and the corresponding precondition is deleted from the list. Responding with anything except one of the numbers in the list causes the delete to abort.

while prompts the user to enter assertions or denials (just like those in the consequents of rules) describing the active-effects of the action during its execution. Like **req**, end with a () entry.

add-while prompts for a single assertion, which is added to the active-effects of the current action object.

del-while displays a numbered list of the active-effects assertions, and asks the user which one to delete. User response is a number, and the corresponding assertion is deleted from the list. Responding with anything except one of the numbers in the list causes the delete to abort.

proc starts the procedure editor. The procedure editor will respond with the prompt "**EDIT-PROCEDURE>**". For details about procedure editor commands, please see below.

results allows the user to specify a list of result assertions and denials (just like those in the consequents of rules) describing the result effects of the execution of the action. Like **req**, end with a () entry.

add-result prompts the user for one assertion or denail which is added to the result list of the action object.

del-result displays a numbered list of the results assertions, and asks the user which one to delete. User response is a number, and the corresponding assertion is deleted from the list. Responding with anything except one of the numbers in the list causes the delete to abort.

time allows the user to enter the default maximum time that should be allotted for execution of the current action object if not enough information is available to compute an estimate. Must be a number, in minutes.

tf displays the precondition patterns so the user has easy access to the local pattern variables, and prompts the user for a LISP form to be used to compute the time necessary to execute the current action object. Variables defined in the preconditions may be used in the expression.

Note that all editing is done to a *copy* of the actual action object, so an edit may be aborted with **new** or **name** without saving the current object.

The content of an action object's procedure slot is an *ordered* list. The order specifies the stepwise order in which the actions will be executed. Each member of the list may be one of:

1. The name of an action which exists in the current scheduler knowledge base, which will be expanded appropriately.

2. The name of an action which does NOT appear in the current knowledge base, which is assumed to be a message to display to the user indicating an activity to carry out.

3. A goal, in the form of a pattern, which the scheduler can satisfy with any action in the current scheduler knowledge base that asserts a match to the pattern.

The procedure editor employes a pointer which points at *one* entry in the procedure list. The starting position is always the *first* entry in the list. The procedure editor, started

from within the action editor, has the following commands:

`?` prints the list of all editor commands.

`show` displays the contents of the current action's procedure slot.

`quit` returns to the action editor.

`>` moves the pointer one step forward in the procedure.

`>` moves the pointer one step backward in the procedure.

`ib` "insert before". Prompts for an entry, and, upon its receipt, inserts it chronologically just before the current entry.

`ia` "insert after". Prompts for an entry, and, upon its receipt, inserts it chronologically just after the current entry.

`del` removes the current entry from the procedure list.

`list-actions` displays a list of all the actions known in the current scheduler knowledge base. This is useful if the user has forgotten the name of an action that is desired for addition to the current procedure.

In addition to the editors, there are two specialized functions to save and retrieve the scheduler knowledge base to or from a file. These files are editable with a standard editor, such as EMACS, but are not particularly "user friendly". It is highly recommended that users employ the editors described above. This will lead to faster system development and fewer errors, because of the extensive cross-referencing and error checking that the dedicated editors perform. The command to save the scheduler knowledge base contained in a rule-system object is:

```
(save-actbase <rule-system> <file-name>)
```

where <rule-system> is the name of the rule-system object, and <file-name> is a file system path name which **must** appear in double quotes ("). The command to retrieve the contents of a saved scheduler knowledge base into a specific rule-system object is:

```
(load-actbase <rule-system> <file-name>)
```

and the same definitions apply.

The function **display-actions** requires the name of a rule system and displays the names of the entries in the scheduler's plan and action knowledge base. Its syntax is simply:

```
(display-actions <rule-system>).
```

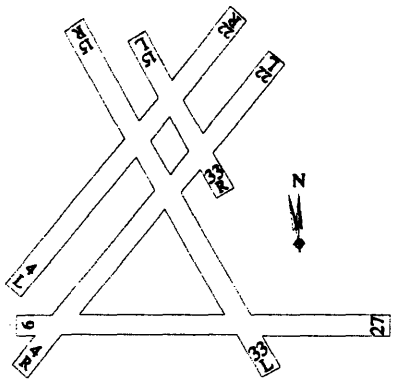
A related function, **detail-actions**, will display all the non-empty slots of each action contained in the knowledge base. It has identical syntax to **display-actions**.

5.6 Summary

In this chapter the operational mechanisms for a new, extended scheduling plan generator (a "scheduler") program were presented. The structures used in this program, like those described in the previous chapters, employ representations changed and expanded in order to incorporate temporal attributes. These changes require changes in the old methods of plan production, particularly in the area of conjunctive goal planning. This is because in a constantly changing temporal domain, a single new schedule request may result in an effective conjunct with an already asserted schedule from an earlier production. In such an environment, backtracking is sometimes unavoidable due to uncertainty and asynchronous acquisition of knowledge. Least commitment planning by delaying plan elaboration until absolutely required appears to be a powerful mechanism to ease this problem, though probabilistic methods are useful, too.

This chapter concludes the description of the overall process management and scheduling system originally introduced in Chapter 2. In the next chapter, an application of the

system to the domain of Runway Configuration Management will be presented.



Chapter 6

Case Study

The previous chapters have described the theory and implementation of a "proof of concept" rule based reasoning shell for process management. The original thrust of this research was to create a runway configuration management system for large, complex airports. As a result of the wide applicability of the solution and the novel ideas developed for it, the reasoning system became more interesting than the original problem it was designed to solve. However, that problem is still of interest, and this chapter presents a preliminary rule and plan base for it. Before that presentation, the problem domain must be more completely defined.

6.1 Runway Configuration Management

The FAA *National Airspace System Plan*[1] forecasts that the demand for aviation services will double within the next fifteen years. This increased demand for air transportation in the United States over the next decade has brought with it a requirement for a better organized and more efficient Air Traffic Control system. Further, air traffic is expected to rise world wide at a rate even higher than in the United States.[2] The

[1] See reference [NASP], page II-1 and following

[2] In particular, if the economic development of the Pacific Rim nations continues at its present level, many of the problems faced commonly in the United States now will plague those nations within a

increasing number of aircraft which are active within the system at any time presents some novel problems with which the current control system appears to be unable to cope.

In particular, the present FAA/NAS (National Airspace System) is occasionally operating near the limits of its capacity in the regions close to some of the nation's larger airports. This capacity isn't always the same as the theoretical maximum capacity of the runway system because the overall system may be limited by the airspace around the airport as well. If aircraft arrive stochastically at the average rate which saturates the runway system, queues will form requiring aircraft to be stacked in "holding patterns". During VMC[3], most airports even today have a higher capacity than the demand made upon them, so little queuing occurs. However, when the weather deteriorates to IMC[4], the capacity of the the airport generally decreases below that necessary to handle all of the incoming aircraft. If the demand is allowed to stay at the VMC level, holding will be required. Before the air traffic controllers' strike in 1981, the system was actually operated in this manner.

During the 1981 strike, in order to allow the system to continue to operate with a drastically reduced controller force, constraints were placed on the demand to seriously limit or avoid the formation of holding stacks. These constraints took the form of "gate holds." In essence, the aircraft are queued on the ground instead of in the air. To manage this task, a new controlling body was formed, called the *National Flow Control System.* In the name of safety and fuel conservation, this system is still in use today, and the gate hold has become infuriatingly familiar to many airline passengers.

Essentially, the flow control system is supposed to operate in the following manner:

The landing demand[5] at any major airport is a fairly well known figure. Long term estimates can be produced from the *Official Airline Guide*, while shorter term forecasts can be calculated using filed flight plans and position reports of enroute aircraft. Airport

---

decade.

[3]Visual Meterological Conditions, see FARS, reference [FAR].

[4]Instrument Meterological Conditions, see FARS, reference [FAR].

[5]Number of aircraft per hour requesting to land

capacity[6] is predictable as well, although it is a much more elusive quantity. Capacity is most strongly affected by the *airport configuration*, i.e. the runways in use and the associated approach and departure procedures.

Given the expected capacity and demand of the destination airport, flow control is designed to guarantee that the demand never exceeds the capacity and the traffic arrives in a homogeneous stream. Each hour, a limited number of landing time slots are available. To control the flow of aircraft into the destination airport, these slots are assigned to each inbound aircraft *before it takes off from its origin airport.* Since the number of slots is relatively small, there are occasions when all the slots for a given hour are filled. All other flights to the destination airport having an estimated time of arrival during this hour must be detained on the ground. Gate delays are assigned to these aircraft. The delayed flights are given landing slots in a future time period. Priority is based on the length of the gate hold. This process continues until all aircraft have been assigned slots.

Since information on predicted demand and capacity becomes less and less reliable in the future, there is an event horizon beyond which it is unrealistic to use the system. Given the average speed of a commercial air transport, this translates into a distance horizon. Thus, flights to a given airport that originate farther than the distance horizon must be exempt from the flow control process. Once enroute, a flight has priority to receive a landing slot. It is only considered for delay under extenuating circumstances, such as an emergency at the destination or on another flight, or if the capacity prediction was too high in the first place and the aircraft finds a congested airport on arrival.

On a fine, sunny day, when the parameters are changing slowly, prediction of demand and capacity can be made with considerable certainty, and the system works admirably. Most large airports under VFR[7] have capacity exceeding the current demand, and use of the flow control system is unnecessary. As the weather gets worse, or changes rapidly, the certainty of the predictions degrades and the planning horizon should shrink. However,

[6]The number of operations (landings and takeoffs) per hour the airport can accommodate.
[7]Visual Flight Rules, see reference [FAR]

allowing the planning horizon to contract beyond a certain limit would defeat the purpose of the flow control system. Instead, the horizon is held constant.

The flow control system attempts to balance capacity and demand. On the one hand, if the actual capacity turns out to be greater than the demand, then there are unused slots and the airport is not being used to its fullest. On the other hand, if the demand allowed by the operator exceeds the capacity, enroute and terminal area delays (*in flight*) will ensue, with their associated increased controller and pilot work load. The first condition is unfortunate and expensive in terms of lost capacity. The second scenario is more expensive in terms of wasted fuel, and may even be dangerous.

Clearly, one of the key factors to the success of the flow control system is the accuracy of the capacity and demand predictions. The availability of OAG[8], flight plan, and radar position data makes prediction of demand relatively easy. Capacity prediction, which depends on factors which are more difficult to measure or quantify, is harder. Since operating a jet transport is costly, and the number of flights involved is large, the return on even a slight increase of the throughput of the system is enormous. Thus, the interest in improving airport capacity prediction is high.

Because capacity depends to a great extent on the configuration, we must begin by looking at the factors which influence the configuration.

1. Runway conditions.

2. Wind velocity and direction.

3. Ceiling and visibility.

4. Time of day and season.

5. Noise abatement procedures.

6. Ratio of takeoffs to landings.

[8]See Reference [OAG]

7. Types of aircraft involved and the number of each type.

8. Configurations in use at surrounding airports.

9. Maintenance of runways and taxiways.

10. Snow removal.

Airports are diverse in their structure and surrounding environments. Some, such as Miami International, are relatively simple. Miami's runway set consists of two East/West parallel runways and a single Southeast/Northwest runway. While the city has grown around it, as happens to every major airport, the area close to the airport is mostly commercial. This reduces the noise abatement problems somewhat. The weather, except for an occasional thunderstorm or rare hurricane, is consistently mild. There are very few combinations that can be constructed with this small set of runways. The number of configurations is small and the transitions are straight forward.

Other systems are more complex. Boston's Logan International Airport has two sets of parallels and a single east/west runway. The airport is partly surrounded by water; there are tall buildings to the West; there is a shipping channel just west of the approach end of the only category three runway; and there are vociferous population centers under most of the airspace on all sides of the airport. Because it is located on the coast of the North Atlantic ocean, the weather can change rapidly and vehemently. In the winter, snow removal and ice treatments must be part of any planning. There are eleven major configurations and over forty minor variations of them that can be used in this system.

Chicago's O'Hare International has 14 runways and over one hundred configurations. It shares some of the environmental problems of Logan, such as rapid weather changes, and has noise abatement problems. It is also the busiest airport in the world.

Finally, the New York - Newark area's three airport system consisting of Newark International, LaGuardia, and John F. Kennedy International must be operated as a single entity. That is, the configurations of all three airports have to be planned, scheduled,

and changed synchronously, because the approach and departure routes of the airports interfere with each other.

Transitions between some configurations and others are very expensive[9]. The large number of choices of possible configurations, together with the complexity of the interactions between the factors outlined above, make the construction of a configuration plan over a several hour period a very difficult task. This, in turn, makes the prediction of airport capacity over such a period a process prone to much error and frustration.

Talks with *FAA* area supervisors at two airports (Boston Logan International and Miami International) indicate that configuration planning is *not* currently employed. Instead, a short term, tactical methodology is used. The area supervisor, acting on whatever information is available, decides what configuration is to be used at the *current time.* **There is no long term plan.** The numbers used for expected capacity in the flow control metering process are telephoned into Washington about every hour. How closely these estimates are to reality is a matter for conjecture.

Some of the items in the list above, such as runway conditions and configurations at nearby airports, are very difficult to quantify. This fact makes the application of linear programming techniques cumbersome and artificial. Use of linear programming requires an objective function which is "optimizable" in some sense. The solution generated by the LP results in some kind of *optimum.* However, the inclusion of arbitrarily quantified terms in the objective function leads to an artificial solution which has little basis in reality.

The usable configurations at an airport change from time to time. Work on runway surfaces or new facilities may invalidate some previously used configurations. On occasion, airports build additional runways. Political pressure from surrounding communities may require discontinuation of the use of one or more approaches. To be successful in this environment, a computerized assistant must be easy to modify and debug by the

[9]Entry into the approach area may have to be suspended during the reconfiguration period.

local control tower staff.

The *FAA* has shown interest in computerizing configuration management for some time. Work on the O'Hare Runway Configuration Management System, an ongoing project of the Mitre Corporation[10], has been underway for over ten years. This system is a first generation expert advisor which can be used for analysis of a single configuration *transition.* The approach is to quantify all the data and use a linear programming optimization scheme to arrive at a ranking of a number of different configurations that might be employed instead of the one currently in use. It does no long term planning, and furthermore, the rules used by the system are hard coded into the FORTRAN program, making changes in the rules and installation at other airports excessively expensive.

A configuration plan is a *pattern* of events which take place over a period of time. It is a type of *strategic plan*, like planned moves in a chess game or troop movements in a battle. The configuration plan is altered based on a forecast of future conditions. This is similar to the way a chess player plans a sequence of moves based on his prediction of what the pattern of the pieces on the board will be later in the game. It has been shown that expert strategic planners, such as chess masters, do *not* attempt to compute their plan of attack. Instead, they recognize current and future patterns of the pieces on the board and are lead by those patterns to an overall game plan[11].

The physical closeness of Logan International to MIT and the magnificent cooperation of the staff of that airport made it the logical choice for a first experiment.

6.2 Physical Description of Logan

Boston Logan International Airport is owned and operated by the Massachusetts Port Authority (MassPort). The physical layout of the airport is shown in figure 6.1. Each runway has different effective departure and arrival lengths and is equipped differently to handle a variety of aircraft and weather conditions. Table 6.1 describes the runways and

[10]See reference [MITRE 81].
[11]See reference [deGroot 65]

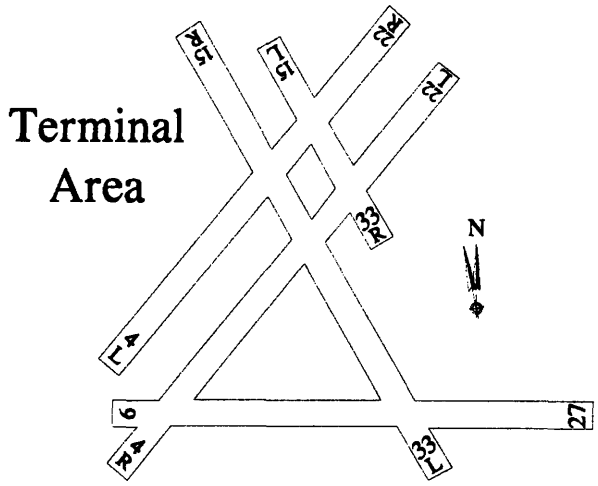


Figure 6.1: Boston Logan International Airport

Name	Direct-ion	Arrival Length	Depart Length	ILS	RL	ALS	TDZL	REIL	VASI	RVR
4R	035	(D) 8840	10005	III-B	HI	HI	X			X
4L	035	7860	7860		HI			X		
9	092	7000	7000		HI					
15L	150	2557	2557		MI					
15R	150	(D) 9191	10081	II	HI	MI	X		X	X
22L	215	(D) 8796	10005	II	HI					X
22R	215	(D) 7032	7860		HI				X	
27	272	7000	7000	II	HI			X	X	X
33L	330	10081	10081	II	HI	HI	X		X	X
33R	330	2557	2557		MI					

ILS=Instrument Landing System, RL=Runway Lights, ALS=Approach Light System.
TDZL=TouchDown Zone Lights, REIL=Runway End Indicator Lights.
VASI=Visual Approach Slope Indicator, RVR=Runway Visual Range.
(D) indicates a displaced threshold.
II and III-B indicate the category of the ILS.

Table 6.1: Logan Runways

the equipment associated with each. Note that the two ends of each length of concrete are considered to be separate runways, so there are *ten* actual runways.

Runways are named by the nearest rounded up ten degree magnetic mark to their direction. Thus the runway whose magnetic direction is 35 degrees is named "runway 4." If a runway is one of a parallel pair, an "R" or "L" (for Right or Left) is appended to the name as appropriate.

The most obvious attribute of a runway is its length. Heavy category aircraft[12] generally require longer runways for their operations. Runways at Logan range in length from 2,557 feet, which can handle only small single and twin engined private and commuter aircraft, to 10,081 feet, which can handle the largest and heaviest planes now in production. All runways at major airports are at least 150 feet wide.

The *touch down zone* of a runway is the initial area where arriving aircraft are supposed to make contact with the pavement. The *threshold* of a runway is the beginning of the touch down zone, and that is usually very near to the start of the hard surface. The line extending from the touch down zone out toward the approaching traffic and rising at the approach angle[13] is called the *glide path*. Often, the touch down zone is moved forward some distance. The effect of such a *displaced threshold* is to raise the glide path so that arriving aircraft clear high terrain. Thus, the *usable* arrival length of a runway may be less than its actual measured length. There are high buildings to the West and Southwest of Logan and a hill to the North and Northeast. To the South and Southeast of the airport is Boston's Outer Harbor. Four of the runways at Logan have their thresholds displaced.

Most of the runways at Logan are outfitted with runway lights along the edges, and "center line lights" inset flush with the pavement.[14] Runways 4L and 27 have "runway end indicator lights" (REIL), a pair of very bright Xenon flash lamps mounted on either side

[12]Such as Boeing 747, 767, and Lockheed L1011 types.

[13]The *approach angle* is the vertical angle at which aircraft approaching to land make their final decent. It averages three degrees.

[14]Only runway 15L/33R does not have center line lights.

of the threshold. Runways 15R, 22R, 27 and 33L have "visual approach slope indicators" (VASI), optical systems which give pilots approaching in good weather a visual indication of their vertical position with respect to the glide path.

Logan must operate in almost all weather conditions. Because it is located on an island in Boston Harbor, it not unusual for the ceiling to be low and the visibility restricted. Runways 4R, 22L, 15R, 33L, and 27 have facilities for operations during Instrument Meteorological Conditions. These facilities include "high intensity runway lights" (HIRL), localizer and glideslope transmitters, marker beacon systems, and, on runways 4R and 33L, "high intensity approach light systems" (HIALS). A localizer is a very high frequency radio transmission emitted from an array of antennae. The array creates an interference pattern which results in a narrow beam directed out along the extended runway center line. A glideslope is a similar transmitter system, using ground reflection, which creates a beam[15] designating the vertical component of the glide path. Marker beacons are similar systems which direct their transmissions straight up. As an approaching aircraft flies through the beacon's energy field, an onboard receiver turns on a panel light. If the pilot chooses, the receiver may make an audible tone in addition. "Outer marker" beacons are generally located close to the point where final decent is initiated. "Inner marker" beacons are usually located very close to the threshold of the runway. "Middle marker" beacons are located somewhere between outer and inner markers. The marker beacons act as indicators of the progress of the approach. Approach light systems are Xenon strobe lights mounted on pylons near the approach end of the runway. These lights flash in sequence to form a highly visible line of light indicating the position and direction of the runway threshold. These systems, when they are all active, allow the associated runways to be used in conditions down to a ceiling height of 200 feet and/or visibility down to one half nautical mile. Systems with this limit are called "Category II" instrument landing systems (ILS).

[15]Actually, *side lobes* exist in the transmission pattern. Pilots must take care not to attempt to follow a side lobe of the glideslope, which would almost certainly lead to disaster.

Runway 4R is equipped with even more special facilities. These include monitors for the localizer, glideslope, markers, and lighting systems, and a special radar which shows shipping traffic (which may have high masts) in the Boston Harbor Channel just to the East of the airport. These additional facilities permit specially outfitted aircraft with specially trained crews to use this runway during weather in which the ceiling may be as low as zero[16] and/or visibility along the runway may be as little as one quarter nautical mile. This runway qualifies as a "Category III-B" instrument landing system. There are severe restrictions on the rate that this system can accept aircraft under the limiting conditions described.

6.3 Transition from Enroute to Terminal Area

Enroute aircraft bound for Boston may be divided into three groups. The South group arrives from an area defined by a line along the eastern seaboard of the United States to a line from Boston southwest through Fort Worth, Texas. The West group arrives from an area extending from the west of the South group area around to the North to a line extending from Boston to the coast of Maine. Finally, the East group, which consists mostly of international traffic, comes from over the ocean to the east of the city.

The Boston terminal area is a "three post" arrival system. Entries to the terminal area are defined by three "gates" or *entry fixes*. The South group enters the terminal area from over the Providence VOR[17], which is located near Providence, RI; the West group enters the area from over Gardner VOR, near Gardner, MA; and the East group enters from Scupp intersection,[18] located approximately forty nautical miles to the East of the airport. These rules are not cast in concrete, and members of any group may be sent to one of the other entry points, particularly if traffic density is high at the place it would normally go.

[16]That is, the clouds come right down to the ground, as in fog.
[17]Variable Omnidirectional Range, a radio navigation aid.
[18]An intersection is an arbitrary point defined by specific signals from two or more VOR's.

Most of the time, aircraft are commanded to cross the entry fix at or below 10,000 feet and with speed reduced to below 250 knots.[19] The decent to this altitude is commanded by the enroute controllers, and may start as much as two hundred fifty miles away from the fix, depending on aircraft type and cruise altitude. After the aircraft arrive at the entry fix, control of them is turned over to "Boston Approach."

6.4 The TRACON

The Federal Aviation Agency (FAA) performs all control of aircraft on the runways and taxiways of the airport as well as in the surrounding airspace. The command center for this operation is the "Terminal Radar Approach Control" facility, or *TRACON*, in which eight to ten persons work at a time. Augmenting this facility is the control tower, which depends on visual contact with the aircraft and employs five to seven individuals at a time. These facilities are operated constantly, twenty four hours per day, every day.

The TRACON "positions," as they are called, consist of an area supervisor, an area manager, two approach controllers, one final approach controller, a "satellite" controller, and a weather and clearance position. The person working the weather and clearance position posts current and predicted weather, along with other information such as runways closed by MassPort for maintenance, on a large white board in the TRACON room. He or she also reads clearances to and checks "read backs" from aircraft about to depart. The "satellite" controller handles instrument approaches to any of the three so called "satellite" airports: Norwood, Bedford, and Beverly.

Tower personnel take over control responsibility of aircraft within five miles of the airport which are no more than 3000 feet above ground. Clearance to land or takeoff and last minute changes, such as a "go around" or side step to a parallel runway, are issued from the tower. All ground movements (for instance, taxi clearances) of aircraft and other vehicles on the runways or taxiways are also controlled by the tower.

[19]This is the speed limit for non-military aircraft below 10,000 feet.

The fundamental operating parameter of the airport is the primary arrival runway.[20] Once this has been established, the airspace around the airport is split by extending the runway center line of the primary arrival runway out to the periphery of the terminal area. Each of the approach controllers takes control of one of the resulting halves of the airspace.

The two approach controllers work as a team to "feed" aircraft at proper intervals to the final approach controller. Control of aircraft is turned over to the final approach controller at approximately the beginning of the turn onto glide path heading.

The area manager is a very experienced controller who *oversees the tactical coordination* of the whole process. The person in this position generally stands in the center of the radar room and watches and (perhaps more importantly) *listens* to the controllers while they do their job. He or she detects and arbitrates coordination problems between the controllers in the few instances that they occur, as well as noting when controllers are becoming fatigued and getting replacements for them when they need a break.

The area supervisor is usually the most experienced controller on duty in the facility. This person is the *strategic planner* for the operation. It is the area supervisor who, with input from the area manager and the tower manager, makes the decisions about which configurations will be used and when the changes to them will occur. He or she is also responsible for keeping written records of the overall activities at the TRACON, including configurations used, runway availability changes, and unusual circumstances like emergencies. When demand is expected to be light, the positions of area manager and area supervisor are sometimes combined, and one individual does both jobs.

6.5 Logan Configurations

While one could (see [FTA 82]) describe the various configurations as independent items having little in common, experience has taught that this is cumbersome. Instead,

[20] More about this in section 6.5 below.

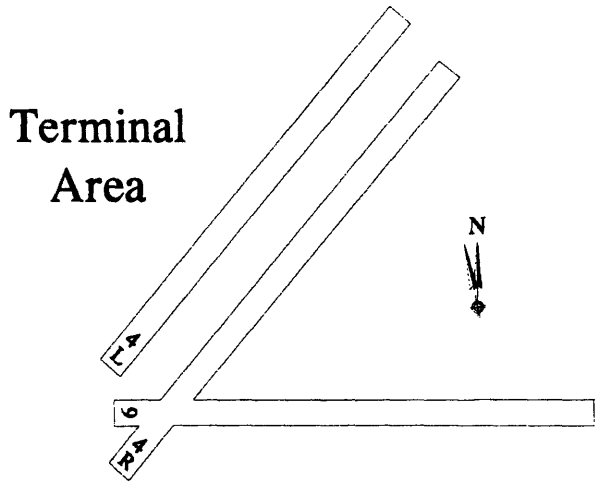


Figure 6.2: "Fours and Nine" Configuration

the system used by TRACON personnel to describe the configuration of the airport is both powerful and flexible.

There are six major configurations formed by combinations of runways, and five single runway configurations. Each of the six combination configurations has minor variations which will be described below.

The primary approach runway is *always* one of the runways that has an instrument landing system.[21] In general, *all* arriving commercial aircraft are given radar vectors to intercept and follow a localizer inbound. This usually takes place at least ten miles out from the airport.

6.5.1 Multiple Runway Configurations

The major combination configurations are named for the runways that compose them. They are:

Fours and nine (figure 6.2) uses runway 4R as the primary arrival runway, 9 as the primary departure runway, and 4L as the secondary arrival runway. All arriving aircraft are vectored to intercept and follow the runway 4R localizer. If visibility and ceiling permit,

[21] That is, localizer and glideslope transmitters.

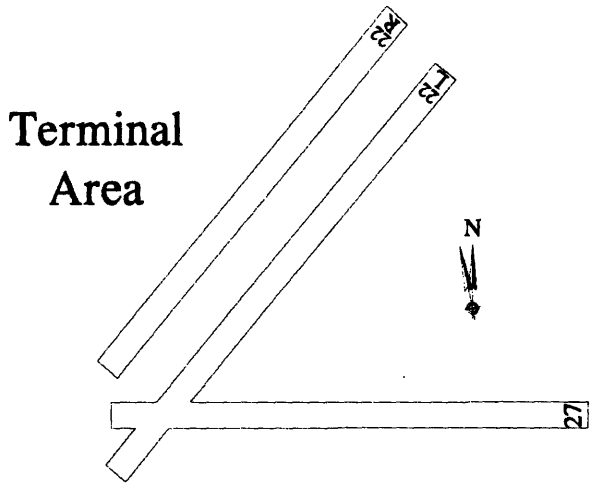


Figure 6.3: "Twenty Twos and Twenty Seven" Configuration

as the arriving queue of aircraft moves toward the airport, every other aircraft moves to the left and aligns to land on runway 4L once the runway can be visually distinguished on the airport. If the ceiling or visibility drops below Visual Meteorological Conditions, landings on 4L are prohibited and the arrival capacity drops. While runway 9 is the primary departure runway for this configuration, departures also take place from both 4L and 4R.

Twenty twos and twenty seven (figure 6.3) makes use of runway 22L for the primary arrival runway, 22R for the primary departure runway, and 27 for the secondary arrival runway. While 22L and 27 intersect, they have 6500 feet and 4700 feet to their intersection, respectively. When weather permits, pilots of aircraft destined to land on 22L are often asked to hold short of the intersection with 27. When this can be done, the two arrival runways can be operated asynchronously as two separate, non-interacting runways. This raises the capacity and lowers the controller work load. This configuration's arrival capacity is limited by the rate of departures on 22R, because all arriving aircraft must cross that runway in order to move to the terminal area.

Thirty threes and twenty seven (figure 6.4) uses 33L for the primary arrival runway, and 27 for the primary departure runway. Runway 33R is so short (2557 feet) that it

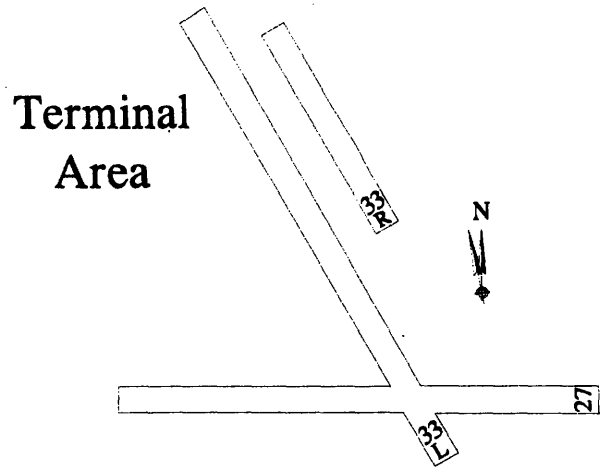


Figure 6.4: "Thirty Threes and Twenty Seven" Configuration

can only be used for small commuter aircraft which make up only about five percent of the total traffic. If departure demand rises beyond the capacity of using 27, or there are heavy category departures, then 33L is used for departures as well as arrivals.

Fours up and down (figure 6.5) uses runway 4R for the primary arrival runway. In Visual Meteorological Conditions, both runways 4R and 4L are used for arrivals and departures. This configuration is a scaled down version of the first configuration discussed in this section. When the arrival demand is high, the same technique of sidesteping aircraft from the 4R ILS to make a visual approach to 4L is employed. This configuration is used when the wind speed is too high to permit safe departures from runway 9, or when 9 is out of service.

Twenty twos and fifteen right (figure 6.6) employs runway 22L as the primary arrival runway, with 22R as the secondary arrival and primary departure runway. Runway 15R is used as the secondary departure runway, primarily for heavy international flights because it is nearer to the International Terminal and its over-water departure pattern is desirable from a noise abatement standpoint.

Fifteens and nine (figure 6.7) handles small aircraft on 15L and heavy category arrivals on the primary arrival runway, 15R. All other transport category arrivals are shared

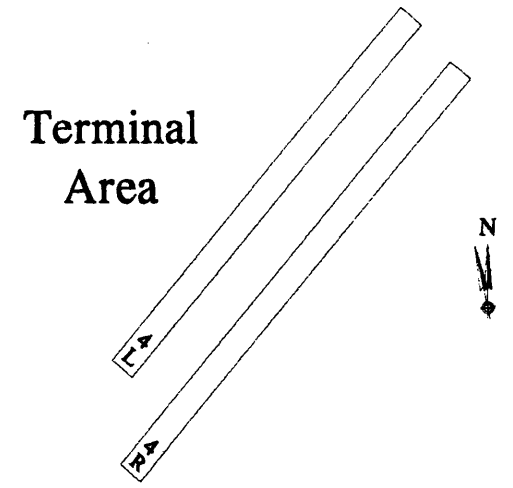


Figure 6.5: "Fours Up and Down" Configuration

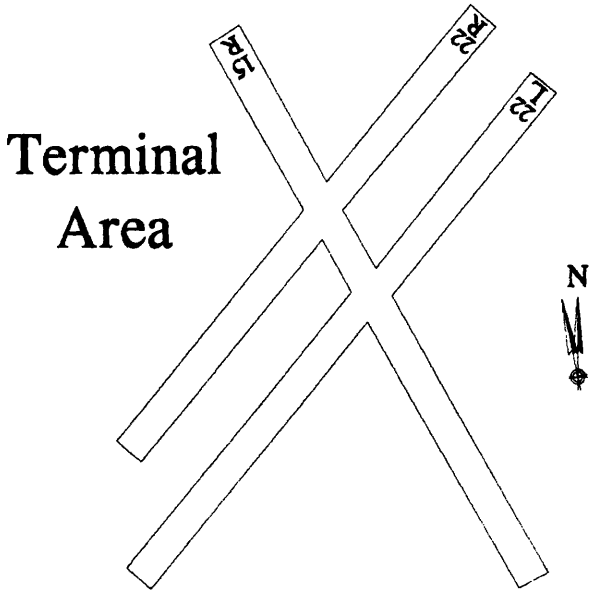


Figure 6.6: "Twenty Twos and Fifteen Right" Configuration

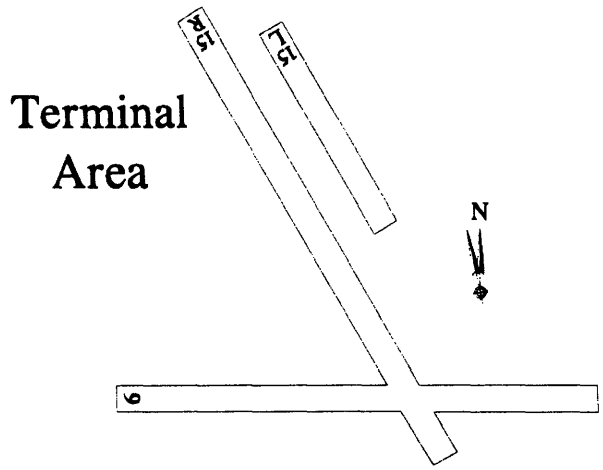


Figure 6.7: "Fifteens and Nine" Configuration

between 15R and 9. Departures are primarily from 15R, with 9 acting as secondary departure runway.

These configurations are used more as *guidelines* than enforced regulations. There is considerable freedom on the part of the TRACON to use any runway compatible with the current configuration for departures. If the wind and traffic load permit, for example, it is desirable for international flights to depart from runway 15R, because the arrival end of that runway is near the international terminal, and the runway is long and can be used by the typically heavy aircraft leaving on international flights. If the wind is coming from the East at ten knots, for example, and the "fours and nine" configuration is in use, a request by an international carrier for a 15R departure may be permitted if demand is not prohibitively high. A "hole" is made in the schedule of operations on the other runways to allow the takeoff on 15R to occur. For the same reasons, landings of international flights on 33L are often desirable and exceptions for them are made when possible. However, it is much harder to coordinate such maverick arrivals than it is for departures, so they are much less common.

6.5.2 Single Runway Configurations

The single runway configurations are made up of those runways that are equipped with instrument landing systems.[22] Single runway configurations are only employed when the wind speed is too high to permit crosswind landings or takeoffs on alternate runways, or when a runway is out of service (see below). The five runways which have instrument landing systems are 4R, 15R, 22L, 27, and 33L.

6.5.3 Rule System Representation of Logan

All of the physical characteristics of Logan must be represented as facts and history cells. For example, the information describing Runway 4R is:

```
(tell tc logan runways - bos-4r :additional t)
(tell tc bos-4r allocation - available)
(tell tc bos-4r owned-by - logan)
(tell tc bos-4r op-status - op)
(tell tc bos-4r runway-direction - 35)
(tell tc bos-4r depart-length - 10005)
(tell tc bos-4r arrive-length - 8840)
(tell tc bos-4r hirl - op)
(tell tc bos-4r loc - op)
(tell tc bos-4r gs - op)
(tell tc bos-4r om - op)
(tell tc bos-4r mm - op)
(tell tc bos-4r im - op)
(tell tc bos-4r cll - op)
(tell tc bos-4r tdzl - op)
```

[22]The reader may wonder why all the runways are not outfitted with instrument landing systems. There are *two* reasons. First, the parallel runways at Logan are too close to each other to allow simultaneous instrument approaches. Second, instrument landing systems are *expensive*, costing about one million dollars per runway end.

```
(tell tc bos-4r rvr - op)
```

The "-" appearing in the time interval position indicates "always" (see page 76). These statements inform the temporal database manager that one of Logan's runways is Runway 4R and that initially the runway is available for use. They also indicate the arrival and departure lengths [23] and direction of the runway, and the assumed operational status of the various special equipment available to this runway. Runway 4R is equipped with high intensity runway lighting (`hirl`), a standard category instrument landing system consisting of localizer (`loc`) and glide slope (`gs`) transmitters, outer, middle, and inner marker beacon transmitters (`om`, `mm`, and `im`), center line lights (`cll`), touch down zone lights (`tdzl`), and a runway visual range (`rvr`) system. All the equipment is assumed to be initially operational.

Each of Logan's nine other runways has a similar set of facts associated with it. Also, there is additional analogous information for each of the "intersection departure" and "hold short" arrival runway segments.

The configurations must be similarly defined. For example, the "fours and nine" configuration (see Figure 6.2) is specified by:

```
(tell tc logan configurations - bos-con-449v :additional t)
(tell tc bos-config-449v wx-cat - vfr1)
(tell tc bos-config-449v max-capacity - 84)
(tell tc bos-config-449v primary-arrival-runway - bos-4r)
(tell tc bos-config-449v primary-departure-runway - bos-9)
(tell tc bos-config-449v arrive-runways - bos-4r)
(tell tc bos-config-449v arrive-runways - bos-4l :additional t)
(tell tc bos-config-449v depart-runways - bos-9)
(tell tc bos-config-449v depart-runways - bos-4r :additional t)
(tell tc bos-config-449v depart-runways - bos-4l :additional t)
```

[23] the difference indicates a "displaced threshold"

Like the runways above, there are several more sets of entries like this one which describe the other configurations.

In addition to these definitions of the physical characteristics of the runway systems, other critical information must be supplied. For example, if the system is to make evaluations concerning snow removal, knowledge about snow brush and plow vehicles will be required. This information can be provided either by specifying the number of vehicles of each type that will be initially available, or by tracking the individual history of each vehicle if a greater level of detailed vehicle scheduling is desired. The forms:

```
(tell tc logan snow-plows - snow-plow-1 :additional t)
(tell tc snow-plow-1 use-status - available)
(tell tc snow-plow-1 crew-status - (not available)
(tell tc logan snow-plows - snow-plow-2 :additional t)
(tell tc snow-plow-2 use-status - available)
(tell tc snow-plow-2 crew-status - (not available)
```

indicate that two of the plow vehicles at Logan are in working order and available, but that the crews to operate them will have to be summoned before they can be put into use.

6.6 Configuration Selection Rules

The primary rules for determining what configuration should be used during which period depends on the weather. During VFR operations when there is little wind, particularly at night, noise abatement rules will be the most important. If demand is high during VFR with low wind speed, capacity considerations will be paramount. As the wind speed exceeds ten knots, or the ceiling descends, the selection of usable runways is reduced, thus reducing the choices of available configurations from which to choose. In the limit of wind speeds above thirty knots or ceilings below 600 feet, the choice of configuration may be forced by lack of any alternative.

6.6.1 Weather Rules

The most important factor in determining the primary arrival runway is the *weather*. Parameters describing the weather include:

Wind Speed and Direction: It is desirable for aircraft to land into the wind. The energy dissipated by an aircraft during a landing is proportional to the *square* of its speed *relative to the ground*. Stopping distance and wear on landing gear and engines depends linearly on the energy. The higher the landing speed of an aircraft, the more important even a few knots of extra ground speed becomes. While federal regulations actually permit landings to occur with up to ten knots of tailwind,[24] use of runways having a tailwind component is generally avoided for any extended period. When practical, the primary arrival runway is chosen to be the runway with the greatest headwind component.

There is a maximum demonstrated crosswind[25] in which it is safe to land an aircraft. The maximum demonstrated crosswind varies with aircraft type and must be demonstrated during the certification testing of each type. In general, however, transport category aircraft are required to be able to land with up to twenty knots of crosswind. Runways which have a crosswind component greater than twenty knots may not be used for landing. The requirements for departures are less stringent.

As wind speed increases, the set of runways which meet the maximum crosswind requirement diminishes, until only one or a parallel pair can be used. In fact, if the wind were to come from 63 degrees at greater than 42 knots, *none of the runways at Logan* would meet the crosswind criterion, and the airport would have to close! Fortunately, the location of the airport makes this an unlikely event.

Ceiling: The ceiling is defined as the height above the ground of the lowest layer of clouds which cover more than 49 percent of the visible sky and are not designated

[24]Tailwind is the component of the wind velocity coming from the direction of the approach end of a runway. Headwind is the component coming from the direction of the departure end of a runway. Headwind is considered as positive, tailwind as negative.

[25]Crosswind is the component of the wind velocity coming perpendicular to a runway.

as "thin." When the ceiling is greater than 2500 feet over an airport, the airport is operating in good Visual Meteorological Conditions (GVMC). When the ceiling is less than 2500 feet, but greater than 1000 feet, the airport is operating in Visual Meteorological Conditions (VMC). When the ceiling is less than 1000 feet, the airport operates in Instrument Meteorological Conditions (IMC). Ceilings between 1000 and 600 feet define Category I IMC. Category II IMC is in effect for ceilings from 600 feet down to 200 feet. Category III-A allows the ceiling to descend to 100 feet, and in category III-B the ceiling may come all the way down to the ground.

Ceiling height can be difficult to measure. Logan is equipped with a "ceilometer", an optical device which reflects a light beam from the cloud deck and accurately measures its height by triangulation. However, sometimes the bases of the clouds are less defined. This can be true due to turbulence or the kind of slow lapse rate found in some warm fronts. In these cases the ceiling is estimated by tower or weather service personnel. If there is dense fog or haze, the sky may be "obscured," and the ceiling simply undefined.

Each weather category defined above has a corresponding set of operating rules and limitations for the airport. Since these operational procedures also depend on visibility, discussion of them will be deferred until the end of the following section.

Visibility: The *prevailing visibility* at an airport "is the greatest distance objects can be seen and identified through at least 180 degrees of the horizon."[26] A more precise definition, perhaps in terms of optical energy loss per unit distance, might be preferable, but this is the definition currently in use.

All of the weather conditions mentioned above are also defined in terms of minimum required visibility. Good Visual Meteorological Conditions require visibility in excess of five miles. Visual Meteorological Conditions require at least three miles visibility. Visibility below three miles is considered Instrument Meteorological Conditions. Category II IMC requires visibility of more than one half mile. Category III-A requires at least one

[26]This definition is from [FAA 85].

quarter mile, and category III-B requires a minimum of one eighth of a mile visibility.

Each meteorological condition has an associated set of rules and requirements for the operation of airports and aircraft.[27] For instance, the rules for operations in Visual Meteorological Conditions are called the Visual Flight Rules (VFR), and those for Instrument Meteorological Conditions are called Instrument Flight Rules (IFR). As the ceiling and visibility are reduced into each succeeding meteorological regime, the requirements become more and more restrictive.

Most private pilots are *not* certified for flight in IMC, but all commercial transport pilots are certified for flight in conditions down to Category II IMC. This might indicate that demand at the airport would be reduced when weather conditions deteriorate to IMC. However, private pilots fly a very small percentage of the operations at Logan, so this effect on demand is small. On the other hand, few crews or aircraft are certificated for operations in Category III conditions, so demand does fall when the weather is that bad.

Runways without operating instrument landing systems (ILS) may not be used for arrivals during IMC. This means that, for example, during IMC the "fours and nine" three runway combination configuration (which has two arrival runways) must be reduced to one arrival runway (4R) because 4L (which has no instrument approach) is not usable for landings. In addition, aircraft approach speed may be increased during operations in IMC to facilitate a possible missed approach procedure. The increased speed requires a longer stopping distance and increases runway occupancy time. This requires greater separation between consecutive aircraft and lowers overall capacity. Finally, holdshort operations of intersecting runways, as in the "twenty twos and twenty seven" configuration, are prohibited in IMC. While this restriction allows the configuration to be used, operations on the intersecting runways can no longer be simultaneous. This excludes asynchronous operations, too, since simultaneous operation might occur if the operations are temporally

[27] For details about these rules, the reader is directed to [FAR].

independent.

6.6.2 Runway Availability

Equipment malfunctions degrade the utility of runways. For instance, failure of almost any equipment associated with runway 4R make it unusable during Category III IMC because the loss of equipment requires an increase in the required minimum ceiling. Loss of the use of the glideslope transmission of an ILS causes an increase in the minimum ceiling required for the use of the associated approach. Failure of a marker beacon has a similar effect.

Runways and their equipment require maintenance. Lighting and radio frequency apparatus need periodic repairs and adjustments, as well as emergency repairs when they fail. In the summer, the grass around the runways must be cut once a month or so. Repainting numbers, center lines, and position indicators requires longer periods. Every time an aircraft lands, some of the tire surface melts (!) and sticks to the asphalt in the touch down zone. The resulting buildup is slippery and must be regularly removed. The entire asphalt surface has to be replaced every few years.

On rare occasions, disabled aircraft occupy a runway. When this happens, the runway must be removed from service until the situation can be corrected. Repair or removing aircraft with minor problems such as flat tires or disabled steering may take a few minutes to a few hours. In the very uncommon event of a crash, a runway may be closed for weeks or even months.

Finally, solid precipitation in the form of freezing rain, sleet, snow, or ice pellets create accumulations on the runways in the winter time. Ice and snow on a runway effects braking action and controllability. Braking action can be measured with specially equipped automobiles or reported by the crews of landing aircraft. Any runway with a measured braking action of "nil" or braking action reported to be "nil" by three aircraft must be removed from service. Runways can be chemically pre-treated with a variety

of substances to cause the frozen water to melt and drain if the quantity of accretion is predicted to be light. Light build-up of snow may be brushed from runways, but heavier accumulations must be plowed. These options are ordered by the length of time they require, from shortest to longest.

All of these conditions necessitate moving men, vehicles, and perhaps heavy equipment onto the runways, making them unavailable while the operations are in progress. Whenever possible, the down time should be scheduled in advance to coincide with a period when the runway to be taken out of service is not part of a desirable configuration.

Currently this kind of advanced scheduling is not generally employed. As noted above, while the FAA directs traffic of all kinds on the runways and taxiways, MassPort owns and operates the airport. All of the maintenance activities described above, with the single exception of repair or adjustment of the ILS equipment, is the responsibility of MassPort. With respect to longer term operations such as painting or re-surfacing, the TRACON is generally informed of the plans well in advance. However, the Port Authority plans and schedules shorter activities, such as lighting maintenance, with little or no communication or regard for FAA operations. The TRACON is simply informed, in the worst case with as little as ten minutes warning, that a runway will be out of service for a period of time. For *any* form of planning and scheduling to work at Logan, communication between these groups must improve, and comprehensive planning and scheduling of configurations and facilities maintenance must be coordinated.

6.6.3 Dwell

There is an interesting dichotomy in the population of the Boston area. On the one hand, there is a large segment of the people who want high levels of commerce. This attitude indicates a tolerance for the less desirable side effects that an airport produces, such as noise pollution and heavy automotive traffic. On the other hand, there is a smaller, but *much* more vociferous group which demands a "clean" environment.

Whenever possible, the configurations are changed in order to spread the noise created by aircraft evenly (weighted by population density) over the populations under the terminal control area. The complex paradigm used to attempt this balancing act is called the "Preferential Runway Assignment System" (see [Eldred 82]), or "PRAS." The contiguous period that a particular configuration is in use is called "dwell." PRAS demands that after a dwell of, say three hours, if there are alternative configurations which are acceptable according to the weather and availablity criteria set forth above that would result in a change in the population receiving the noise from the operation of the airport, then a change to one of these alternatives should be made.

Longer term (up to a year) usage data are used to create averages which are utilized to correct imbalances in the noise profile caused by periods when the weather does not allow changes for noise equality to be made.

Any automated system which is to be a realistic benefit to the TRACON must take the PRAS paradigm into account. In terms of the *Tower Chief* effort, this means that, at a minimum, there must be rules which "turn" the airport at intervals of a few hours if other parameters such as wind, ceiling, and visibility permit.

6.6.4 Shift Changes and Workload

The job of controlling aircraft involves creating tactical plans over a period lasting ten to fifteen minutes. This means that a controller cannot be replaced by a colleague who simply arrives, sits down, and takes over. Instead, a replacement controller requires a period of time ranging from one to ten minutes (depending on the traffic load) to observe and understand the tactical situation and process devised by the person he or she is replacing. While position substitutions take place at various times for rest breaks and meals, they are more or less random. However, during the periods around personnel shift changes, when a whole new crew comes on duty, there are always many of these events.

Whenever a configuration change occurs, the controllers must decide which aircraft in

each of the arrival and departure queues will be the last one to use the old configuration. Once this has been determined, the controllers must begin to set up the tactical situation for the new configuration while they continue to handle the execution of completing operations for the old one. The task may be relatively quick and easy or may take as long as twenty minutes and involve complicated maneuvers. In most cases, a configuration change requires increased concentration which adds to the controllers' workload and stress level.

Configuration changes should not be scheduled near or during personnel shift change times. Furthermore, given the time necessary to accomplish a configuration change and the associated increase in controller workload, no more than two changes should be scheduled per hour unless weather or other safety requirements demand more.

6.7 An Initial Rule Set

A subset of the rules used for scheduling configurations is presented below. The set exhibits most of the techniques used to model the processes described earlier in this chapter to select configurations.

The first set of rules determines the category of weather that the airport must operate under at any given time.

```
(defrule gvfr tc ((?airport visibility ?x)
(?airport ceiling ?y))
((and (>= ?x 5.0)
(>= ?y 2500.0)))
(rs-assert ?airport 'op-status 'vfr1 tc))

(defrule vfra tc ((?airport visibility ?x)
(?airport ceiling ?y))
((and (< ?x 5.0) (>= ?x 3.0)
(>= ?y 1000.0)))
(rs-assert ?airport 'op-status 'vfr2 tc))

(defrule vfrb tc ((?airport visibility ?x)
```

```
(?airport ceiling ?y))
((and (< ?y 2500.0) (>= ?y 1000.0)
(>= ?x 3.0)))
(rs-assert ?airport 'op-status 'vfr2 tc))

(defrule ifr1a tc ((?airport visibility ?x)
(?airport ceiling ?y))
((and (< ?x 3.0) (>= ?x 1.0)
(>= ?y 600.0)))
(rs-assert ?airport 'op-status 'ifr1 tc))

(defrule ifr1b tc ((?airport visibility ?x)
(?airport ceiling ?y))
((and (< ?y 1000.0) (>= ?y 600.0)
(>= ?x 1.0)))
(rs-assert ?airport 'op-status 'ifr1 tc))

(defrule ifr2a tc ((?airport visibility ?x)
(?airport ceiling ?y))
((and (< ?x 1.0) (>= ?x 0.5)
(>= ?y 200.0)))
(rs-assert ?airport 'op-status 'ifr2 tc))

(defrule ifr2b tc ((?airport visibility ?x)
(?airport ceiling ?y))
((and (< ?y 600.0) (>= ?y 200.0)
(>= ?x 0.5)))
(rs-assert ?airport 'op-status 'ifr2 tc))

(defrule ifr3a tc ((?airport visibility ?x)
(?airport ceiling ?y))
((and (< ?x 0.5) (>= ?x 0.25)
(>= ?y 100.0)))
(rs-assert ?airport 'op-status 'ifr3a tc))

(defrule ifr3b tc ((?airport visibility ?x)
(?airport ceiling ?y))
((and (< ?y 200.0) (>= ?y 100.0)
(>= ?x 0.25)))
(rs-assert ?airport 'op-status 'ifr3a tc))
```

```
(defrule ifr3b1 tc ((?airport visibility ?x))
((<= ?x 0.25))
(rs-assert ?airport 'op-status 'closed tc))

(defrule ifr3b2 tc ((?airport ceiling ?x))
((<= ?x 100.0))
(rs-assert ?airport 'op-status 'closed tc))
```

The next rule determines the headwind and crosswind component for every runway at an airport when the wind speed and direction at the airport are known. Note that the consequents of this rule "calls" a LISP function to actually compute the components.

```
(defrule chw1 tc
((?runway runway-direction ?rd nt) ; for some runway
(?runway op-status op nt) ; which is operational
(?runway owned-by ?airport nt)       ; at this airport
(?airport wind-direction ?wd) ; and there is known wind
(?airport wind-speed ?ws)) ; dir and speed
((rs-assert ?runway 'crosswind (crosswind ?wd ?ws ?rd) tc)
(rs-assert ?runway 'headwind (headwind ?wd ?ws ?rd) tc)))
```

The LISP functions that are used in the consequent are:

```
(defun headwind (wind-dir wind-speed runway-dir)
(* wind-speed (cos (deg-rad (- runway-dir wind-dir)))))

(defun crosswind (wind-dir wind-speed runway-dir)
(abs (* wind-speed (sin (deg-rad (- runway-dir wind-dir))))))
```

and the function **deg-rad** changes degrees into radians.

The following rule is universally quantified. It creates the subset of all the runways at an airport that are feasible for use during IMC. The rule requires that the crosswind and headwind components of the wind must be less than the limits set forth in section 6.6.1, and that the runways have operating instrument landing systems. Of course, this

is not really complete, but the addition of the other equipment requirements is straight forward.

```
(defrule ry2 tc ; this one is for IFR
((?airport is-a airport)
(?airport op-status ?status)
(?runway owned-by ?airport)
(?runway headwind ?hw nt)
(?runway crosswind ?cw)
(?runway loc op)
(?runway gs op))
((and (not (eq ?status 'vfr1))
(not (eq ?status 'vfr2))
(> ?hw -10.0)
(< ?cw 22)))
(rs-assert ?airport 'feasible-runways ?runway tc
:additional t))
```

The rule below is a preference rule, which chooses the primary arrival runway from those that make up the feasible set defined by the previous rule.

```
(defrule choose-arrival tc ; for a runway which has it's
((?runway headwind ?nhw nt) ; winds updated
(?runway crosswind ?ncw nt)
(?runway allocation available nt) ; no double use!
(?airport feasible-runways ?runway) ; acceptable candidate?
(?airport arrival-runway-1 ?arway nt) ; should be arrival
(?runway not-same ?arway nt) ; shouldn't compare same ones
(?arway headwind ?ohw nt) ; headwind and crosswind are
(?arway crosswind ?ocw nt)) ; already known
((and (> ?nhw -10.0) ; longitudinal wind acceptable
(> ?nhw ?ohw) ; and better than that past
(< ?ncw ?ocw))) ; and the new crosswind is less
((rs-assert ?airport 'arrival-runway-1 ?runway tc) ; use it!
(rs-assert ?runway 'allocation 'used tc) ; and remember we have
(rs-assert ?arway 'allocation 'available tc)))
; and the the old one is usable
```

As usual, a preference rule must be "seeded" with an initial choice for comparison. The default rule that seeds this rule is:

```
(def-default arrival-default tc
((?airport arrival-runway-1 ?arway)
(?runway owned-by ?airport))
((rs-assert ?runway 'allocation 'used tc)
(rs-assert ?airport 'arrival-runway-1 ?runway tc)))
```

The rule below uses the primary arrival and departure runways chosen by rules like the one above together with the predicted demand to choose a configuration.

```
(defrule select-config tc
((?airport arrival-runway-1 ?arway)
(?airport departure-runway-1 ?depway nt)
(?airport demand ?demand)
(?airport configurations ?config nt)
(?config prime-arway ?arway nt)
(?config prime-depway ?depway nt)
(?config capacity ?capacity nt))
((> ?capacity ?demand))
(rs-assert ?airport configuration ?config tc))
```

The set of rules presented here shows the expressiveness and power of a rule based inferrence system extended to embrace temporally constrained knowledge. The system can, given forecast data about wind and ceiling over some period of time, generate a simple "schedule" of runway configurations. It does not, however, have the ability to make a correct schedule with regard to negative interactions between configurations. If, for example, the wind should change rapidly from a southeasterly to a northwesterly direction, the system described thus far will suggest using Runway 15R prior to the change, and Runway 33L after it has occurred. While this certainly *is* a possible configuration switch, it requires some preparation before the actual switch is made. For example, about fifteen to twenty minutes **before** the proposed change, aircraft arriving in the terminal control area should not be vectored to the approach queue for 15R. Instead, they should be

given vectors to the queue for 33L. Additionally, departing aircraft should be given taxi instructions to 33L (or whatever will be used as the primary departure runway) starting about ten minutes prior to the configuration change. The kind of reasoning involved in generating these decisions cannot readily be described in terms of the rule based system described in Chapter 4. Instead, they are generated by the scheduling system which was presented in Chapter 5.

6.8 Scheduling Planned Sequences of Operations

Decisions concerning what *should* happen under a given set of conditions is still the responsibility of the Temporal System Analyzer. The TSA makes the decision that a specified configuration should be employed for some period of time. However, the specification of what operations the supervisor should use and when they should be executed in order to accomplish this goal are defined by a schedule. Thus, the last rule presented in the previous section needs to have its consequent modified to a request to the scheduler to create a schedule of events that will result in the consequent being asserted. The consequent, which used to read:

```
(rs-assert ?airport configuration ?config tc)
```

should actually read:

```
(schedule ?airport configuration ?config tc)
```

The printed representation of an initial action to accomplish this goal is:

```
name:    CHANGE-CONFIG
context: ((?CONFIG PRIMARY-ARRIVAL-RUNWAY ?ARWAY-1)
(?CONFIG PRIMARY-DEPARTURE-RUNWAY ?DEPWAY))
tests:   NIL
reqs:    ((?ARWAY SNOW-DEPTH 0)
```

```
(?DEPWAY SNOW-DEPTH 0)
(?AIRPORT ARRIVAL-QUEUE ?ARWAY-1)
(?AIRPORT DEPARTURE-QUEUE ?DEPWAY-1))
while:   NIL
proc:    NIL
results: ((RS-ASSERT ?AIRPORT CONFIGURATION ?CONFIG TC
:TYPE PERSISTENT))
time:    0
tf:      NIL
```

This action is called **CHANGE-CONFIG**. The variables **?AIRPORT** and **?CONFIG** will be determined at the time the action is selected. The primary and secondary runway identities are necessary to make sure that those runways are clear of snow and to change the queuing points in the air and on the ground. The queuing point on the ground is the end of the taxiway nearest the departure end of the primary departure runway, while the queuing point in the air is usually the final approach fix for the primary arrival runway. The values for these variables will be determined by standard searching and pattern matching in the temporal database.

Notice that the time required for this action is zero. This means that, once all the preconditions are satisfied, the result is to be immediately asserted. The scheduler next attempts to see if the precondition patterns (represented in the **reqs** slot of the action) are satisfied at the time the **CHANGE-CONFIG** action is scheduled to be executed. Each one that is not satisfied becomes a new goal which must be scheduled.

The goal to change the queuing point for the departure runway can be satisfied by the following action:

```
name:    SET-AIRPORT-DEPARTURE-QUEUE
context: nil
tests:   NIL
```

```
reqs:    NIL
while:   NIL
proc:    ((OPERATOR-REQUEST
"Route gate departures to runway ~a"
?RUNWAY))
results: ((RS-ASSERT ?AIRPORT DEPARTURE-QUEUE ?RUNWAY TC
:TYPE PERSISTENT))
time:    5
tf:      NIL
```

The operator request will be scheduled to occur five minutes before the scheduled time for the configuration change. The action to change the arrival queuing point looks very similar, except that the time is ten minutes instead of five.

Should there be snow on one or both of the runways, clearing it must be accomplished before the change can be asserted. Two candidate actions are available for clearing snow, one to brush the snow off of the runway, the other to use plows. If the snow is deeper than two inches, it must be plowed. If it is less, brushing is preferred since it is faster.

```
name:    BRUSH-RUNWAY
context: ((?RUNWAY SNOW-DEPTH ?DEPTH)
(?RUNWAY DEPARTURE-LENGTH ?L)
(LOGAN AVAILABLE-SNOW-BRUSHES ?BRUSHES))
tests:   ((<= ?DEPTH 2.0))
reqs:    ((?BRUSHES CREW-STATUS AVAILABLE))
while:   ((RS-ASSERT ?RUNWAY STATUS CLOSED TC)
(RS-ASSERT ?BRUSHES STATUS IN-USE TC))
proc:    ((OPERATOR-REQUEST "Close ~a and commence brushing."
?RUNWAY)) results: ((RS-ASSERT ?RUNWAY SNOW-DEPTH 0 TC
:TYPE PERSISTENT))
```

```
time:     45
tf:       (/ (* ?L 150) (* (NUMBER-OF ?BRUSHES) 35200))
```

The **BRUSH-RUNWAY** action can only be used if the depth of snow accumulation is not greater than 2 inches and there are available brushes to apply to the task. While it is in progress, the runway has to be closed and the brushes are not available for other duty. The default, worst case time necessary to carry out this action is 45 minutes. If enough information is available, a more accurate time requirement can be computed. The **PLOW-RUNWAY** action is, of course, quite similar:

```
name:     PLOW-RUNWAY
context: ((?RUNWAY SNOW-DEPTH ?DEPTH)
(?RUNWAY DEPARTURE-LENGTH ?L)
(LOGAN AVAILABLE-SNOW-PLOWS ?PLOWS))
tests:    ((> ?DEPTH 2.0))
reqs:     ((?PLOWS CREW-STATUS AVAILABLE))
while:    ((RS-ASSERT ?RUNWAY STATUS CLOSED TC)
(RS-ASSERT ?PLOWS STATUS IN-USE TC))
proc:     ((OPERATOR-REQUEST "Close ~a and commence plowing."
?RUNWAY)) results: ((RS-ASSERT ?RUNWAY SNOW-DEPTH 0 TC
:TYPE PERSISTENT))
time:     60
tf:       (/ (* ?L 150) (* (NUMBER-OF ?BRUSHES) 26400))
```

The worst case is based on the assumption that only one of the vehicles is available. If, for example, the accumulation is small enough to allow brushing and there are two brushes available, the request to the operator to close the runway and begin brushing will come about 22 minutes before the configuration change is scheduled. If the crews are not available, the following action will be scheduled prior to the **BRUSH-RUNWAY** action:

```
name:     CALL-BRUSH-CREW
context: NIL \\ \verbtests: NIL
reqs:     NIL \\ \verbwhile: NIL
proc:     ((OPERATOR-REQUEST "Please alert the brush crews.")
results: ((RS-ASSERT ?BRUSHES CREW-STATUS AVAILABLE TC
:TYPE PERSISTENT))
time:     60
tf:       NIL
```

This will cause the supervisor to get a message suggesting that the brush crews be alerted one hour prior to the scheduled starting time for the brushing operation.

Of course, if there were no forecast of snow accumulation for the airport, none of the snow removal plan would be scheduled. Further, if the forecast should change from an expectation of *not* having snow to one requiring snow removal, the schedule for brushing or plowing can be added to the existing schedule *provided* that there is enough time left from now to the scheduled configuration change to accomplish it. If there is not enough time, the scheduler must report back to the temporal system analyzer that its request cannot be honored. It is then up to the TSA's rules to decide what to attempt instead.

Note that the rules and actions presented in the last two sections are a representative *sample* of the rules necessary to actually run the case study. There are additional rules covering areas such as noise abatement, demand, and the dwell time of the use of any runway configuration, for example.

6.9 Results

The scheduling system has been informally tested by having it schedule configurations for a twenty three hour period using a small set of thirty rules and half a dozen actions. The data for the period approximates passage of a common winter storm caused by a low pressure center and associated cold front. Typically, such low pressure centers

travel Northeast on a line South and East of Boston, bringing low IFR conditions and precipitation. The chain of events in the example have been accelerated somewhat, but it is otherwise realistic. The ceiling data for the period is:

```
(tell tc logan ceiling ((last-midnight) 3) (10000))
(tell tc logan ceiling (3 8) (8000))
(tell tc logan ceiling (8 10) (7000))
(tell tc logan ceiling (10 11) (5500))
(tell tc logan ceiling (11 12) (3500))
(tell tc logan ceiling (12 1330) (2500))
(tell tc logan ceiling (1330 14) (1800))
(tell tc logan ceiling (14 15) (700))
(tell tc logan ceiling (15 1630) (1100))
(tell tc logan ceiling (1630 1715) (1300))
(tell tc logan ceiling (1715 18) (1000))
(tell tc logan ceiling (18 19) (600))
(tell tc logan ceiling (19 20) (300))
(tell tc logan ceiling (20 21) (1000))
(tell tc logan ceiling (21 23) (1200))
```

while the visibilities are:

```
(tell tc logan visibility ((last-midnight) 3) (35))
(tell tc logan visibility (3 5) (20))
(tell tc logan visibility (5 8) (15))
(tell tc logan visibility (8 9) (9))
(tell tc logan visibility (9 10) (7))
(tell tc logan visibility (10 11) (8.5))
(tell tc logan visibility (11 12) (7))
```

```
(tell tc logan visibility (12 13) (5))
(tell tc logan visibility (13 14) (3.5))
(tell tc logan visibility (14 15) (1))
(tell tc logan visibility (15 16) (3.1))
(tell tc logan visibility (16 17) (3.5))
(tell tc logan visibility (17 18) (2))
(tell tc logan visibility (18 19) (0.5))
(tell tc logan visibility (19 20) (0.3))
(tell tc logan visibility (20 22) (3))
(tell tc logan visibility (22 23) (5))
```

The wind direction and speed information is as follows:

```
(tell tc logan wind-direction ((last-midnight) 4) (180))
(tell tc logan wind-direction (4 5) (170))
(tell tc logan wind-direction (5 6) (160))
(tell tc logan wind-direction (6 9) (140))
(tell tc logan wind-direction (9 10) (130))
(tell tc logan wind-direction (10 11) (120))
(tell tc logan wind-direction (11 14) (100))
(tell tc logan wind-direction (14 16) (90))
(tell tc logan wind-direction (16 17) (75))
(tell tc logan wind-direction (17 18) (50))
(tell tc logan wind-direction (18 19) (40))
(tell tc logan wind-direction (19 20) (35))
(tell tc logan wind-direction (20 21) (10))
(tell tc logan wind-direction (21 22) (0))
(tell tc logan wind-direction (22 23) (345))
```

```
(tell tc logan wind-speed ((last-midnight) 2) (8))
(tell tc logan wind-speed (2 5) (10))
(tell tc logan wind-speed (5 9) (12))
(tell tc logan wind-speed (9 10) (13))
(tell tc logan wind-speed (10 16) (8))
(tell tc logan wind-speed (16 17) (15))
(tell tc logan wind-speed (17 18) (18))
(tell tc logan wind-speed (18 19) (23))
(tell tc logan wind-speed (19 21) (20))
(tell tc logan wind-speed (21 22) (15))
(tell tc logan wind-speed (22 23) (12))
```

The resulting runway configuration schedule created by the scheduler is:

Time	Arrival	Departure	Reason
00:00 – 02:00	33L	15R	Noise Abatement
02:00 – 06:00	15R	15R	Increased Wind, Noise Abatement Low Demand
06:00 – 12:00	15R	9	Increased Demand
12:00 – 17:00	4R/4L	9	Wind Direction, Noise Abatement
17:00 – 20:00	4R	4L	Low IFR
20:00 – 21:00	4R/4L	4L	Marginal VFR
21:00 – 22:00	33L	4L/4R	Changing Wind
22:00 – 23:00	33L	27	Changing Wind

The first entry in the schedule is the result of a rule which states that during the late night and early morning hours when demand is very low, if the tail-wind component is below 10 knots and the runway is clear and dry, then runway 33L should be used for arrivals and runway 15R should be used for departures. This is a noise abatement rule which keeps all activity of the airport over water. During the 2:00 A.M. to 6:00 A.M. period, the tail-wind component is too great to continue to use 33L for arrivals. The demand is still low enough during these hours to permit single runway operation and

the wind direction makes arrivals on 15R the most appealing. This particular choice for the arrival runway is the best for the wind component, but might not be best from an environmental point of view. It would not be difficult to change the rules so that another runway, such as 4R, could be used instead. However, later in the day, the weather associated with the storm passage forces the use of 4R for an extended period, so the choice of 15R is justified. Continuing the use of runway 15R for departures is good from both the standpoint of noise abatement and wind direction. After 6:00 A.M., demand begins to increase and a configuration with a higher capacity is chosen to accomodate it.

The choice of time to change to the 4R/4L and 9 configuration depends on demand and dwell, as well as wind direction. The capacity of this configuration is higher than that of the 15R and 9 configuration, particularly for departures. If the demand were higher than the capacity of the 15R and 9 configuration, the change to 4R/4L and 9 would occur earlier. In the case described here, demand does not play a part in the decision, and since the wind speeds are low, dwell time assumes the major role.

As the storm activity becomes worse, the restrictions on the configuration choice become more severe due to low ceilings, decreased visibility, and increased wind speed. During the worst period of the storm, the airport is operating in IFR category III-A and only runway 4R may be used for arrivals. As is typical as such storms pass out to sea, the wind direction continues to become more northwesterly, the strength of the wind subsides, and the ceiling and visibility improves.

Changes in the wind data were given to the system after the schedule had been developed, causing backtracking and re-scheduling to produce a workable alternative. The system is able to answer the queries described in Chapter 4 and thus, in limited form, to explain its reasoning.

When operated on either of two UNIX based workstations (a VAXstation 3100 using Kyoto Common Lisp and an Apollo DN-4500 using Lucid Common Lisp), its performance was fast with the small rule set. When run on a 10 MHz IBM PC/AT clone using Gold

Hill Common Lisp the performance was about twenty three times slower than on the workstations. This level of performance would not be acceptable in a real-world situation, but it is an interesting point of data and it demonstrated the high degree of portability of the program.

Most of the time spent in performing the analysis is in searching the fact database for matches to the patterns. The facts are simply members of a LISP list in the current implementation, and search is effected by iterating down the list and testing. No attempt was made to try to speed up this process, though there are some obvious ways to do so. If, instead, the attribute names were interned as atoms in a special LISP package, and the facts with each attribute then became members of lists bound to those atoms, the powerful hashing function used by the LISP system itself could be harnessed to decrease search time.

Another way to increase the performance of the system is to take advantage of the "object oriented" nature of the implementation. Each instance of "expert system" created using **def-rule-system** has separate databases, so it is possible to create several communicating, but independent, systems with particular sub-domains (thus bringing into existence a *committee* of "experts"). This will increase the effective execution speed of the system because it will eliminate searching through information not relevant to the particular sub-domain. This idea could be used in conjunction with the one described above by making a package for each instance of rule system in the "committee."

Chapter 7

Conclusion

The main idea of this thesis is that the application domain of expert systems and planners can be extended into the realm of scheduling operations for qualitative control of complex systems. This new technology can be applied to diverse problems including maintenance scheduling, general resource scheduling, and control of systems that require a plan for the future.

The cognitive process of scheduling involves temporal reasoning, default reasoning, reasoning about the allocation of resources, reasoning about parallel activities, and the generation of and assignment of execution times to plans. Analysis of these kinds of cognition led to a single integrated representation and reasoning paradigm.

7.1 Recapitulation of the Thesis

In chapter 1, the overall problem of *operations management* was defined. A historical overview of the methods used in traditional automatic planning (i.e., action *ordering*) systems was presented. It was argued that these methods require unrealistic restrictions which severely cripple or even prevent the application of these automatic planning systems to many important "real world" problems. The requirements for a more powerful "dynamic scheduling system," which relax or eliminate the restrictions, were then specified. They were: the capacity to efficiently represent and make use of information which is temporally bounded; the ability to reason about parallel activities involving multiple

actors; the capability to represent a new kind of domain knowledge regarding the presistence of inferences; the capacity to manipulate changing beliefs about the future; and the capability to reason about, and create schedules with, actions which take varying periods of time to perform. The standard issues involved in most expert systems, ease of customization, the ability to generate explanations and speculation, are also important for a scheduling system.

A theoretical foundation for reasoning about scheduling in a dynamic temporal domain was developed in chapter 2. An extended temporal logic, based on Shoham's [Shoham 88] nonmonotonic temporal logic, was required to resolve a paradox encountered when reasoning about *willfully changing the future.* The new "picture" of the future depends logically on the previous beliefs which made up the "picture" of the future before the decision was made to change it. This led to the concept of *belief interval,* i.e., the time interval during which some belief about the future was held. A *belief interval* is associated with the reasoning process itself, as apart from an *activity interval* which defines a period associated with a domain state. In addition, a clarification of the ambiguous meaning of **not**, required for a clear understanding of default logic, was introduced.

Chapter 3 described *TIMEBOX,* a *Time Map Manager* (TMM) or temporal database manager. A set of structures allowing the efficient storage and organization of temporal knowledge was presented. Then, mechanisms to update the knowledge to reflect changes were set forth. Because such changes could originate from rule-based inference, a detailed explanation of the specification of the temporal extent (or persistence) of inferred knowledge was given. In particular, it was pointed out that the persistence of an inference is *domain knowledge* which must be associated with each assertion in the consequent of a rule. Several examples implementing various kinds of persistence were exhibited.

Chapter 4 discussed an implementation of a *Temporal System Analyzer* (TSA) called *RULESYS,* a rule-based inference program to project and analyze the future states of

a domain. This program, developed in object oriented programming style, forms the central core of the entire scheduling engine. All information, either observed or inferred, is channeled through the TSA on its way to being stored by the TMM. In addition to inferring new state information, rules in this system may assert requests to the scheduler to satisfy goals designed to change the percieved future of the domain. Two innovative features of the implementation were presented. The first was the breaking down of the antecedent of a rule into a set of patterns determining the *context* of the rule, and a separate set of *tests* to determine if the rule is applicable in the circumstance defined by the context. The second was the introduction of non-triggering antecedent patterns to reduce search time and improve rule specificity.

The third component of the operations management system, the *Scheduler* is detailed in chapter 5. Extending the ideas of domain independent planning in order to reason about multiple actors and parallel activity led to the creation of a "scheduler". The structure and content of the objects representing "actions" and "plans" required new attributes not present in previous planning systems, because scheduling requires more than just ordering the steps of a plan. The scheduler must know how much time each step will require in order to ascertain the time at which it should be executed and to be sure that the plan is completed on time. Initially, the scheduler assigns execution times for actions based on worst case estimations defined within each action. The schedule may be refined by computing a more precise execution time using context information as it becomes available. The fact that an action may require a time period to execute motivated inclusion of a slot to indicate the changes to the domain state *during* execution of the action, in addition to the traditional resulting changes to the domain *after* the action has been executed.

The uncertainty of the future and the asychronous receipt of domain information makes backtracking unavoidable. However, least commitment planning and, particularly, the use of scheduling horizons can reduce the impact of uncertainty on the scheduling

task.

Finally, in chapter 6, an application of the scheduling system to runway configuration management was described. Scheduling runway configurations has all the characteristics of an interesting operations management problem: it involves reasoning about a complex dynamic system with a predictable but somewhat uncertain future; some of the parameters affecting the dynamics are not quantifiable; and there are actions which the supervisor may invoke which can have a controlling effect on the future of the system. This problem is of particular interest because of its impact on safety and the high financial yield of an increase in airport efficiency of even a few percent.

The fact that even a prototype of a runway configuration scheduler could be represented using about 300 initial facts, 30 rules and half a dozen actions is a testament to the power and expresiveness that this approach provides.

7.2 Areas for Further Research

There is still much to be done. Currently, all input activity intervals must be clock times. It would be much more convenient to be able to enter times in qualitative terms by reference to information already known. Clock times often give a false sense of precision to information whose actual accuracy is fuzzy at best. However, it is well known that the process of maintaining a temporal database of the qualitiative kind described here is NP-hard. Perhaps a different approach can be found to handle this problem.

The database organization and search methods employed in the *Time Map Manager* are crude and can certainly be improved. The scheduler currently spends most of its time on this task.

More kinds of persistence functions need to be explored and implemented to describe domain dynamics. For example, an *accumulating* type of persistence to describe such processes as the deepening of snow as a storm progresses. Probabilistic consequents, like those described in [Dean 88], would be useful in many domains. Other types of

consequent assertions may be added in the future as experience dictates.

Perhaps the most important issue is that of uncertainty and efficient mechanisms to deal with it. Including temporal horizons to exclude consideration of information which is considered to be "too far in the future" to be reliable might be a good idea, in addition to restricting elaboration of schedules as described in section 5.3.

Other questions will, without a doubt, arise as the operations management system is applied to different problems. Currently, work is planned to continue to explore airport runway configuraton management and the classroom scheduling problem. Another candidate is aircraft maintenance scheduling, and yet another is airline crew scheduling. There appears to be no end in sight.

Bibliography

[AHD] *The American Heritage Dictionary*, Second College Edition. Houghton Mifflin, Boston, 1985.

[Allen 83a] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, **26**(11):832–843, November 1983.

[Allen 83b] James F. Allen and J. A. Kooman. Planning using a Temporal World Model. In *Proceedings of the International Joint Conference on Artificial Intelligence*, IJCAI-83, 741–747, 1983.

[Altermann 88] R. Altermann. Adaptive Planning. *Cognitive Science*, **12**, 1988

[Ambros 83] J. Ambros-Ingerson and S. Steel. Integrating Planning, Execution, and Monitoring. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 83–88, AAAI, 1983

[Andriole 85] Stephen J. Andriole. *Applications in Artificial Intelligence*. Petrocelli Books, Princeton, NJ, 1985.

[Bell 85] C. E. Bell and A. Tate. Using Temporal Constraints to Restrict Search in a Planner. In *Proceedings of the Third Workshop of the Alvey IKBS Programme Planning Special Interest Group*, IEEE, 1985.

[Bocheński 70] I. M. Bocheński. *A History of Formal Logic*. Chelsea Publishing Co., New York, 1970.

[Buchanan 84] B. Buchanan and E. H. Shortliffe, editors. *Rule-Based Expert Systems*. Addison-Wesley, 1984.

[Chapman 87a] David Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, **32**:333–377, 1987.

[Chapman 87b] David Chapman and Phillip Agre. Abstract Reasoning as Emergent from Concrete Activity. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, eds. Michael Georgeff and Amy Lansky, Morgan Kaufmann, 1987.

[Chapman 85] David Chapman. Nonlinear Planning: A Rigorous Reconstruction. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (IJCAI-85), 1022–1024, 1985.

[Charniak 85] Eugene Charniak and Drew McDermott *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, MA, 1985

[Copi 72] Irving M. Copi. *Introduction to Logic*. MacMillan, New York, 1972.

[Copi 64] Irving M. Copi. *Readings on Logic*. MacMillan, New York, 1964.

[Currie 85] K. Currie and A. Tate. O-Plan — Control in the Open Planner Architecture. In *BCS Expert Systems Conference*, Cambridge University Press, Cambridge, 1985.

[Daniel 83] L. Daniel. Planning and Operations Research. In *Artificial Intelligence: Tools, Techniques, and Applications*, Harper and Row, New York, 1983.

[Dean 90] Thomas Dean and Greg Siegle. An Approach to Reasoning about Continuous Change for Applications in Planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, AAAI, 132–137, 1990.

[Dean 89a] Thomas Dean and Mark. Boddy. Solving Time-Dependent Planning Problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (IJCAI-89), 979–984, 1989.

[Dean 89b] Thomas Dean, J. Firby, and D. Miller. Heirarchical Planning Involving Deadlines, Travel Time, and Resources. *Computational Intelligence*, **3**, 1989.

[Dean 88] Thomas Dean and Kenneth Kanazawa. Probabilistic Causal Reasoning. In *Proceedings of the Fourth Workshop on Uncertainty in Artificial Intelligence*, 73–80, 1988.

[Dean 87a] Thomas Dean. Intractability and Time-Dependent Planning. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, eds. Michael Georgeff and Amy Lansky, Morgan Kaufmann, 1987.

[Dean 87b] Thomas Dean and Mark Boddy. Incremental Causal Reasoning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, AAAI, 196–201, 1987.

[Dean 87c] Thomas Dean. Large-Scale Temporal Data Bases for Planning in Complex Domains. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (IJCAI-87), 860–866, 1987.

[Dean 85] Thomas Dean. Temporal Reasoning Involving Counterfactuals and Disjunctions. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (IJCAI-85), 1985.

[DeLong 70] Howard DeLong. *A Profile of Mathematical Logic*. Addison-Wesley, Reading, MA, 1970.

[Doyle 86] Richard Doyle, D. J. Atkinson, and R. S. Doshi. Generating Perception Requests and Expectations to Verify the Execution of Plans. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, AAAI, 1986.

[Doyle 79] Jon Doyle. A Truth Maintenance System. *Artificial Intelligence*, **12**:231–272, 1979.

[Drummond 88] M. Drummond and K. W. Currie. Exploiting Temporal Coherence in Non-Linear Plan Construction. *Computational Intelligence*, **4**(4).

[Duda 85] R. O. Duda, and J. G. Gaschnig. Knowledge-Based Expert Systems Come of Age. In *Applications in Artificial Intelligence*, Stephen J. Andriole (editor), 45–86, Petrocelli Books, Princeton, 1985.

[Duda 79] R. O. Duda, J. G. Gaschnig, and P. E. Hart. Model Design in the Prospector Consultant System for Mineral Exploration. In *Expert Systems in the Microelectronic Age*, D. Michie (editor), Edinburgh University Press, 153–167, 1979.

[Eldred 82] K. M. Eldred. *MassPort: Phase II Preferential Runway Study for Logan International Airport*. Massachusetts Bay Port Authority, 1982.

[FAA 85] Federal Aviation Administration. *Aviation Weather Services*, AC 00-45C. U.S. Department of Transportation, 1985.

[FAR] Federal Aviation Administration. *Federal Aviation Rules*, Parts 91 and 135. U.S. Department of Transportation, 1986.

[Feigenbaum 71] E. A. Feigenbaum, B. G. Buchanan, and J. Lederberg. On Generality and Problem Solving: A Case Study Using the DENDRAL Program. *Machine Intelligence*, B. Meltzer and D. Michie (editors), 165–190, Elsevier, New York, 1971.

[Fikes 72a] R. E. Fikes, P. E. Hart and N. J. Nilsson. Strips: Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, **3**, 1972.

[Fikes 72b] R. E. Fikes, P. E. Hart and N. J. Nilsson. Strips: Some New Directions in Robot Problem Solving. In *Machine Intelligence 7*, B. Meltzer and D. Michie (editors), Edinburgh University Press, 1972.

[Fikes 71] R. E. Fikes and N. J. Nilsson. Strips: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, **2**:189–208, 1971.

[Filman 88] R. E. Filman. Reasoning with Worlds and Truth Maintenance in a Knowledge-based Programming Environment. *Communications of the ACM*, **31**(4), April 1988.

[Firby 87] R. James Firby. An Investigation into Reactive Planning in Complex Domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, AAAI, 202–206, 1987.

[Fox 81] M. S. Fox, B. Allen, and G. Strohm. Job Shop Scheduling: An Investigation in Constraint-Based Reasoning. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (IJCAI-81), 1981.

[FTA 82] Flight Transportation Associates, Inc. *Airfield Capacity Analysis for Boston-Logan International Airport*. Flight Transportation Associates, Inc., Cambridge, MA, 1982.

[FTH] Federal Aviation Administration. *Flight Training Handbook*, AC61-21A. U.S Department of Transportation, 1980.

[Gaschnig 80] J. G. Gaschnig. *Development of Uranium Exploration Models for the Prospector Consultant System*. Final Report, SRI Project 7856, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1980.

[Georgeff 87a] M. P. Georgeff and A. Lansky. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, AAAI, 677–682, 1987.

[Georgeff 87b] M. P. Georgeff. Actions, Processes, and Causality. In *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, 99–122, Morgan Kaufmann, 1987.

[Georgeff 82] M. P. Georgeff. Communication and Interaction in Multi-Agent Planning Systems. In *Proceedings of the Third National Conference on Artificial Intelligence*, AAAI, 1982.

[deGroot 65] Adriaan deGroot. *Thought and Choice in Chess*. Mouton Press, The Hague. 1965.

[Gupta 90] N. Gupta and D. Nau. Optimal Blocks World Solutions are NP-Hard. Technical Report, Computer Science Department, University of Maryland. Forthcoming. Check this out... they've probably finished it by now.

[Hanks 90] Steve Hanks. Practical Temporal Projection. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, AAAI, 158–163, 1990.

[Hart 68] P. E. Hart, Nils Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimun Cost Paths. *IEEE Transactions on System Science and Cybernetics* SSC-**4**(2):100–107, 1968.

[Hart 75] P. E. Hart. Progress on a Computer-Based Consultant. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, (IJCAI-75), 1975.

[Hayes-Roth 79] B. Hayes-Roth and F. Hayes-Roth. A Cognitive Model of Planning. *Cognitive Science*, **30**:275–310, 1979.

[Hendler 90] James Hendler, Austin Tate, and Mark Drummond. AI Planning: Systems and Techniques. In *AI Magazine*, **11**(2):61–77, 1990.

[Hendrix 73] G. Hendrix. Modelling Simultaneous Actions and Continuous Processes. *Artificial Intelligence*, **4**:145–180, 1973.

[Hewitt 72] Carl E. Hewitt. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot.* MIT AI Laboratory Technical Report TR-258, 1972.

[Hofstadter 79] Douglas R. Hofstadter. Gödel, Escher, Bach: an Eternal Golden Braid. Basic Books, New York, 1979.

[Horvitz 88] E. Horvitz. Reasoning under Varying and Uncertain Resource Constraints. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 111–116, 1988.

[Kabanza 90] F. Kabanza. Synthesis of Reactive Plans for Multi-Path Environments. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, AAAI, 164–169, 1990.

[Kahn 77] K. Kahn and G. A. Gorry. Mechanizing Temporal Knowledge. *Artificial Intelligence*, **9**:87-108, 1977.

[Kanazawa 89] Kenneth Kanazawa and Thomas Dean. A Model for Projection and Action. In *Procedings of the Eleventh International Joint Conference on Artificial Intelligence* (IJCAI-89), 985–999, 1989.

[Kautz 86] H. Kautz and M. Vilain. Constraint Propagation Algorithms for Temporal Reasoning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, AAAI, 1986.

[deKleer 86] J. deKleer. An Assumption-Based Truth Maintenance System. *Artificial Intelligence*, **28**:127–162, 1986.

[Konolige 83] K. Konolige. A Deductive Model of Belief. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (IJCAI-83), 377–381, 1983.

[Lansky 87] Amy Lansky. A Representation of Parallel Activity Based on Events, Structure, and Causality. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, eds. Michael Georgeff and Amy Lansky, Morgan Kaufmann, 1987.

[Lenat 75] D. B. Lenat. BEINGS: Knowledge as Interacting Experts. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (IJCAI-75), 126–133, 1975.

[McCarthy 69] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Readings in Artificial Intelligence*, pgs. 466–472, Tioga Publishing Company, Palo Alto, CA, 1981.

[McCarthy 65] John McCarthy. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, MA, 1965.

[McDermott 82] Drew McDermott. A Temporal Logic For Reasoning about Processes and Plans. *Cognitive Science*, **6**:101–155, 1982.

[McDermott 80] J. McDermott. R1: An Expert in the Computer Systems Domain. *Proceedings of the First Annual National Conference on Artificial Intelligence*, 269–271, 1980.

[McDermott 79] Drew McDermott and Jon Doyle. An Introduction to Nonmonotonic Logic. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (IJCAI-79), 562–567, 1979.

[McDermott 78] Drew McDermott. Planning and Acting. *Cognitive Science*, **2**.

[McDermott 71] Drew McDermott. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorms and Manipulating Models in a Robot.* Ph.D. Thesis, M.I.T., 1971.

[Miller 86] R. A. Miller, H. E. Poole, and J. P. Myers. INTERNIST-1, An Experimental Computer-Based Diagnostic Consultant for General Internal Medicine. In *New England Journal of Medicine*, **307**(8):468–470, 1986.

[Miller 85] D. Miller, J. Firby, and T. Dean. Deadlines, Travel Time, and Robot Problem Solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (IJCAI-85), 1052–1054, 1985.

[MITRE 81] Mitre Corporation. *Overview of the O'Hare Runway Configuration Management System.* Federal Aviation Administration, Report FAA-EM-82-5, U.S. Department of Transportation, September 1981.

[NASP] Federal Aviation Administration. *National Airspace System Plan.* U.S Department of Transportation, December 1981.

[Newell 63] A. Newell and H. A. Simon. GPS: A Program that Simulates Human Thought. In *Computers and Thought*, eds. E. A. Feigenbaum and J. Feldman, McGraw-Hill, New York, 1963.

[Odoni 79] Amedeo. R. Odoni and Robert W. Simpson. *Review and Evaluation of National Airspace System Models.* Federal Aviation Administration, U.S. Department of Transportation, October 1979.

[OAG] Official Airline Guide. *Official Airline Guide.* Official Airline Guide, Inc., 1986.

[Patil 81] Ramesh S. Patil. *Causal Representation of Patient Illness for Electrolyte and Acid-Base Diagnosis*. Ph.D. Thesis, M.I.T., 1981.

[Pelevin 87] Richard Pelevin and James Allen. A Model for Concurrent Actions Having Temporal Extents. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, AAAI, 246–250, 1987.

[Reiter 80] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, **13**:81–132, 1980.

[Sacerdoti 77] E. D. Sacerdoti. *A Structure for Plans and Behavior*. Elsevier, 1977.

[Sacerdoti 75] E. D. Sacerdoti. The Non-Linear Nature of Plans. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence* (IJCAI-75), 1975.

[Schoppers 87] M. J. Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (IJCAI-87), 1039–1046, 1987.

[Schank 73] Roger Schank and Kenneth Colby. *Computer Models of Thought and Language*. W. H. Freeman, San Francisco, 1973

[Schank 77] Roger Schank. *Scripts, Plans, Goals, and Understanding*. John Wiley and Sons, New York, 1977.

[Shoham 89] Yoav Shoham. Time for Action. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (IJCAI-89), 954–959 & 1173, 1989.

[Shoham 88] Yoav Shoham. *Reasoning about Change:* Time *and Causation from the Standpoint of Artificial Intelligence*. MIT Press, 1988.

[Shoham 86] Yoav Shoham. Chronological Ignorance: Time, Knowledge, Nonmonotonicity and Causation. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, AAAI, 1986.

[Shortliffe 76] Edward H. Shortliffe. *Computer-Based Medical Consultations: MYCIN*. Elsevier, New York, 1976.

[Smith 83] Reid G. Smith and James D. Baker. The Dipmeter Advisor System — A Case Study in Commercial Expert System Development. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (IJCAI-83), 122–129, 1983.

[Stallman 77] Richard Stallman and Gerald J. Sussman. Forward Reasoning and Dependency Directed Backtracking. *Artificial Intelligence*, **9**:135–196, 1977.

[Steele 90] Guy L. Steele. *Common LISP: The Language*, Second Edition. Digital Press, 1990.

[Stefik 81a] M. J. Stefik. Planning and Meta-Planning. *Artificial Intelligence*, **16**:141–169, 1981.

[Stefik 81b] M. J. Stefik. Planning with Constraints. *Artificial Intelligence*, **16**:111–140, 1981.

[Sussman 75] Gerald J. Sussman. *A Computer Model of Skill Acquisition*. Elsevier, 1975.

[Swartout 83] W. R. Swartout. XPLAIN: A system for creating and explaining expert consulting systems. *Artificial Intelligence*, **21**(3):285–325, 1983.

[Tate 84] A. Tate and A. M. Whiter. Planning with Multiple Resource Constraints and an Application to a Naval Planning Problem. *First Conference on the Applications of Artificial Intelligence*, Denver, CO, Morgan Kaufmann, 1984.

[Tate 77] A. Tate. Generating Project Networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (IJCAI-77), 1977.

[Tate 75] A. Tate. Project Planning Using a Heirarchical Non-Linear Planner. Report 25, Department of Artificial Intelligence, Edinburgh University, 1975.

[Vere 83] S. Vere. Planning in Time: Windows and Durations for Activities and Goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-**5**(3):246–267, 1983.

[Vesonder 83] Gregg T. Vesonder, *et al.* ACE: An Expert System for Telephone Cable Maintenance. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (IJCAI-83), 122–129, 1983.

[Vilain 80] M. B. Vilain. A System for Reasoning about Time. In *Proceedings of the Second National Conference on Artificial Intelligence*, AAAI, 1980.

[Wellman 90] Michael Wellman. The STRIPS Assumption for Planning Under Uncertainty. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, AAAI, 198–203, 1990.

[Whitehead 23] Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica*, Second Edition. University Press, Cambridge, UK, 1927. Reprinted by Lowe and Brydone, 1959.

[Wilkins 83] D. E. Wilkins. Representation in a Domain Independent Planner. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (IJCAI-83), 733–740, 1983.

[Winston 81] Patrick H. Winston and Berthold K. P. Horn. *LISP*. Addison-Wesley, Reading, MA, 1981.