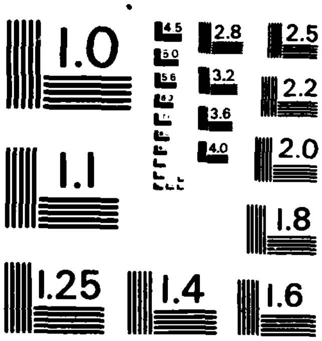
AD-A161 242 TEMPORAL IMAGERY AN APPROACH TO REASONING ABOUT TIME 1/3
FOR PLANNING AND PRO..(U) YALE UNIV NEW HAVEN CT DEPT
OF COMPUTER SCIENCE T DEAN OCT 85 YALEU/CSD/RR-433
UNCLASSIFIED N00014-83-K-0281 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

12

Temporal Imagery: An Approach to Reasoning about Time for Planning and Problem Solving

Thomas Dean

YALEU/CSD/RR #433

October 1985

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

12

Temporal Imagery:
An Approach to Reasoning about Time for Planning and Problem Solving

Thomas Dean

YALEU/CSD/RR #433
October 1985

Temporal Imagery: An approach to Reasoning About Time for Planning and Problem Planning

Thomas Dean

YALEU/CSD/RR #433

October 1985

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored under the Office of Naval Research under contract N00014-83-K-0281 and an IBM Graduate Fellowship.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER
YALEU/CSD/RR #433

2. GOVT ACCESSION NO.
AD-A161242

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Temporal Imagery: An Approach to Reasoning About Time for Planning and problem Solving

5. TYPE OF REPORT & PERIOD COVERED
Research Report

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Thomas Dean

8. CONTRACT OR GRANT NUMBER(s)
N00014-83-K-0281

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Yale University - Department of Computer Science
10 Hillhouse Avenue
New Haven, Connecticut 06520

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

12. REPORT DATE
October 1985

13. NUMBER OF PAGES
266

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)
Office of Naval Research
Information Systems Program
Arlington, VA 22217

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, If different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
temporal reasoning
planning
reason maintenance
frame problem

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Reasoning about time typically involves drawing conclusions on the basis of inncomplete information. Uncertainty arises in the form of ignorance, indeterminacy, and indecision. Despite the lack of complete information a problem solver is continually forced to make predictions in order to pursue hypotheses and plan for the future. Such predictions are frequently contravened by subsequent evidence. This dissertation presents a computational approach to temporal reasoning that directly confronts these issues. The approach centers around techniques for managing a data base of assertions

corresponding to the occurrence of events and the persistence of their effects over time. The resulting computational framework performs the temporal analog of (static) reason maintenance [Doyle 79] by keeping track of dependency inforamtion involving assumptions about the truth of facts spanning various intervals of time.

The system developed in this dissertation extends classical predicate-calculus data bases, such as those used by Prolog [Bowen 81], to deal with time in an efficient and natural manner. The techniques presented here constitute a solution to the problem of updating a representation of the world changing over time as a consequence of various processes, otherwise known as the frame problem [McCarthy 69]. These techniques subsume the functionality of current approaches to dealing with time in planing (e.g., [Sacerdoti 77]. [Tate 77], [Vere 83], and [Allen 83]).

Applications in robot problem solving are stressed, but examples drawn from other applications areas are used to demonstrate the generality of the techniques. The issues involved in processing temporal queries, propagating metric constraints, noticing the invalidation of default assumptions, and reasoning with incomplete knowledge are discussed in conjunction with the presentation of algorithms.

OFFICIAL DISTRIBUTION LIST

Defense Documentation Center 12 copies
Cameron Station
Alexandria, Virginia 22314

Office of Naval Research 2 copies
Information Systems Program
Code 437
Arlington, Virginia 22217

Dr. Judith Daly 3 copies
Advanced Research Projects Agency
Cybernetics Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209

Office of Naval Research 1 copy
Branch Office - Boston
495 Summer Street
Boston, Massachusetts 02210

Office of Naval Research 1 copy
Branch Office - Chicago
536 South Clark Street
Chicago, Illinois 60615

Office of Naval Research 1 copy
Branch Office - Pasadena
1030 East Green Street
Pasadena, California 91106

Mr. Steven Wong 1 copy
New York Area Office
715 Broadway - 5th Floor
New York, New York 10003

Naval Research Laboratory 6 copies
Technical Information Division
Code 2627
Washington, D.C. 20375

Dr. A.L. Slafkosky 1 copy
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Office of Naval Research 1 copy
Code 455
Arlington, Virginia 22217

Office of Naval Research — 1 copy
Code 458
Arlington, Virginia 22217

Naval Electronics Laboratory Center — 1 copy
Advanced Software Technology Division
Code 5200
San Diego, California 92152

Mr. E.H. Gleissner — 1 copy
Naval Ship Research and Development
Computation and Mathematics Department
Bethesda, Maryland 20084

Captain Grace M. Hopper, USNR — 1 copy
Naval Data Automation Command, Code OOH
Washington Navy Yard
Washington, D.C. 20374

Dr. Robert Engelmore — 2 copies
Advanced Research Project Agency
Information Processing Techniques
1400 Wilson Boulevard
Arlington, Virginia 22209

Professor Omar Wing — 1 copy
Columbia University in the City of New York
Department of Electrical Engineering and
Computer Science
New York, New York 10027

Office of Naval Research — 1 copy
Assistant Chief for Technology
Code 200
Arlington, Virginia 22217

Computer Systems Management, Inc. — 5 copies
1300 Wilson Boulevard, Suite 102
Arlington, Virginia 22209

Ms. Robin Dillard — 1 copy
Naval Ocean Systems Center
C2 Information Processing Branch (Code 8242)
271 Catalina Boulevard
San Diego, California 92152

Dr. William Woods — 1 copy
BBN
50 Moulton Street
Cambridge, MA 02138

Professor Van Dam 1 copy
Dept. of Computer Science
Brown University
Providence, RI 02912

Professor Eugene Charniak 1 copy
Dept. of Computer Science
Brown University
Providence, RI 02912

Professor Robert Wilensky 1 copy
Univ. of California
Elec. Engr. and Computer Science
Berkeley, CA 94707

Professor Allen Newell 1 copy
Dept. of Computer Science
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Professor David Waltz 1 copy
Univ. of Ill at Urbana-Champaign
Coordinated Science Lab
Urbana, IL 61801

Professor Patrick Winston 1 copy
MIT
545 Technology Square
Cambridge, MA 02139

Professor Marvin Minsky 1 copy
MIT
545 Technology Square
Cambridge, MA 02139

Professor Negroponte 1 copy
MIT
545 Technology Square
Cambridge, MA 02139

Professor Jerome Feldman 1 copy
Univ. of Rochester
Dept. of Computer Science
Rochester, NY 14627

Dr. Nils Nilsson 1 copy
Stanford Research Institute
Menlo Park, CA 94025

Dr. Alan Meyrowitz 1 copy
Office of Naval Research
Code 437
800 N. Quincy Street
Arlington, VA 22217

Dr. Edward Shortliffe 1 copy
Stanford University
MYCIN Project TC-117
Stanford Univ. Medical Center
Stanford, CA 94305

Dr. Douglas Lenat 1 copy
Stanford University
Computer Science Department
Stanford, CA 94305

Dr. M.C. Harrison 1 copy
Courant Institute Mathematical Science
New York University
New York, NY 10012

Dr. Morgan 1 copy
University of Pennsylvania
Dept. of Computer Science & Info. Sci.
Philadelphia, PA 19104

Mr. Fred M. Griffee 1 copy
Technical Advisor C3 Division
Marine Corps Development
and Education Command
Quantico, VA 22134

Temporal Imagery:
An Approach to Reasoning about Time for Planning and Problem Solving

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Thomas Linus Dean

May 1986

Abstract

Temporal Imagery:
An Approach to Reasoning about Time for Planning and Problem Solving

Thomas Linus Dean
Yale University
1986

Reasoning about *time* typically involves drawing conclusions on the basis of incomplete information. Uncertainty arises in the form of ignorance, indeterminacy, and indecision. Despite the lack of complete information a problem solver is continually forced to make predictions in order to pursue hypotheses and plan for the future. Such predictions are frequently contravened by subsequent evidence. This dissertation presents a computational approach to temporal reasoning that directly confronts these issues. The approach centers around techniques for managing a data base of assertions corresponding to the occurrence of events and the persistence of their effects over time. The resulting computational framework performs the temporal analog of (*static*) reason maintenance [Doyle 79] by keeping track of dependency information involving assumptions about the truth of facts spanning various intervals of time.

The system developed in this dissertation extends classical predicate-calculus data bases, such as those used by Prolog [Bowen 81], to deal with time in an efficient and natural manner. The techniques presented here constitute a solution to the problem of updating a representation of the world changing over time as a consequence of various processes, otherwise known as the *frame problem* [McCarthy 69]. These techniques subsume the functionality of current approaches to dealing with time in planning (*e.g.*, [Sacerdoti 77], [Tate 77], [Vere 83], and [Allen 83]).

Applications in robot problem solving are stressed, but examples drawn from other application areas are used to demonstrate the generality of the techniques. The issues involved in processing temporal queries, propagating metric constraints, noticing the invalidation of default assumptions, and reasoning with incomplete knowledge are discussed in conjunction with the presentation of algorithms.

Acknowledgements

I'd like to thank my adviser, Drew McDermott, for his patience, honesty, and unflagging interest throughout the period I was working on this dissertation. He provided direction and support while giving me the freedom to explore my own ideas. His intellect, energy, and enthusiasm have contributed a great deal to making Yale an excellent place to do research in computer science.

I'm also indebted to Dave Miller and Jim Firby, with whom I worked on the FORBIN planner. I hope they got as much out of our collaboration as I did.

Brad Alpert, Steve Hanks, Stan Letovsky, Yoav Shoham, and Jim Spohrer were patient enough to sort through numerous drafts of my often incoherent attempts at communication. Their feedback was invaluable.

Gene Charniak and Chris Riesbeck, the other two readers on my committee besides Drew, provided constructive criticism and a much needed alternative perspective on my work.

Contents

List of Figures

Chapter 1

Temporal Imagery

> "Nothing is predetermined; it is determined, or was determined, or will be determined. No matter, it all happened at once, in less than an instant, and time was invented because we cannot comprehend in one glance the enormous and detailed canvas we have been given – so we track it, in linear fashion, piece by piece. Time, however, can be easily overcome; not by chasing the light but by standing back far enough to see it all at once."
>
> — *Winter's Tale* by Mark Helprin

1.1 Introduction

The world about us is full of processes, most of which are beyond our control. In order to pursue goals and, in general, reason about the changing world around us, we have to make predictions about how these processes might actually manifest themselves. Some processes are relatively simple to anticipate, at least at the level required for us to cope successfully with them (*e.g.*, turning on a light or putting coins in a parking meter). Other processes require a great deal of effort in order to predict the precise manner of their unfolding (*e.g.*, the economic ramifications of the collapse of a chain of privately insured savings and loan institutions).

We can divide the class of processes that are worthy of our attention into two categories:[1] those that are *chaotic* or capricious and those that are *trackable*. Capricious processes are

[1] This particular categorization is due to Daniel Dennett [Dennett 84].

difficult if not impossible to predict with any high degree of precision (*e.g.*, the trajectory of a rapidly moving vehicle in traffic or the location of an untethered child). The best strategy for dealing with such processes is to simply avoid situations involving them that demand precise information. For instance, in traffic you generally try to keep some distance between yourself and other vehicles. In the case of watching over a young child, it is wise to confine the child in some reasonably safe area. If you are forced to deal closely with an unpredictable process, then the best you can do is attend to it carefully at execution time (*i.e.*, at the time when you actually have to carry out actions in order to deal with it). The trackable processes are those that one can accurately predict to the degree necessary to interact successfully with them. It is the trackable processes and the methods whereby one goes about anticipating them that this dissertation is concerned with.

Why are we all such avid prognosticators? What does it buy us to be continually second-guessing everyone and everything we encounter? It buys us time to take into account the possible consequences of acting in the variety of ways open to us and gives us the perspective to decide intelligently among the alternatives. It allows us to make opportunities by preparing us to exploit conditions which, had they come upon us unawares, we would have been powerless to take advantage of. These answers aren't all that deep or revealing. They reflect the simple fact that we have limited computational resources at our disposal for real-time processing. There are situations in which, due to the unpredictability of the processes involved, the only reasonable response is a tactical response (*i.e.*, one that can be determined only with information available at execution time). Luckily for us, many of the processes that directly impinge upon our lives are predictable. Anticipating such processes helps us to decide what has to bε done now in order to forestall, exploit, or direct the course of future events.

Anticipation also helps us to coordinate our own actions — actions often conceived in isolation but discovered during planning to interact in many subtle and not-so-subtle ways. The representation of a particular action or process is determined by a set of choices that the planner makes in the course of problem solving. Such decisions are guided by what *is currently known* and *what the planner currently aspires to*. Each choice constitutes a commitment on the part of the planner as to how the world will be. In the course of understanding the world and constructing plans to achieve goals, a planner has to make commitments it is prepared to later retract in the face of evidence of their inappropriateness. This evidence has typically [Sussman 75] [Sacerdoti 77] [Tate 77] taken the form of interactions found to lead to unpleasant consequences. Anticipation in this case helps us realize when we have made poor choices so that we can amend them before it is too late. This same basic functionality is critical in any sort of deduction that requires us to make predictions on the basis of incomplete knowledge. Predictions are generally defeasible. We make guesses that may turn out to be unwarranted. In planning and temporal reasoning in general, we have to be able to make assumptions and realize when those assumptions are no longer valid and the predictions they certify open to question.

Our knowledge of the world will admit to many possible extensions. One task faced by a problem solver is to construct a picture of the past, present, and future that conforms to what is known and provides a reasonable matrix for explaining critical phenomena and pursuing goals. The following chapters will describe a mechanism for exploring the possibilities given what is known about events occurring over time and the truth of propositions that vary with time. This mechanism is to be employed for extending what is known (making predictions) and recovering from conflicts in the event that new information clashes with old (*e.g.*, when actual observation conflicts with prediction).

Temporal imagery was conceived as a reasoning process akin to visually scanning large amounts of data arranged in the form of a map. This analogy has limited usefulness, but we can get a bit of mileage out of it before we get down to the harsh realities of data structures and algorithms.

The intuitive picture you should have of the process of temporal imagery is one of constructing maps, much like the maps used for plotting spatial information, and then scanning those maps to extract information and notice patterns. In many respects, time is a lot like space. The events in time are like the objects in a single-dimensional space. For instance, consider the towns along a railroad track. My information about these towns, just like my information about events, is likely to be incomplete. I may not know if Greenwich precedes or follows Stamford on the commuter line connecting New Haven and New York, but I know that both are closer to New York than Milford. My spatial information is often metric though fuzzy. The distance from New Haven to Bridgeport is between 15 and 20 miles. Similarly for events. I know that my recent trip to the grocery store took between 15 minutes and half an hour. It is also convenient to store temporal information in a form not unlike that used for planning trips through space.

In understanding history, it is often helpful to think about the past in terms of a time line upon which known events are carefully marked. This can help in making observations like, "I can understand the peasant unrest at this point in time due to the preceding years of famine and the increased taxes imposed as a result of a protracted war to secure their borders from invasion." Of course even in dealing with the past there are uncertainties. There are events whose order is unknown and about which detailed information is denied us. The present generally seems clearer due to its accessibility, but obviously there is much that is beyond our knowing. As we extrapolate further into the future, our predictions become more and more tenuous.

The right picture to hold in your head is of a graph in which the nodes are instants of time associated with the beginning and ending of events, and the arcs connecting these nodes describe relations between pairs of instants. Certain pairs of instants are more important than others, as they indicate important events and spans of time over which facts are said to be true. Reasoning about time in this framework consists of scanning this graph in order to determine how one event is related to another and what might be true during, before, or after an event. Since the instants are not totally ordered one can make hypotheses about what the consequences of certain additional ordering constraints might be and then proceed to explore some of those consequences. Moving from one instant in time to another does not require cranking a simulation backwards or forward. The reasoner is not confined to a single instant of time. It is assumed that it is quite easy to jump about on this graph and simultaneously keep track of a number of situations occurring at different times.

One of the important characteristics of this sort of graphical representation is that it can be updated incrementally. That is to say, as new events and facts are added only those parts of the graph which are affected need be changed. This is known as propagating the effects of a change. As this propagation proceeds it is important to be able to recognize when predictions formerly believed to be true are no longer so. This sort of graph scanning and updating is referred to as "reasoning about time from the side" [McDermott 82]. It's as though all of what you know about the past, present, and future is laid out in front of you. A simple example should help to strengthen this intuition.

Figure 1.1 shows a graphical representation of some events and their effects. Events are represented by two vertical bars connected by double horizontal bars (*e.g.*, |===|). Effects are represented as a vertical bar indicating when the effect is first believed to be true and either a second vertical bar providing some indication when the effect ceases to be true or an angle bracket > indicating that the effect is believed to persist indefinitely into the future. The end markers for an effect are connected by a single horizontal bar (*e.g.*, |------>).

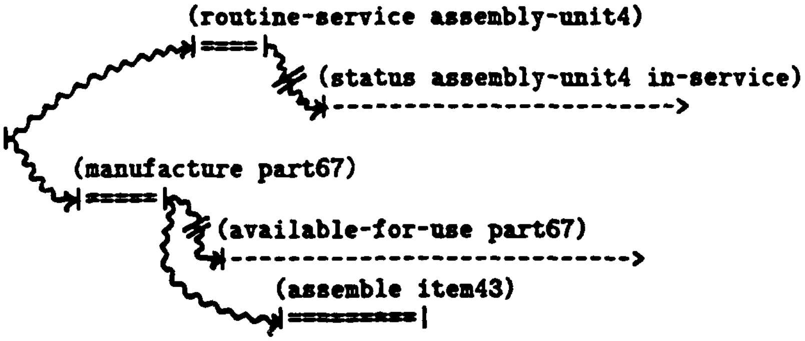

(routine-service assembly-unit4)
(status assembly-unit4 in-service)
(manufacture part67)
(available-for-use part67)
(assemble item43)

Figure 1.1: Simple graphical representation of events and their effects

Each event and effect is labeled with a formula describing the type of event or effect.

In Figure 1.1, the only orderings are those shown as squiggly lines and these indicate that one point precedes another. The order of precedence is indicated with an arrow. A squiggly line with double hash marks indicates that the two connected points are coincident. In Figure 1.1, there are three events (or in this case tasks). The first involves performing routine service on an assembly machine, **assembly-unit4**, and it has the effect that immediately following the task **assembly-unit4** is ready for work. The second task makes **part67** available for use and the third task involves the assembly of some composite object denoted **item43**. Now at the time the assembly of **item43** is to begin it is obvious that **part67** is available for use. It is not clear, though it appears possible, that **assembly-unit4** will be ready for work at this same instant. Let's suppose that the assembly task requires some assembly machine in order to successfully achieve its purpose. One way that this could be ensured would be to constrain the routine service task so that it ends before the assembly task begins. In Figure 1.1, this seems to be straightforward enough to accomplish but let's consider a slightly more complicated situation. Let's suppose that the assembly task also relies upon the fact that the conveyor, **conveyor34**, is running throughout the interval associated with the assembly. And, as an added complication, suppose that one side effect of performing routine service on **assembly-unit4** is to shut down **conveyor34** (assume that it is hazardous to perform service work on one machine while the other is running). The resulting situation is shown in Figure 1.2.

Now we proceed with the ordering constraints proposed earlier to ensure that **assembly-**

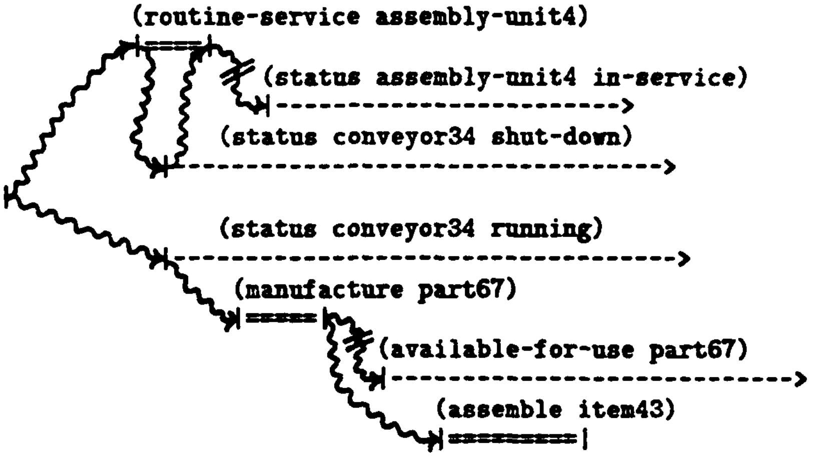

(routine-service assembly-unit4)
(status assembly-unit4 in-service)
(status conveyor34 shut-down)
(status conveyor34 running)
(manufacture part67)
(available-for-use part67)
(assemble item43)

Figure 1.2: Event with a side effect

unit4 can be employed in the assembly of item43. In Figure 1.3, the routine service task has been constrained to occur after the manufacture of part67 and before the assembly of item43.

Now we have a problem. Figure 1.3 has one effect indicating that conveyor34 is running and a second showing that it is shut down. Given that these two effects appear to overlap, it would seem that we have some sort of contradiction. It also seems clear that the contradiction should be resolved by eliminating the portion of the interval corresponding to conveyor34 running that overlaps with the interval corresponding to it being down (*i.e.*, the portion indicated with ?s in Figure 1.3). This resolution can be accomplished by constraining the effect indicating that conveyor34 is running to end before the beginning of the effect stating that conveyor34 is down. One apparent consequence of this is that it's no longer true that the conveyor is running throughout the assembly process. Such consequences should come to the attention of the planner so that it can take steps to correct the problem. Perhaps there is another assembly machine that will work instead of assembly-unit4 or, even simpler, the robot in charge of the routine service task could be instructed to turn the conveyor back on when it's through working on assembly-unit4.

This simple example illustrates the basic functionality involved in temporal imagery: the ability to determine if a fact or conjunction of facts is true at a time or could be made

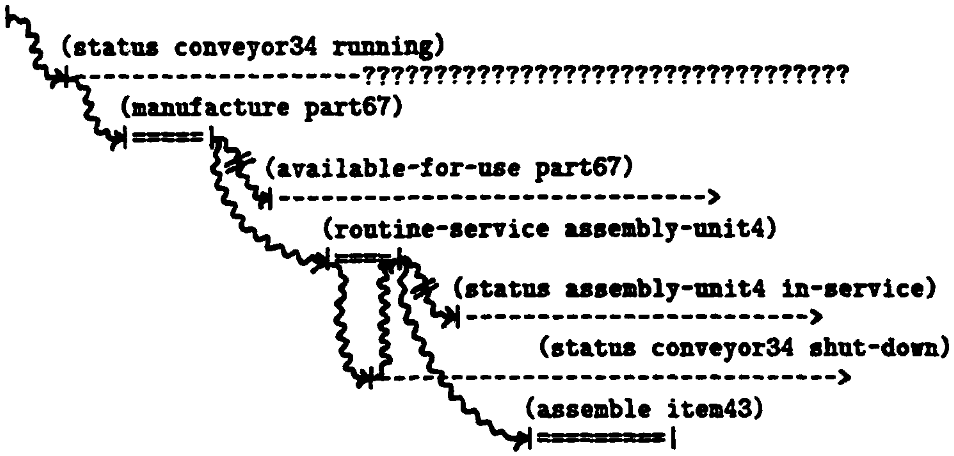

(status conveyor34 running)
(manufacture part67)
(available-for-use part67)
(routine-service assembly-unit4)
(status assembly-unit4 in-service)
(status conveyor34 shut-down)
(assemble item43)

Figure 1.3: Complications involving overlapping contradictory intervals

to be true with some additional constraints on the existing partial order. In addition, such a device should be able to point out the consequences which follow from some proposed change. It must be possible to make conditional predictions (*i.e.*, predictions that depend upon aspects of the data base that might possibly change) and then determine when such conditions are no longer met. Described in this way the requisite programs sound less like a set of routines for scanning maps and more like a system for retrieving and storing information in a data base. Indeed another way of viewing our graphical representation is as an indexing scheme for assertions in a temporalized data base. In Section 1.3, I'll shift the discussion to explore this perspective, but first I want to consider some more general aspects of temporal reasoning.

1.2 Shallow temporal reasoning

In the introduction, I referred to "a mechanism for exploring the possibilities given what is known about events occurring over time and the truth of propositions that vary with time." The use of the word "explore" is important here. It is difficult to encompass all the possible repercussions implicit in the information stored in a person's head or a program's data base — the closure of one's knowledge so to speak. A temporal data base captures what is known about events and their effects occurring over time. The information in a temporal data base only partially specifies the way things could be; there may be a great number of possible completions. Exploration involves constructing some of those possibilities and then choosing the one (or ones) that appear to be most likely (or most appealing if one has a choice in the matter). In planning, exploration might involve determining if certain preconditions for a plan might possibly be satisfied, given what is already known. In medical diagnosis, it might involve seeing if the known facts can be used to explain a symptom. In addition to extracting information from the data base concerning what is possible, it is necessary to confront the consequences of accepting certain possibilities. A diagnostician or planner will want to know what further commitments are required in order to accept an explanation or plan. For instance, if I'm to pick up a friend arriving on a 2:00 PM plane, I'll have to start for the airport right after my morning appointments; if I expect to meet my friend at the gate, I'll have to assume that the traffic on the turnpike will be light. Making commitments invariably leads to having to deal with certain *unforeseen* consequences: if I drive out to the airport this afternoon, I'm going to miss the colloquium scheduled for 3:30. Noticing when the assumptions supporting a given prediction are undermined by new information is a critical part of coping with uncertain circumstances. If I'm driving along the turnpike and I hear that traffic on the bridge leading to the airport is backed up five miles due to an accident, I will want to be aware that I may be late to pick up my friend. My response might be to revise my strategy for finding my friend once I enter the airport terminal: I'll look in the baggage claim area before I go to the boarding gates. If I were meeting a child, I might feel compelled to stop at a pay phone and have someone at the airline meet the child at the gate.

A large part temporal reasoning (and problem solving in general) consists of making predictions (guesses) on the basis of incomplete information, explicitly noting the assumptions under which those predictions are warranted, and then noticing when those assumptions are violated in the course of subsequent prediction and information gathering. This section introduces a general framework that supports this sort of reasoning. The framework is called *shallow temporal reasoning*. The reasoning is referred to as shallow because it is broken down into a number of steps, each of which is assumed to require only a small allowance of computational resources to carry out. Despite the relative simplicity of the individual steps, shallow temporal reasoning is not confined to making only simple inferences. Complex reasoning tasks are managed in this framework by performing the steps repeatedly in a cycle of inferences. Shallow temporal reasoning consists of the following steps:

1. generate a set of candidate hypotheses
2. select one hypothesis from among the candidates
3. use the selected hypothesis as a basis for prediction
4. respond to unforeseen consequences noticed in the course of prediction

These steps are quite similar to the expand/criticize cycle used in NOAH [Sacerdoti 77] and serve much the same purpose in general temporal reasoning. Hypotheses correspond to particular outcomes warranted by the currently available information about events, their effects, and their duration and time of occurrence. Committing to a given hypothesis may involve assumptions about the order in which various events occur, how long a fact will remain true, or which of several possible alternatives an agent is likely to choose. The selected hypothesis is used as the basis for making certain inferences or predictions, which are said to *depend upon* the selected hypothesis. These predictions can be divided into two broad categories: *projection* and *refinement*. Projection refers to positing the effects of an event (*i.e.*, determining what new facts follow from the occurrence of an event). Refinement consists of providing a more detailed description of an event, usually by producing a sequence or partially ordered set of (sub) events. Nc'icing and responding to unforeseen consequences constitutes a form of *debugging*.

It should be noted that in reasoning about time there are a number of combinatorial problems lurking in the background. Job-shop scheduling, travel time optimization, and resource management are all generally assumed to be intractable. The techniques described in this dissertation are not aimed at "solving" such problems. Their "solution" will, in general, require the clever application of knowledge (usually knowledge specific to a given domain). Shallow temporal reasoning is meant to capture the reasoning of an adept: a problem solver that is good at dealing with the sort of problems that normally confront it. Such a problem solver rarely explores a significant portion of the full search space associated with the problem at hand. The problem solver is not perfect, however, and sometimes it must perform a bit of surgery on a flawed solution in order to deal with some unanticipated event or complicated interaction. The problem solver is knowledgeable enough to analyze the situation and propose a patch that will work in most cases. If after several patches there is no solution in sight, the problem solver should admit that it's in over its head.

1.3 Temporal data base management

The primary claim of this dissertation is that it is possible to implement a temporal data base that naturally and efficiently extends classical predicate-calculus data bases. A significant amount of common-sense reasoning involves time in one aspect or another. Many of the facts we are accustomed to dealing with change over time, and hence many of the inferences we make depend crucially upon whether or not a fact or conjunction of facts is true at a point or throughout an interval. It seems reasonable that the machinery for performing routine temporal inference be built into the deductive engine underlying the data base. This dissertation demonstrates one particular way in which this extension of classical predicate-calculus data bases might be accomplished.

A temporal imagery device supporting the sort of functionality outlined in the previous sections can be viewed as a special sort of data base management system. Such a system might consist of:

1. A data base that captures what is known about events and their effects occurring over time. In particular, it is important to record information about the truth of propositions changing over time. The system should be able to handle the addition of new information and the removal of old information in an efficient manner using some criteria for internal consistency. If the user inadvertently adds information that can't be resolved with existing information, then the system should solicit the user's cooperation in order to exorcise the problem. Recognizing such situations and providing appropriate assistance is critical.

2. An interactive query language which allows the user to construct and explore hypothetical situations. This language should support simple retrieval of the form: Is it possible that P is true at time T given what is currently known? It should also handle retrieval of the form: Find an interval satisfying some initial constraints such that the conjunction (and P_1 ... P_n) is true throughout the interval. If retrieval depends upon additional constraints on the ordering of events, then the system should inquire if the user is willing to make such a commitment or consult some user-supplied program for permission to proceed.

3. A method for extending the information in the data base. On the basis of information extracted from the data base (antecedent conditions), the user should be able to engage

in some sort of forward inference (prediction). The predictions added to the data base in this way should depend upon the antecedent conditions in some meaningful way.

4. A mechanism for monitoring the continued validity of conditional predictions. This mechanism would extend the functionality of reason maintenance systems [Doyle 79] to temporal domains.

The data base is called a *time map* [McDermott 82]. The routines for retrieval, maintaining internal consistency, and handling forward inference are combined in what is called a *time map management* system or TMM. I will devote the next two subsections to a more detailed introduction to time maps and the complications involved in temporal reason maintenance.

1.3.1 Time maps

A time map is a graph. The vertices are points in time, corresponding to the beginning and ending of events. Constraints are represented as directed edges linking two points. Each edge is labeled with an upper and lower bound on the distance separating the two points in time. An interval is just a pair of points. Some intervals are more important than others, because they correspond to a particular occasion when a general type of occurrence happens. These intervals are referred to as *tokens* of that *type*. For example, to the *event type* "lunch at Ray's Greasy Spoon" there may correspond many *event tokens*, "lunch at Ray's yesterday", "lunch at Ray's today", assuming that one often eats at Ray's. In addition to tokens referring to events, a token can denote an instance of a fact becoming true and remaining so for some period of time (*e.g.,* Sino-Soviet relations are tense or funding is available for deserving researchers interested in mobile robots). For reasons that should soon become apparent, such *fact tokens* are called *persistences*.

The distinction between fact tokens and event tokens is often strained in common language. "Lunch yesterday" appears event-like but during this event it was true that I was engaged in "eating lunch", something distinctly fact-like. It is, however, a useful computational distinction. A token denoting the occurrence of an event has an associated interval whose duration is generally bounded rather closely. The bounds represent an estimate of how long the event took to occur. If someone makes a trip to the grocery store, it will probably take more than five minutes and less than an hour. My usual lunch seldom requires more than half an hour. Many events cannot be interrupted and still occur. The event described as "the inflation of the dirigible" cannot be said to have occurred if halfway through the operation of pumping hydrogen gas the dirigible burst into flame and collapsed to the ground.

The bounds on a persistence are treated quite differently. It is not assumed that the bounds specified when a persistence is first created are indicative of the duration of that fact. The bounds on the duration of a persistence are maintained so as to capture a default rule concerning the nature of facts: once a fact becomes true it will remain so until something happens that makes it false. This amounts to saying that the lower bound on a persistence is 0, or at least very small, and the upper bound is just the longest I am willing to believe that the fact will endure or persist unmolested. For instance, in lieu of information to the contrary, I'm willing to believe that the book I left on my desk this morning will remain there for several hours. I won't, however, make any bets on it being there next week. If, on the other hand, someone tells me that he borrowed that book, I'm willing to suspend my belief: effectively lowering the upper bound on the persistence. The lower bound on a persistence can be changed to reflect knowledge that overrides the default. If I was sitting at my desk reading the book all morning, then I'm not likely to accept someone's claim that they removed the book sometime during that period. Their claim contradicts my beliefs in a way that cannot be simply resolved.

A classical data-base assertion specifies a fact (type) that is timelessly true. Fact tokens, on the other hand, corespond to intervals during which the token's fact type is true. In general, there will be many fact tokens with the same type. Temporal data base queries generally refer to an interval or temporal index that defines the scope of the query. For example, the query `(tt ?pt1 ?pt2 (and P Q))` is interpreted as "determine if both `P` and `Q` are true throughout (`tt`) the interval from `?pt1` to `?pt2`". For this query to succeed, the data base management system must find a token of type `P` and a token of type `Q` such that each of their corresponding intervals span the interval `?pt1` to `?pt2`. Since there are likely to be many tokens of a given type, it is important to store them in such a way that the above sort of query can be performed efficiently. Among other things, this involves keeping track of how long fact tokens can be assumed to endure.

It is obviously possible to add tokens to the time map that contradict one another. There is no need for alarm, however, unless the two tokens span temporal intervals that necessarily overlap. It can't simultaneously be true that a light is both on and off. It is one or the other. If, on the other hand, I turned the light on in the morning and off in the evening, then it is likely that the light was on from the time I turned it on until the time I turned it off. If I had not turned it off, the light might have remained on indefinitely. If two tokens asserting contradictory facts are ordered such that one begins before the other and the earlier could persist longer than the beginning of the later, then the two are said to be apparently contradictory.

The time map machinery attempts to resolve apparently contradictory tokens by forcing the end of the earlier to precede the beginning of the later. If this can't be done, the system attempts to assist the user in removing the contradiction.

Tokens as they are used in the time map are akin to intervals in James Allen's work [Allen 83]. Persistences are sometimes confused with histories in the sense that Hayes uses the term [Hayes 79]. A Hayesian history is used to describe a chunk of space/time in which a given proposition is true. While it is true that persistences have no spatial extent, that is not the critical difference. Histories form a record that is essentially complete. You can retract or modify a history, but you can't amend its temporal extent by simply adding more information. Persistences are first and foremost a device for default reasoning. They were designed in such a way that their temporal extent can be easily modified in the face of new information. Persistences are used in the time map in order to efficiently update the temporal data base to reflect what events are believed to have occurred and the intervals of time over which certain facts are believed to be true.

The time map management system is not simply a simulator. You don't reason by stipulating a set of initial conditions and then simulating those events to generate a time line. As I mentioned earlier, the time map is a means of viewing time "from the side". You reason with a time map by suggesting modifications to the data base and then examining the repercussions of those modifications. The time map machinery sees to it that only important changes are brought to the attention of the calling program. As we'll see in the next subsection, the "important changes" are those that involve antecedent conditions used as a basis for making predictions. The TMM has to be able to detect when certain previously established antecedent conditions are no longer tenable and then alert the user that all predictions that depended upon these conditions are no longer supported. Persistences are the key to doing this efficiently.

1.3.2 Temporal reason maintenance

Resolving apparent contradictions is only one part of a strategy for handling shallow temporal reasoning. Most inconsistencies are neither so easily noticed nor so easily resolved. I can't go to a play in New York which starts at 8:00 PM and also attend a reception in Boston which begins at 7:30 PM on the same evening. The fact that these are not both possible depends upon the inference that attending both events requires traveling between their respective locations, a feat which is beyond most of us given the time constraints. Recognizing that some set of tasks or predictions are incompatible often requires a good bit of inference. That is to say the repercussions (projections and detailed descriptions) of a set of events must be explored in order to detect the source of their conflict. Since you can never tell in advance just where things might go wrong and what you may be forced to later retract, you have to keep track of why you believe things so that if the reasons for believing a fact go away, your belief in that fact and its consequences will evaporate as well. Hypothesis generation can be seen as the means for establishing the antecedent conditions supporting a given set of consequent predictions. Hypothesis generation is accomplished using the time map management system's query routines. Querying a changing data base is a tricky thing. It's not sufficient that the query mechanism return correct information; it must also keep track of the conditions under which that information continues to be warranted. If the antecedent conditions should fail, then the consequent predictions should be retracted or a new warrant established.

Systems that make the sort of conditional inferences described in the previous paragraph tend to be nonmonotonic (in the sense that Minsky used the term [Minsky 81]) in that the addition of new information is likely to result in the removal of old. The term "nonmonotonic" is bandied about in today's literature as though systems exhibiting nonmonotonic behavior were something special, as though one could choose whether to introduce nonmonotonicity into a system for dealing with the real world. Actually, nonmonotonicity is inherent in just about every aspect of reasoning in nontrivial domains. Nonmonotonicity arises as a result of the need to commit to predictions about how the world might be, despite the fact that you are bound to guess wrong on occasion.

The time map machinery extends the functionality of reason maintenance systems like Doyle's TMS [Doyle 79] to handle temporalized assertions. A (temporal) antecedent condition is generally something of the form: P is believed to be true throughout an interval. In the time map this translates into: there exists a time token of type P such that the token begins before the beginning of the interval and can't be shown to end before the end of the interval. If such a condition obtains, we say that P is *protected* throughout the interval. Notice that this is nonmonotonic. While it may be true initially that it can't be shown that the token ends before the end of the interval, additional constraints may change this. Such conditions are monitored using nonmonotonic data dependencies called *protections* (after [Sussman 75]). Protections and persistences can interact when one token is constrained by another (contradictory) token as a result of resolving an apparent contradiction. In order to deal with this and similar interactions, the TMM employs a temporal reason maintenance system that keeps track of what protections are warranted and hence what predictions supported by protections are warranted.

The TMM takes care of constructing protections for antecedent conditions and installing justifications where necessary. The underlying machinations of the TMM are not visible to the user. The query language appears no more complex than the typical pseudo-predicate calculus format used in Prolog [Clocksin 84]. The details will be left for later, but let's look at some examples to see how this figures in temporal reasoning.

Suppose that I'm trying to print a large document on one of the department's laser printers. In order to print the whole thing I estimate that it will take between 30 and 40 minutes during which the host machine must be up and the printer functioning. The document is being submitted for consideration in an upcoming conference and must be ready for express mail pickup by 4:30 PM. It's now 2:00 PM. My plan is quite simple: I will send the document file to the printer, I'll pick up the printer output sometime before 4:00, place it in an addressed envelope, and leave it in the express mail bin well before the 4:30 deadline. My prediction that this simple plan will work is dependent upon a number of assumptions that I have explicitly made in satisfying myself that everything will go according to plan. I went and made sure that the printer was functional. On the basis of this observation and my experience with the system I predicted that it would probably remain functional throughout the rest of the afternoon. I also called the express mail service and confirmed that the pickup would occur sometime after 4:30 and before 5:00.

Despite my care in initially establishing the warrant for my plans, any number of things could go wrong to threaten these assumptions. The printer could break unexpectedly. I could be told that the clerk at the express mail service was misinformed and that the pickup occurs regularly at 3:30. I could be delayed in sending the document file to the printer. Determining that my plan is threatened as a consequence of one of these three, or any of a score of other facts or observations, requires reasoning about time.

Suppose that I am told the laser printer is malfunctioning and will be taken out of service. Then my prediction that the machine would be up and running for the rest of the afternoon is no longer true. The persistence asserting that the printer will be taken out of service contradicts the persistence asserting that it is functional which was initially established by observation. Since the the first is not constrained to end before the second, the two are involved in an apparent contradiction. This can be resolved by adding the constraint that the persistence asserting that the printer is functional end before the persistence asserting the printer is out of service. Unfortunately my plan for getting the paper delivered on time depends upon the printer being functional throughout the time reserved for printing the document. In the system described in this dissertation the user would be notified of a failed assumption and told just what beliefs were implicated in bringing about the failure.

As another example, suppose that you're a detective investigating a theft from an art museum. You are trying to explain how the thieves could have absconded with a priceless painting right under the night watchman's nose. The watchman claims to have been in another part of the building from 12:00 AM till nearly one: the period during which the robbery occurred. His story is corroborated by the fact that there exists a record of his visiting one of the special electronically monitored security checkpoints located in a distant part of the museum at precisely 12:00. If he continued his rounds as usual he would not have returned to the scene of the crime until after 1:00. Later, however, you are told by one of the night janitors that the watchman was seen skulking around in the vicinity of the service entrance at a quarter past midnight. This entrance was apparently used by the thieves for their entry into the museum. This would mean that the watchman would have had to pass the security checkpoint at twelve and then run clear across the building to have been seen by the janitor at twelve fifteen. A quick jog tells you that it is quite possible to run the required distance in less than fifteen minutes. The watchman's alibi is now threatened by the new information and attention is focussed upon him as a possible collaborator in the crime.

The watchman's story was called into question by noticing that both the watchman's story and that of the janitor could not be simultaneously true. The janitor's story was upheld by the fact that there was an explanation which fitted the facts and his statement. The explanation is that it was possible for the surveillance system to have noted the presence of the watchman in one location at 12:00 and, following a fifteen minute run across the building, he could have been seen in the vicinity of the service door. Since the watchman (but not the janitor) possessed the means to disarm the alarm on the service door, you now have a possible explanation of how the thieves entered and exited the building without being detected.

Let's try to put these examples in the context of shallow temporal reasoning. I want to explain how the thieves could have avoided alerting the watchman. One hypothesis is that the watchman was elsewhere during the time the crime was occurring. In this case, the antecedent condition is simply that the watchman remained in the general vicinity of the electronic checkpoint throughout the critical period. On the basis of this condition, a consequent prediction is asserted: something to the effect that the watchman would not notice disturbances in the area where the crime occurred. Later, we try to incorporate the story of the janitor. Again new events are added (*e.g.*, the janitor observed the watchman) and new effects are predicted (*e.g.*, the watchman was in the vicinity of the service entrance). In this case the new predictions clash with existing ones, and the discrepancies between the watchman's alibi and the janitor's story have to be reconciled.

In the printer story, one of the predictions is that my plan for getting the manuscript safely to its destination will work. To make this prediction I have to assume that the printer will continue to function throughout the period I require its services. When this assumption fails, I am forced to construct a new plan or simply patch the existing one (perhaps the printer can be repaired in time or another printer can be found).

This pattern of using the time map to establish a set of antecedent conditions followed by the assertion of some set of consequent predictions illustrates a sort of controlled forward chaining which is typical in shallow temporal reasoning. The deduction is controlled in the sense that checking that the antecedent conditions are met for certain antecedent/consequent rules is the responsibility of the applications program. This control enables the program to determine when it is appropriate to apply certain rules and thereby carefully direct search. The time map also handles a sort of general purpose temporal forward chaining rule that can be used to reason about the physics of a given situation. The advantage of such rules is that they don't require that the applications program to continually check to see that their antecedent conditions are met. It is assumed that whenever the antecedent conditions are met, it is appropriate to make the consequent inferences. Suppose that you are reasoning about some complex operations in a factory involving flammable liquids and machinery that might produce sparks or generate intense heat. You might want to be particularly sensitive to conditions in which the former is likely to be exposed to the latter. Monitoring such situations by continually querying the data base is clumsy and inefficient. Disallowing such situations altogether is overly restrictive. All you really want is to be alerted to such situations when they arise so that you can exercise caution or prepare for contingencies. The TMM allows one to implement a restricted (no loops) form of envisionment [deKleer 82] [Forbus 84] for reasoning about simple processes. The rules can be applied selectively by specifying the rule as a fact token with limited temporal extent. This sort of forward chaining, which I call *auto-projection*, can play an important role in reasoning about time and responding to complex situations. Auto-projection makes it particularly easy to reason about the effects of actions in planning. A plan need only specify the actions, the (partial) order in which they must occur, and the dependencies between the actions in terms of prerequisites. Side effects and actions with conditional effects are handled by the auto-projection facility. Forward-chaining rules do not add to the deductive power of a system. They can however considerably alter the performance of a system [Moore 75], especially one that purports to be good at noticing important changes in the world.

1.4 A solution to the frame problem

There are two issues that loom large in any discussion of temporal data base management systems. The first concerns the integration of new information into an existing body of facts (data base maintenance), and the second concerns the extraction of selected information (data base retrieval). Both of these issues are related to a classic problem in AI known as the *frame problem* [McCarthy 69]. The frame problem involves inferring what has and has not changed in the relatively stable configuration of facts that surround a situation. Actually, the frame problem can be divided into two problems. The first problem is concerned with the economical statement of what are called *frame axioms*, preferably in a first order predicate calculus formalization of time. Frame axioms allow us to determine what facts remain unchanged after a situation in which some action is carried out. The second problem is concerned with efficiently updating a data base of facts involving events and their effects occurring over time as new facts are added and old ones removed. It is this second frame problem that concerns us here.

In a temporal data base, certain facts are true over some intervals, false over others, and unknown over still other intervals. This presents a problem for the routines responsible for retrieval and performing various sorts of conditional inference. For example, given that a fact P is made true at one point, it should be inexpensive to determine whether or not P is true at some later point. If there are a large number of intervening events, none of which mention either P or its negation, then those intervening events should not impose an additional burden on the retrieval process.

The time map management system uses a traditional predicate-calculus database to store facts that are believed to be timelessly true but which refer to particular intervals or instants of time. So for example, `(tt (date 9/1/82 12:00 AM) (date 12/3/85 12:00 AM) (enrolled Avery graduate-school))` might be used to represent the fact that Avery was enrolled in graduate school from September of '82 until December of '85. You might also have statements of the form `(before (graduation Avery high-school) (date 9/1/82 12:00 AM))` indicating that Avery graduated from high school before September of '82. In order to answer, "Yes.", to a question like, "Was Avery in graduate school during June of '83?", the system has to supply routines that interpret the facts stored in the data base appropriately. Interpretation is fairly simple if we're only dealing with points known precisely in a single global frame of reference. Once we introduce partial orders, fuzzy metric constraints, and multiple frames of reference, interpretation becomes considerably more difficult. To deal with this we can design routines for organizing and interpreting the set of timelessly true facts that encode the stored temporal information. For instance, to determine whether a fact P is true at a point pt1, find a point pt2 in the past where P became true, determine the first point pt3 after pt1 where P was made false, and then see if you can determine the order of pt1 and pt3. Some of these operations may require a certain amount of search, but if you're careful in organizing your facts, it's fairly straightforward to perform such basic inferences without examining every single thing that went on between pt1 and pt2. Using standard data dependency techniques we can cache deductions and perform indexing operations in such a way that the data base remains invariant relative to certain inferential criteria, despite additions and deletions performed by the user.

Maintaining temporal data bases efficiently is made difficult by the fact that changes with regard to one event potentially require reconsidering all inferences made with regard to later occurring events. The problem of data base maintenance in time maps is further complicated by the TMM's sophisticated representation of time. Unlike the early situation based approaches [McCarthy 69], the TMM deals with metric constraints, incomplete information, overlapping events, and simultaneous actions. If you're not careful, you can waste a considerable amount of effort just determining whether or not two points are ordered with respect to one another. The TMM uses selective caching and heuristic graph search techniques in order to carry out these operations efficiently. As in classical data bases augmented with reason maintenance systems [deKleer 78], it is convenient to make assumptions based on incomplete knowledge, and realize when those assumptions are no longer warranted. The TMM supports various methods for performing conditional predictions that extend the classical techniques to deal with time. The addition of a new event, fact, or constraint on the occurrence or duration of an event can result in the reconfiguration of a considerable portion of the data base. In most cases, however, the changes are minor. Temporal reason maintenance makes use of persistences and protections in order to see to it that data base updates are performed efficiently.

The techniques described in this dissertation constitute a solution to the frame problem. A major portion of my research involved figuring out a reasonable functionality for temporal reasoning, and then demonstrating that this functionality could be efficiently and naturally supported. Coming up with a solution to the temporal data base update problem was critical in providing this demonstration.

1.5 Strategies for reasoning about choices and commitments in planning

In this section, I want to look into the problem of recovering from bad choices made in planning. The issues addressed here are similar to those that have to be dealt with by any system responsible for making predictions from incomplete knowledge. I have chosen planning as a context in which to explore these issues because they are especially apparent in planning and because planning is an important area of application for temporal reasoning. The basic technique employed in hierarchical planning systems capable of recovering from bad choices [Tate 77] [Vere 83] is to make choices while keeping alternatives on a stack (or more complicated data structure). When you run into trouble you return or *backtrack* to some previous choice point and try one of the alternatives. If you always return to the last choice made (the top of the stack), then the recovery strategy is referred to as *chronological backtracking*. Backtracking is somewhat complicated by the fact that in the interim between choice points the data base may change significantly. Since the data structures involved are generally quite large, *transition-oriented* search is the method most often used for modifying the data base to reflect changes that you may wish to reverse at some later time. In transition-oriented search, you keep track of all actions performed between choices and then simply undo them during backtracking.

The backtracking scheme most often employed is simple chronological backtracking (the reason being that it is the simplest to implement), but other methods have been tried. Austin Tate's NONLIN [Tate 77] could return to any previous choice point by maintaining a record of how various prerequisites are dealt with and the structure of the planner's goals and the tasks designed to satisfy those goals. The only problem with this approach is that when a choice is undone all choices between that choice and the last choice made have to be undone as well. Tate's approach is better than pure chronological backtracking (assuming that the decision concerning what choice to backtrack to was intelligently made), but it still leaves a lot to be desired. The AMORD system [deKleer 78] introduced the idea of *dependency-directed backtracking*. This enabled one to return to any previous choice point (presumably the one most strongly implicated in the problem at hand) and attempt to fix the problem locally without (necessarily) throwing away all the work done in the intermediate steps of the problem solving effort. Keeping track of dependency information allowed the problem solver to undo just those steps which were dependent upon the decision being reversed. Lesley Daniel [Daniel 83] produced an extension of NONLIN which used a decision graph to keep track of dependency information and support much of the same functionality.

In the time map, we have incorporated certain of the strategies used in AMORD. In particular, the TMM subscribes to the basic idea of dependency-directed fault detection and analysis. Fault detection involves noticing when something that was previously believed is contravened by new information. For instance, suppose that I am constructing a house and decide to have the sheetrock walls installed earlier than previously planned. Dependency-directed fault detection should enable me to realize that my current plan for installing a custom stereo system will no longer work. Fault analysis involves determining precisely what conditions are implicated in causing the fault. Continuing with the construction example, fault analysis should help to determine that the reason the custom stereo installation will fail is because it depends upon placing the speaker wiring behind the sheetrock. In keeping with the dictates of shallow reasoning, these dependency-directed techniques should be coupled with methods for patching plans or recovering from predictive failures. It's also quite easy to incorporate a (full) dependency-directed-backtracking scheme into the time map routines, but that would be counter to the sort of shallow reasoning encouraged in this thesis. By caching information about failures, a planner can jump around in the search space, attempting to fix the problem locally, without sacrificing completeness. This information about failures is used to remove portions of the search space from consideration and to avoid exploring the same possibilities twice. Of course every planner that admits to the possibility of making poor choices has to keep around some information about what paths it has already explored, if for no other reason than to avoid endlessly cycling. In chronolgical backtracking the necessary information is kept to a minimum due to the systematic manner in which the search space is explored. Dependency-directed backtracking requires caching more information due to the way the planner jumps around in the search space. The method is complete, since once you have tried all your best guesses about how to fix the problem, you can fall back on a more systematic approach without fear of repeating yourself. But there is a practical flaw in relying upon such methods as a crutch to fall back on. Once you exhaust your local patches there is very little direction that the system can provide for exploring what's left of the search space. Unfortunately, the portion of the search space left is generally quite large. If the problem has few or even no solutions, then the system will not likely terminate in a reasonable amount of time, or at all. To rely upon combinatorial methods as a default strategy is a luxury available only to those working in trivial domains.

Planning schemes of the sort we are discussing here rely heavily upon knowledge of the domain they are used in. If you don't have the knowledge to make good choices, then perhaps exhaustive search is the best you can hope for (that is to say you have very little hope at all). But even if you can make good choices some of the time, the world is too variable, and planning in nontrivial domains too complex, to expect that one will make the right choice all of the time. A planner must also have the knowledge to recover from minor setbacks. If things become too complex, you might as well give up or reconcile yourself to a combinatorial search. Despite a certain amount of hype and misunderstanding concerning their use, dependency directed methods do not offer us an alternative in this situation. Using dependency-directed methods to explore an exponential space is misguided. I've tried such methods, and it's enlightening to experience firsthand how many times you can avoid an exponential amount of work and yet still have an exponential amount of work left to do.

It's interesting to see just how often exponential computations crop up in systems designed to support "critical" functionalities. Resource management is a prime example. In planning, everything is a resource in need of management. Even the truth of a proposition over a span of time can be viewed as a shareable resource. Managing resources inevitably requires making choices. Let's suppose that we have a bunch of objects (called a *pool*) alike in certain respects but different in others. For example, say we have a pool of lathes all of which can be used to turn metal (manufacture objects with circular cross sections), but some of them cut threads while others cut slots in shafts. The object during planning is to delay commitment to a particular, lathe until it's absolutely necessary to make a choice. You want to avoid the situation in which you commit to a particular lathe for use on one job, and then later find out that the special features of that lathe are essential to some other job that must be carried out in parallel with the first. By procrastinating, the hope is that you will avoid the need for backtracking in cases where you might have commited prematurely.

The management of lathes as a resource can be modeled as a partial order on *transactions*. Each transaction consists of either withdrawing a latne (exactly which lathe is not specified) from the pool (*i.e.*, putting it to use) or depositing a lathe in the pool (*i.e.*, making a lathe available for use). You want to put off committing to either the order in which the jobs will be executed, or the machines they will employ. Procrastination lets you avoid, in some cases anyway, the need to backtrack, but it also makes certain crucial deductions difficult. In particular it is costly to make sure that you don't overcommit yourself (*i.e.*, allocate what you don't have). You want to make sure that at all times there exists some schedule (total order) consistent with the constraints imposed thus far (the current partial order) such that the number of lathes in the pool never dips below 0 (*i.e.*, somebody needs a lathe when none are available). This task is generally assumed to be computationally intractable [Garey 79]. I claim that the planner would be as well served by committing to a particular lathe and then patching or debugging the plan when it ran into trouble.

Even if the scheme described above were computationally feasible, it provides no means for reasoning about several resources being managed simultaneously. While the planner may construct "some" plan, it may also ignore opportunities for merging tasks and consolidating effort. My main objection, however, is similar to that raised in response to dependency-directed backtracking. This sort of resource management system provides a uniform method that distracts attention from what I claim is the only real hope of dealing with such problems: namely the application of domain-specific patches in response to the detection of local interactions.

The strategy for resource management by making tentative commitments relies upon two basic functionalities. First you have to keep track of the reasons why you believe various things. This enables you to recognize an interaction, obtain information concerning the reasons for that interaction occurring, and reverse decisions that led to that interaction. The second functionality (definitely the harder of the two) is concerned with good guesses about which interactions to resolve. It is only the first that is handled by the TMM.

All of the schemes discussed thus far have problems in reasoning about merging plans involving several different sorts of resources. The problem can be traced to an inability to recognize opportunities in sets of exclusive alternatives. Let's consider a simple example involving the management of machine tools. Suppose that the planner has a task which involves reducing the diameter of a special shaft. Given the required close tolerances, this task can only be accomplished using the small screw-cutting lathe or the larger engine lathe. The task will take longer on the smaller machine but it won't tie up the more versatile engine lathe. Still the planner wishes to leave its options open. Now, let's suppose that the planner is given a second task which requires it to mill a slot in the same shaft. For this job it can use either the engine lathe or a milling machine specially designed for the job. The milling machine is located at some distance from the screw-cutting lathe so travel time between the two machines has to be taken into account. The shaft could be transferred from the screw cutting machine to the engine lathe but that would require that the shaft be positioned twice, once in each machine. Positioning a workpiece can occupy a significant amount of time. The best solution would be to perform both tasks in the engine lathe. In fact, the ability to perform several operations with minimal setup time is precisely the reason that machines like the engine lathe are purchased. Reasoning about alternatives allows one to recognize opportunities for using a versatile piece of equipment while avoiding its use in situations where its services are wasted.

The problem of recognizing opportunities to improve plans is distinctly different from the problem of dealing with potential failures. When you have a failure, at least you know that you have to consider some changes to your plan. In the example described above, not using the engine lathe for both tasks would result in a suboptimal plan, but the plan would still work. There was just an opportunity that the planner had no way of anticipating and would miss altogether unless it was able to represent certain of its alternatives. By representing several alternatives at once, a planner could piece together a plan by combining alternatives that are seen to complement one another. What I am suggesting here is rather disturbing. I'm suggesting that in addition to simply enumerating or sequentially exploring selected portions of an exponentially large search space we need to represent some number of the alternatives simultaneously. Not being satisfied with our precarious dance with a problem requiring exponential time, we're going to risk using an exponential amount of storage as well.

Well, it's not quite as crazy as it sounds. If we are considering n independent decisions consisting of 2 (exclusive) alternatives each, then there are 2^n possibilities that we have to consider. In most cases, however, these possibilities share an enormous amount of structure. If the interaction between alternatives is low, then the storage overhead can generally be kept far closer to $2n$. The time map provides an efficient mechanism for reasoning about exclusive alternatives. Using this mechanism, one can keep track of a number of alternatives at once. It keeps exclusive alternatives separate and provides the means for reasoning about compatible alternatives. By using virtual-copy techniques (assertions have labels that state under what combinations of alternatives they can be considered valid), the system can offer this functionality without an exorbitant storage overhead. The machinery can handle a wide class of temporal reasoning tasks involving disjunctions. It provides the functionality that supports the optimization of plans and provides an interesting framework in which opportunistic merging of plans can occur.

Not surprisingly, there are numerous opportunities for performing an exponential amount of work in supporting the above sort of reasoning. A certain amount of work is avoided by using structure-sharing methods so that only one copy of an assertion need be dealt with. In other cases, we sacrifice completeness for speed by using heuristics that avoid work during constraint propagation. But there's no way to avoid it altogether. If you want to solve an intractable problem, then either you're going to have to do an exponential amount of work or you already almost have the answer. Obviously the latter is preferable, and in some sense it's the only alternative open to us. Of course a good hunch about where to look is almost as good as having the right answer handed to us straight out, and it is good hunches that we are relying on to make the above scheme practical. I think the methods developed for the time map for exploring a number of alternatives simultaneously are as good as one could hope for.

The methods for handling exclusive alternatives are also employed for reasoning about counterfactuals and prevention tasks [Dean 85]. In order to determine if a task to prevent an event E is warranted, you have to keep in mind a world in which you don't take any preventive action. Once you have decided upon an action to prevent E, you have to be alert to changes that might render your efforts unnecessary or ineffectual.

The techniques for handling disjunctions and reasoning about exclusive alternatives constitute a departure from the standard approaches to planning. It is hoped that they may prove to be an efficient alternative to sequential exploration. I am afraid that I will not be able to provide any conclusive evidence one way or another on this score. These techniques are the newest addition to the TMM, and much of the research reported in these pages is preliminary. It's obvious that if they are to be used effectively they will have to be integrated into a well-thought-out strategy. The planner might selectively generate exclusive alternatives in situations where it "guesses" there may be some payoff. Equipped with the knowledge of what sets of alternatives are likely to be useful in a given set of circumstances, my expectation is that exploring several possibilities simultaneously will be considerably more effective than exploring them sequentially using standard techniques. This expectation is borne out of the fact that this notion of reasoning about several alternative descriptions of the world simultaneously is a natural extension of the idea of least commitment first demonstrated in Sacerdoti's use of a partially ordered network of tasks for planning [Sacerdoti 77]. The general principle is simply that if you have no principled way of distinguishing between two or more alternatives, consider them all. If this imposes no major decrement in predictive power, then procrastination is likely to payoff. There are of course situations in which indecisiveness leads to the reasoner being swamped. Least commitment and the techniques described here for reasoning about alternatives are simply heuristic strategies. If the number of outstanding alternatives is small, then the problem may be feasible (the exponential overhead may be within the computational capacity of the machine given the time constraints). You can't solve hard problems without the requisite experience or knowledge. The TMM provides machinery for exploring the possibilities in what you already know. It makes no effort to supply uniform methods for "solving" intractable problems. It does, however, provide all the necessary machinery for solving problems within the grasp of a planner suitably informed. It should come as no great surprise that these problems frequently correspond to the sort that a reasonably well endowed human can solve with proper training.

1.6 Related work

The literature on time in philosophy, logic, and computer science is quite extensive. In the following, I will restrict my attention to work that has been done in AI regarding automated temporal reasoning. For a wider perspective the interested reader is urged to consult the 1982 survey by Bolour et al [Bolour 82].

Much of the early work focussed on extending data bases to take into account the temporal dimension. Findler and Chen [Findler 71] described a data base system used for reasoning about cause-and-effect relationships. Their system dealt with events having duration and start and finish times that needn't be completely specified. One of the interesting aspects of this work is that it sought to deal with the problem of reasoning with incomplete knowledge. Around the same time (1971-2) research in natural language was attempting to deal with the problems of tense and temporal reference in processing text. The CHRONOS system [Bruce 72] was able to answer questions about incompletely specified information extracted from natural language input. Medical diagnosis was another area in need of techniques to deal with partial information. The "time specialist" module of Kenneth Kahn [Kahn 77] was capable of representing inexact *temporal facts. This imprecision* was represented using plus/minus error intervals for event dates, the duration of events, and the spans of time separating them. The work emphasized the need to organize events using reference events, before/after chains, and chunks of time (*e.g.*, historical periods). It also made use of the fact that reasoning could occur from different perspectives or frames of reference.

The system to be described in this dissertation directly addresses the problem of dealing with partial information. Every point in a time map is itself a frame of reference. Constraints between points are described as fuzzy intervals [McDermott 84] similar to Kahn's plus/minus error intervals. The TMM can exploit the structural properties of events (*e.g.*, causal and task/subtask relationships) in order to expedite queries. James Allen [Allen 83] has also understood the importance of such organizational strategies for dealing with large amounts of information. The indexing techniques described Chapter 4 were motivated in part by Allen's notion of *reference interval* and Malik and Binford's *reference frames* [Malik 83]. In [Dean 84], I compare Allen's techniques involving reference intervals with those used in managing time maps.

Another critical issue concerns what sort of things it's important to reason about (ontology) and what sort of basic axioms or inferences are required for common sense reasoning about time. Pat Hayes' "Naive Physics Manifesto" [Hayes 79] stressed the need to encapsulate both temporally and spatially the truth value of propositions. His notion of *history* was used to capture the temporal and spatial extent of propositions in order to facilitate reasoning about everyday phenomena. Drew McDermott developed a temporal logic [McDermott 82] aimed at much the same sort of thing. McDermott ignored the spatial extent of propositions and replaced histories with the notion of the *persistence* of a fact. The most important aspect of thinking in terms of the persistence of a fact is the default character of the reasoning process. A predication of the form `(persists some-fact from-some-instant for-some-length-of-time)` represents a guess about how long a fact, once made true, will endure. This guess can be amended in the presence of contrary information. McDermott's system was nonmonotonic: a property that made it less appealing to logicians but seemingly closer to the sort of reasoning a real program would engage in. Around this same time James Allen came out with an interval-based temporal logic [Allen 83] that had some very nice properties. Allen focussed on the *interval* as the most important object of manipulation (McDermott's is said to be a point or instant based logic). Allen also looked into the problem of implementing a system that employed his interval approach. This system computed the transitive closure of a set of interval relations in order to speed temporal queries.

The time map management system has taken something from each of these approaches. The TMM uses points or intervals with equal facility. As in Allen's logic, intervals, called tokens, associated with specific events and effects are the primary object of interest. A fact-lik^e^ token or persistence is like a Hayesian history in that it attempts to capture the temporal extent of a proposition. Persistences in the time map, like Hayes' histories, are terms or objects to be manipulated. Certain inferences are said to depend upon the duration of a persistence. Such dependencies are nonmonotonic. The query mechanism sets up these nonmonotonic dependencies (called *protections*) in order to monitor the continued validity of inferences made on the basis of the response to a query. The time map incorporates a temporal reason maintenance system to keep track of what's true as the user modifies the data base. This makes it possible to reason with precision about the repercussions of specific changes to the data base. To my knowledge this general approach is unique though there have been some interesting special-purpose mechanisms which have been developed for planning systems. Lesley Daniel [Daniel 83] developed an extension to NONLIN [Tate 77] which employed a decision graph to enable the system to backtrack efficiently. Her approach allowed the planner to undo any previous choice in such a way that only those subsequent choices that depended upon the recanted choice were affected. Steven Vere [Vere 85] has been looking into similar mechanisms for the DEVISER planner.

Planning research over the last two decades has in one way or another had to address the problem of reasoning about time. The STRIPS planner [Fikes 71] tried to construct (temporal) sequences of operators to achieve tasks. Gerald Sussman [Sussman 75] built a planner that attempted to recover from its mistakes by carefully simulating the plan, noticing potential problems, and then suggesting patches to avoid bugs in the form of *protection failures*. One of his important contributions was to incorporate the idea of a *protection* (*i.e.*, that a fact once made true should be preserved until it has served its purpose) into planning along with the idea of *criticism*. A critic is just a program that analyzes what went wrong when a protection fails. In the TMM noticing protection failures is carried out by the temporal reason maintenance system. There still is no real theory of criticism though some progress has been made [Wilensky 83].

Hendrix [Hendrix 73] considered methods for reasoning about continuous processes in the STRIPS paradigm. Hendrix' job was made somewhat easier by the fact that STRIPS only dealt with linear orders. Earl Sacerdoti demonstrated with his NOAH planner [Sacerdoti 77] that with the proper representation (*procedural networks*) it was not only possible but profitable to reason about partially ordered tasks. Austin Tate's NONLIN planner [Tate 77] showed how keeping the right information around allowed you to deal with inopportune choices by efficiently backtracking. The TMM is designed to reason about partial orders, but it is also capable of reasoning about metric time which neither NOAH nor NONLIN could. Steven Vere [Vere 83] corrected this deficiency by constructing an extension of NONLIN which was capable of reasoning about tasks that took time. Vere's program DEVISER assigned each task a duration and a window of time stipulating an earliest and latest start time. These two numbers might begin rather loosely specified and then become more and more tightly constrained as planning progressed and decisions about ordering tasks were made. The time map's fuzzy interval representation of constraints allows it the same sort of power. This has been demonstrated in the FORBIN planner [Firby 85] [Miller 85a] which solves problems in an automated factory domain.

The need to reason about deadlines is an important aspect of planning. The basic problem is one of scheduling a set of tasks in order to avoid deadline failures. Stephen Smith [Smith 83] designed a constraint propagation technique and temporal representation for solving job-shop scheduling problems in the ISIS project [Fox 82]. The same general approach (but more domain specific) was used in Goldstein's NUDGE system [Goldstein 75]. In the FORBIN planner the scheduling problem and the general problem of reasoning about partial orders have been separated out. In addition to the TMM, the FORBIN planner uses a scheduling module designed by David Miller [Miller 83] [Miller 85b]. Miller's scheduler efficiently explores the space of possible schedules to avoid making decisions that might lead to deadline failures. FORBIN uses the information supplied by the scheduler to make ordering decisions where necessary, while at the same time retaining the partial order on independent tasks to avoid backtracking.

The basic techniques used in NOAH have also been extended to employ interval based logics [Vilain 82] [Cheeseman 84] and reason about alternatives [Wilkins 84]. In regard to the latter David Wilkin's SIPE planner employs a context mechanism [McDermott 83] to reason about alternative plans for achieving tasks. The TMM extends this functionality by providing sophisticated techniques that enable a planner to reason about a number of exclusive alternatives simultaneously. In the area of reasoning about several alternatives simultaneously, I know of no work that directly addresses the temporal issues from a data base management perspective. However, my ideas on this were considerably influenced by the work of deKleer [deKleer 84], Martins and Shapiro [Martins 83], and McDermott [McDermott 83].

1.7 Summary

The machinery developed in this research was designed expressly to support common-sense, shallow, reasoning about time needed for planning and problem solving. It directly confronts the two most important issues involved in temporal reasoning: temporal information is generally incomplete and predictions made in the course of reasoning about events are generally defeasible. The result is a useful computational framework for planning and problem solving. This framework subsumes existing systems both in terms of representational power and in terms of functionality and computational efficiency. The time map routines have been employed in several planning systems for robot problem solving and promise to be a practical tool for both research and development. The rest of this dissertation will explore the details of the framework and possible applications for such a tool.

1.8 Thesis organization

The first chapter, which you are presently reading, is a broad overview and introduction to reasoning about time. The second chapter describes the terminology and ontological commitments behind temporal imagery: the objects and operations around which this approach to temporal reasoning revolve. The notions of token, event, and protection which we briefly mentioned in this chapter are discussed at greater length along with a number of new terms. The third chapter describes how to go about using the TMM system. This includes a large number of examples covering the full range of the system's capabilities. Chapter 3 is quite long as chapters normally go. You can consider Chapter 3 as a user's manual whose chapters correspond to the section headings of Chapter 3. Chapter 4 is rather technical. It describes time maps in terms of data structures and algorithms. A proof of correctness for the temporal reason maintenance algorithm plays a central role in this chapter. The fifth chapter deals with the application of temporal imagery to planning. The last chapter is a catchall: summary, suggested extensions, problems, and offhand remarks.

Chapter 2

Basic Terminology

2.1 Introduction

This will be a relatively short chapter. I want to introduce some terminology and explain how it fits into a general framework for reasoning about time. The basic notions are few and they are fairly easy to describe. The hard part will come in later chapters when I describe how these few notions provide the foundations for my approach to temporal reasoning. The concepts that appear in this chapter are basic to the rest of the dissertation and they will be explained more than once in the following chapters. Some of them will be familiar to you from the introductory chapter. The main reason for this chapter is to gather these concepts into a single place and set the stage for the computational theory to follow.

2.2 Ontological commitments: points, intervals, and time tokens

We begin with the idea of a time *point* or *instant*. Points are generally associated with things that are believed to happen in the world. They are always fictitious to some degree. A point may correspond to an "event" that never occurred. Points also have the property that they are often hard to pin down. Nevertheless, it is harmless and often useful to speak about "the point at which he realized the error of his ways" or "the point at which he entered the room". The sort of points we are speaking about needn't correspond to "real"

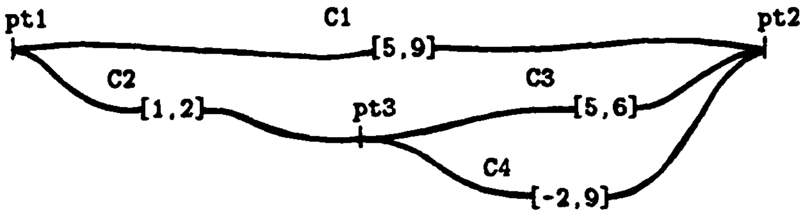


Figure 2.1: Relating pairs of points with constraints

events at all. I don't know when I will wake up tomorrow morning. Assuming that "the point corresponding to my awakening Thursday morning" makes sense, it's not likely that, unless I take special pains, I will ever know anything very precise about this event. For most purposes "between 6:00 and 7:00 AM" will suffice.

A point provides me with a frame of reference for reasoning about the temporal relationships between events. What may be described as a point from one perspective might be better viewed as an interval or pair of points from a second perspective. In the context of some esoteric debate concerning say, "the first conscious thought of the day", I may find it constraining to speak of the "the moment of my awakening" as a point. I may wish to speak instead of the interval of time over which I was in the "process of waking". I haven't eliminated the need to speak of points, however. In this case, it will probably be convenient to refer to the events corresponding to the beginning and ending of that interval, where these two events can be described as points.

The point of my awakening provides a reference point of sufficient resolving power for a wide variety of reasoning tasks. I can speak about the morning paper's arrival or my first appointment of the day in the frame of reference of "waking up this morning". I don't demand that the points I ascribe to have any physical reality. Points are simply inventions that serve to organize my knowledge of the world about me. Of course, it's desirable that the points I make reference to roughly correspond to events that actually happen in the world. Such a correspondence ensures that my perception of the world is not too far out of synch with reality.

To relate one point to another, we introduce the notion of a *point-to-point constraint* or simply a *constraint*. A constraint is just an upper and lower bound on the distance separating two points. There can be any number of constraints relating the same two points. If the constraints are consistent with one another, then the maximum of the lower bounds will always be less than or equal to the minimum of the upper bounds. I can relate any two points by finding a path from one point to the other, where a path from pt_1 to pt_n is just a sequence $\{pt_0, c_1, pt_1 \ldots c_n, pt_n\}$ such that pt_0 through pt_n are points and c_i is a constraint relating pt_{i-1} to pt_i. Figure 2.1 shows three points related to one another by several constraints. The constraints are directed. For every constraint from a point pt_1 to a point pt_2 with lower bound *low* and upper bound *high* there is an implicit companion constraint (usually not shown in diagrams) from pt_2 to pt_1 with lower bound $-high$ and upper bound $-low$. For each path connecting a pair of points, I can compute an estimate of the distance separating the two points by summing the upper (lower) bounds of the individual constraints in the path. There may be several such paths for any given pair of points. In Figure 2.1, {pt1,C1,pt2}, {pt1,C2,pt3,C3,pt2} and {pt1,C2,pt3,C4,pt2} are paths relating pt1 and pt2. Their respective bounds are [6,9], [6,8], and [-1,11]. Generally I'm interested only in paths that give either the greatest lower bound or the least upper bound. In figure 2.1, {pt1,C2,pt3,C3,pt2} provides both the greatest lower bound 6 and the least upper bound 8 on the distance separating pt1 and pt2. A set of constraints is said to be consistent just in case for all pairs of points <pt_0, pt_n> and all paths $\{pt_0, c_1, pt_1 \ldots c_n, pt_n\}$ where each c_i is in the set of constraints, the sum of the lower bounds of the c_i is less than or equal to the sum of upper bounds of the c_i. The set of constraints shown in Figure 2.1 is consistent.

Constraints are the glue that binds sets of points together in a network. That network is referred to as a *time map*. A good deal of the rest of this chapter will be concerned with adding more structure to this simple network.

An *interval* is just a pair of points such that one point is constrained to precede or be coincident with the other. Each interval consists of a *begin* and an *end* point such that the beginning precedes or is coincident with the ending. In Chapter 1 we introduced the notion of event and fact tokens corresponding to the occurrence of phenomena in the world. Such phenomena are categorized by *type*. A type is denoted by a formula like (operational-status lathe17 in-service) or (routine-service assembly-unit34). An interval together with a type will often be referred to as a *time token* (or simply *token* in situations where it should cause no confusion). An event or fact type is a class of phenomena (*e.g.*, all of the times, past, present, and future that I have attended, am attending, or possibly will attend a concert in Woolsey Hall). Time tokens are particular occurrences of types of events or particular instances of facts becoming true and enduring over a period of time (*e.g.*, the event token corresponding to my attending the concert featuring the Juilliard String Quartet in Woolsey Hall on December 17, 1985). The type/token distinction is meant to distinguish between classes of things and instances of those things.

In the computer, a token is described by a data structure. From now on, I will use the term "token" to mean this data structure rather than the actual occurrence. This abuse is harmless. In the computer it's possible for there to be two time tokens having the same event or fact type and coincident begin and end points. The two tokens are not identical as data structures, and it is quite likely that they came into being via different deductive paths; yet it would seem that they correspond to the same phenomenon. Making the identification may turn out to be critical in ascribing blame in complex situations [Shoham 85a]. However, we'll want to retain both data structures in the event that later evidence serves to distinguish them. Tokens are manipulated by a program as descriptions and like any other descriptions, a pair of tokens may turn out to describe the same thing. A simple example should illustrate.

Suppose that I know that some saboteurs have mined a bridge and then detonated the mines, causing the bridge to collapse. In addition, I am told that a plane dropped a bomb on that bridge at exactly the instant that the saboteurs detonated their mines. This also would cause the collapse of the bridge. Internally I have two tokens denoting the bridge's collapse. Each token has a different derivation. According to what I currently know, these two tokens denote the exact same phenomenon. If I wish to know what caused the bridge to collapse, then I will have to refer to the derivations for each token. If I later find out that the plane actually arrived 5 minutes after the mines were detonated, then I may want to amend my beliefs and deny that the plane's bomb caused the bridge's collapse.

I now want to distinguish fact tokens from event tokens, and, by connection, fact types from event types. An event token generally refers to an interval over which some activity takes place or some process runs its course. For example the time during which I rode the bus from home to work this morning would be an event token with event type `(transport self bus34 (home self) (work self))`. What distinguishes it as an event is that it has a fairly well defined duration. The criteria for its beginning and ending are self-contained, as it were. The "definition" of `(transport self bus34 (home self) (work self))` implicitly includes the conditions for its beginning, boarding bus34 somewhere in the proximity of my home, and ending, stepping off bus34 outside of Dunham Lab. If for some reason these conditions are not met (*e.g.*, bus34 breaks down somewhere along the way), then the event has not really occurred. Fact tokens generally refer to propositions which are made true and remain so for some (indeterminate) period of time. For instance I can reason about the fact that boarding bus34 resulted in my being on bus34 using a fact token with fact type `(on self bus34)`. It so happens in this case that the time tokens for `(transport self bus34 (home self) (work self))` and `(on self bus34)` have coincident begin and end points. If things had turned out differently, say the bus had broken down, then this might not be the case. If the bus had broken down halfway to work, then the token for `(transport self bus34 (home self) (work self))` would not represent a real event. The token for `(on self bus34)`, however, would still represent a real instance of a fact becoming true and persisting over some period of time; it's just that its duration would be different depending on whether or not the bus made it.

Fact tokens are referred to as *persistences* (after [McDermott 82]). The end point of a persistence is usually rather weakly constrained. That is to say, the end point of a persistence follows the beginning of the persistence but by how much cannot be determined in advance. The duration of event tokens will also depend upon context but not quite so much. It always takes at least 15 minutes and never longer than 45 minutes for me to get to work on the bus. Just how long depends upon the weather, the traffic, and any number of other factors. The duration of the `(on self bus34)` token lasts just until some action occurs (either I voluntarily exit or am forcibly expelled for one reason or another) which has the effect of my not being on bus34. The introduction of a second token with schema `(not (on self bus34))` (or alternatively something like `(on self ambulance17)`) is said to *clip* the persistence of an earler occuring token with schema `(on self bus34)`.

The distinction between event tokens and fact tokens (or persistences) is a pragmatic one, which will break down if you force it too hard. The programmer is free to choose whether to represent a phenomenon as an event or a fact. In most cases the choice will be clear. Persistences are employed for performing default reasoning about how facts change over time. It's convenient to be able to state that one effect of performing some action is that some fact will become true and persist for some period of time without having to be explicit about how long the fact will remain true. The default is simply that it will last just until something is known to make it false. Events, on the other hand, take a predictable amount of time. The predicted duration of an event is one of its most important attributes.

If it takes between 15 and 20 minutes to drive to the airport, then I will want to take that into account in planning to meet a plane. If I have 3 tasks to finish before leaving the office this evening, each of which take about half an hour, then I should consider this when setting up a dinner date. One of the main reasons for choosing one plan over another is that it takes less time than any of its alternatives. In the next chapter we'll see in more detail how this distinction between facts and events can be put to good use.

Now we have points glued together with constraints to comprise networks called time maps. The points have been minimally organized in terms of tokens denoting the occurrence c: events and instances of facts becoming true and persisting over time. There is a great deal more structure that we can impose on these time maps in order to facilitate temporal reasoning. In the next section I'll speak of complexes of tokens arranged hierarchically and causally.

2.3 Prediction: projection and refinement

We are constantly modifying and augmenting what we know about events and their effects. In planning we try to anticipate what might happen in order to plan accordingly. In other sorts of problem solving we try to make guesses about how processes interact with one another in order to formulate some sort of explanation that fits with our observations. One of the main operations in temporal reasoning consists of making predictions about the future, and then seeing how those predictions stand up under further scrutiny. These predictions generally fall into one of two categories. The first is called *refinement* and it refers to taking an event description (*i.e.*, a time token and its associated schema) and providing a more detailed description in terms of subevents and ordering constraints upon those subevents. A time map may contain several different descriptions of the same event where each description provides a different level of detail or perspective on the event. I might describe the construction of a house in terms of events corresponding to the activities of the various subcontractors (*e.g.*, rough framing, installation of plumbing and wiring, and landscaping). If I was the chief contractor for the house and a bank was underwriting the cost, I might also want to describe the construction in terms of events leading to the release of funds (*e.g.*, foundation poured, roof completed and shingled, final inspection passed). Each description is suitable for different reasoning tasks. The second sort of prediction is called *projection* and it involves additional tokens which describe the *effects* of an event. An effect is a persistence or event token which can be said to be "caused" by the event token which is being projected. I'm not going to be very precise about what I mean by "cause" (but see [McDermott 82] [Shoham 85b]) except to say that events cause other events and facts and we have rules that stipulate just what effects a given fact type has under what conditions.

There are many different ways of describing most events, and different descriptions suit different purposes. For instance, the token with schema `(transport self bus38 (home self) (work self))`, along with the information that it will take between 15 and 30 minutes, will suffice for many routine planning tasks. However, I can provide more detailed descriptions if need be. One description may include actions that I am required to take in order that this event proceed smoothly. So I might have tokens corresponding to boarding, paying, taking a seat, signalling the driver to let me off at my stop, and exiting the bus. Such a description would also include information about the order in which these subevents occur and estimates of how long they might take. I could even break each of these down further. The description of paying the driver might be broken down into finding a token or the correct change, putting it in the payment receptacle, and getting a transfer slip or receipt. I could also describe the `(transport self bus38 (home self) (work self))` event more in terms of the transporter, `bus38` in this case. This might include a more detailed description of the route taken by the bus (*e.g.*, `(translink bus38 PutnamAve-Station Hamden-Line)` and `(translink bus38 Hamden-Line Yale-Station)`). Such a description might be useful for reasoning about complications that arise in bad weather or under unusual traffic conditions. Each of these more detailed descriptions is called a refinement and for a given token there may be many such refinements.

The refinement of an event token E will typically include:

1. a set of event tokens $\{E_1, E_2, \ldots E_m\}$ suggested by refinement rules to provide a more detailed description of E

2. constraints upon the time of occurrence of the tokens in the sets mentioned in (1) relative to E

3. a better estimate on the duration of E

It is assumed that a good planner will choose a refinement appropriate for the immediate circumstances and its current plans and goals (obviously this can be an enormously complex process). The set of all descriptions of events in a time map constitutes a *refinement hierarchy*. One token is inferior to another in the refinement hierarchy if the first participates in the refinement of the second. In planning, the refinement hierarchy is essentially a task/subtask hierarchy. A planner like NOAH [Sacerdoti 77] *reduces* tokens corresponding to tasks by choosing an appropriate refinement and then ordering the steps (or subtasks) of that refinement to avoid problems. This is what is called *plan expansion* or *task reduction*.

There is a little more to task reduction than simply choosing a refinement of a task corresponding to a plan for carrying out that task. One also has to reason about the possible effects of the events in that refinement. Predictions about the effects of an event are called *projections*.

There are two basic sorts of effects that an event might have. The first concerns facts that will change as a consequence of an event occurring. McDermott [McDermott 82] refers to this as *persistence causation*. One consequence of `(transport self bus38 (home self) (work self))` is that I will be in the proximity of `(work self)`. This might be represented in terms of a persistence with schema `(location self (work self))` such that the beginning point of this token is coincident with the end point of the token associcated with `(transport self bus38 (home self) (work self))`. The other sort of effect an event can have is to cause another event to happen. If I ask the bus driver for a transfer slip, then, assuming that the bus driver knows his job, at some time later (usually no longer than a few seconds) the driver will hand me a transfer slip. If I pull on the bus bell cord to indicate I want to get off at the next stop then that will result in the event of the bell clapper striking against the bell.

To review, the projection of an event token E includes:

1. a set of fact tokens $\{F_1, F_2, \ldots F_n\}$ representing those facts that change as a consequence of E occurring
2. a set of event tokens $\{E_1, E_2, \ldots E_m\}$ suggested by causal inference rules to occur given occurrence of E
3. constraints upon the time of occurrence of the tokens in the sets mentioned in (1) and (2) relative to E

2.4 Conditional projection and refinement

Almost all projections and refinements are conditional in the sense that they depend upon certain facts being true in order to warrant their predictions. These are called *antecedent conditions* or sometimes (where it should cause no confusion) just *assumptions*. Suppose that I have to call the campus print shop before 9:30 AM if I want to make some changes to a manuscript that they'll be printing this morning, and suppose that I'm planning to wait until I get to work to make the call. I predict that the manuscript will be printed with the last minute changes assuming that I get to work in time. Giving myself some leeway I estimate that the phone call will take no more than fifteen minutes and therefore that everything will work out fine if I arrive at the office by 9:15. In planning it is often useful to make this dependency explicit. This allows me to notice when things are not going well. If I am delayed leaving the house to catch the bus, I should realize that this might endanger the plan for having my last minute changes incorporated into the printing. Having been alerted to this fact I might make the call from home before leaving.

There are other situations in which it is inappropriate to make certain dependencies explicit. Pulling the bell cord will result in the clapper hitting the bell only if the bell cord is attached to the switch that activates the bell. Unfortunately there are, in general, a great number of antecedent conditions required to (completely) justify such a prediction: the switch must be operational, the switch contacts can't be corroded, the wires from the switch to the clapper mechanism must be capable of delivering current, and on and on. For most of these things we're not likely to have the appropriate information or the time to gather it. And even if we did, in most cases it would not pay off. A reasonable planner does not try to prove that every action will have its intended effect. Instead, such a planner sets up tasks to check that actions result in certain anticipated effects. This is called execution monitoring [Charniak 85]. If an action fails to have its intended effeci, then the planner can instigate a contingency plan to correct for the failed task. For instance, if you pull the bell cord and you don't hear the bell ring, then you can yell loudly, run to the front of the bus, or cast yourself out the window.

Most sorts of strategic planning are such that it is crucial that the planner anticipate not only the actions of other agents and processes beyond his direct control, but also the consequences of his own actions for other tasks he is considering. In strategic planning (*e.g.*, investment, inventory control, and military planning), it is usually necessary to anticipate the future in order to determine how a set of plans and predicted events will interact.

Having got some idea about how a set of plans will work in a given set of circumstances, the planner may wish to reconsider those plans. In execution monitoring, you assume that the contingency can be handled when and if it becomes apparent. In strategic planning, you assume that by the time it is apparent it's too late to do anything about it. There are activities such that almost everything can be handled at execution time (*e.g.*, moving a robot arm in cramped quarters). Such a task is best carried out using local information and feedback [Lumelski 85]. Strategic planning methods, on the other hand, are useful in situations where an ill-considered action can cause irreparable damage. This is especially evident in dealing with deadlines and limited resources.

One of the main concerns in the following chapters will be to reason about the dependencies between predictions and their explicit antecedent conditions. In particular I will be discussing methods for setting up these dependencies and detecting when antecedent conditions are violated thereby undermining belief in the validity of predictions. The sort of antecedent conditions I am chiefly interested in here involve temporal dependencies. Temporal dependencies in the time map are called ***protections*** after Sussman [Sussman 75]. A protection consists of a fact type and an interval. A protection is said to have failed if the fact is not true over the interval. What this generally means is that there is no persistence with the specified fact type that can be shown to span the specified interval. The system we will be considering in the following chapters is responsible for setting up protections and detecting and annotating their failure.

2.5 Reasoning about alternatives

In our discussion thus far, a time map is a partially ordered set of tokens representing events and their effects over time. Each token represents the result of some projection or refinement step. These tokens can be thought of as the output of a decision process. They represent what the program has determined is correct or reasonable to believe about its present, past, and future. The resulting time map is still a far cry from a complete description of the world. Obviously there is a great deal that the planner doesn't even want to consider. But even for those few events the planner does consider, the time map does not necessarily commit to the order in which those events will occur. You can think of this as saying that all orderings consistent with the current partial order are equally probable (or perhaps equally uninteresting), given what the planner currently believes. Keeping the tokens partially ordered means that the planner can leave open or procrastinate about ordering decisions. While the planner does not have to commit to a given order, it is rather easy to reason about various orderings by simply restricting the current partial order. When it comes to projection and refinement decisions, it's not quite so easy to represent the alternatives open to the planner. If you wanted to reason about two exclusive alternative plans for achieving the same task you would have to expand the first plan, erase it, and then expand the second. One alternative to this would be to use some sort of *context mechanism* [McDermott 83] [Wilkins 84]. The basic idea here is that you have two data bases identical except for the fact that in one you've expanded the first plan and in the second you've expanded the second. Of course you don't really have two data bases; you just make it look that way by marking assertions carefully. You can visit one data base or the other simply by changing context (*e.g.*, considering only assertions with a given set of markings).

The main disadvantage of the context approach stems from the fact that contexts have to be explicitly constructed and then selected in order to draw inferences in them or examine their contents. What we would like is to specify the components[1] for constructing possible contexts and then have the system respond to requests by suggesting a context formed from these components which satisfies the request. This idea extends the functionality of delayed commitment in task ordering [Sacerdoti 77] to allow procrastination concerning plan choice.

Consider why the idea of using a partially ordered network of tasks and their effects to represent a plan is so appealing. Suppose that you have an action that you would like to carry out. And suppose that the action realizing its intended effect depends upon some conjunction of facts being true over the interval in which the action is carried out. With a partially ordered network of tasks, finding a suitable interval involves searching through the space of possible restrictions to the current partial order. The object of such a search is a set (or set of sets) of additional ordering constraints which, when imposed on the partially order network of tasks, result in the conjunction of facts being true over the required interval.

Now, we'd like to do something similar for decisions involving alternative predictions. By not making a prediction at all, we simply put off exploring some part of the search space. We'd like to represent that part of the search space that we think might be useful without committing to a specific alternative. The time map management system allows one to represent and reason about several alternatives simultaneously. I can say, for instance, that

[1] In planning the components might correspond to various options for achieving tasks. In diagnosis the components might be alternative explanations for observed phenomena.

I'll either take the plane or drive my car to Boston. I want to keep my options open. Later, I might ask the time map if it's possible that I'll be in time for a seven o'clock concert, and the time map might respond that yes, but only in the event that I decide to fly. Similarly if I wanted to take a side trip to Providence while in Boston it might be necessary that I have the car there. Of course, not all options are compatible (*e.g.*, I can't both fly and drive my car) so the time map provides a mechanism for avoiding consideration of impossible or undesirable combinations of alternatives. The search now is through the space of possible restrictions and compatible options consistent with the current partial order. This allows the planner to enlarge the scope of its search space in a controlled manner. A planner can also reduce this scope easily by committing to a particular alternative and rejecting its competitors.

A set of compatible alternatives constitutes what is called a *partial world description* or PWD. A given time map may contain many PWDs. Partial world descriptions are similar to Drew McDermott's *chronsets* [McDermott 82] in that both are used to represent possible courses of events. But McDermott envisioned implementing chronsets with contexts. In such an approach, the context corresponding to a chronset has to be explicitly constructed by the planner. In addition, inference occurs in a single context at a time, and specifying exactly which context to use is up to the planning system. In the time map management system, inference proceeds in every partial world description simultaneously. In response to a query, the TMM tries to find a set of alternatives corresponding to a partial world description satisfying the query. The planner is also notified when a particular PWD gives rise to a situation that the planner might be interested in. It's still up to the planner to specify what alternatives he's interested in. The system, however, takes on much of the responsiblity for recognizing combinations of alternatives which the planner should be aware of.

2.6 Summary

As I promised, this chapter is rather short. The main objective was to introduce some terminology and provide some preliminary intuitions about how the important concepts fit together. A time map is essentially a partially ordered set of points. Pairs of points (intervals) are associated with tokens denoting the occurrence of events and instances of facts persisting over time. A distinction between event tokens and fact tokens (or persistences) was made on purely pragmatic grounds. Event tokens capture the idea that actions and processes take time while persistences are used to support a default strategy for reasoning about facts that change over time. The event tokens are organized in refinement hierarchies so that several descriptions of a given event at various levels of detail (or from different perspectives) can be represented in the same time map. New tokens are generated in the process of prediction by either refinement (providing a more detailed description (in terms of subevents) of an event token) or projection (considering the effects of an event token).

Retrieval from a time map was presented in terms of searching through the space of possible restrictions to the current partial order. In addition to representing ordering options, it was argued that it is also useful to represent alternative (possibly exclusive) refinements and projections for a given event token. This requires that we extend our notion of retrieval to cover search through possible sets of alternatives.

The number of primitive concepts in this ontology is relatively small, but I think you will find it surprising just how much work they can made to do. The following chapters will place these concepts in a computational setting so that we can evaluate their utility and scope.

Chapter 3

Temporal Data Base Management

3.1 Introduction

This chapter discusses how to go about using the time map management system for temporal reasoning tasks. The *ontology presented* in the previous chapter will be carried over into a notation for temporal queries and a set of procedures for constructing and modifying time maps. A pseudo-predicate calculus notation is used for clarity and compatibility with an existing deductive retrieval system. The language presented here can be thought of as a variant of Prolog [Bowen 81], at least as far as the use of unification and backward chaining are concerned. Each formula has a specific procedural interpretation provided by the TMM. Explicating these interpretations will occupy a considerable part of this chapter.

The TMM is presented in the context of a data base management system. It is important to keep in mind the nature of information contained in a data base concerning events in the real world. First of all, in the best of circumstances that information is incomplete. There are:

- things you don't know but could if you had better information (ignorance)
- things that in principle you can't know (indeterminacy)
- things that you have absolute control over but are as yet unwilling to commit to their precise unfolding (indecision or indifference)

This incompleteness requires that the TMM be capable of distinguishing between what is possible and what is necessary given the extent of one's knowledge. It must be able to assist the user in dealing with the inevitable uncertainty that arises in reasoning about time.

The second thing to remember in dealing with time is that you are bound to make mistakes. You have to make predictions or commitments about how you think the world will be. Without commitment there is no basis for formulating plans or extending your predictions. If, however, you do commit yourself, the predictions you make are likely to be, on occasion, contravened by new information (*e.g.*, observations or other forms of strongly supported evidence). It's also quite likely that you will occasionally modify your own plans so as to invalidate assumptions made in the early stages of plan formulation. Recovering from this sort of situation typically requires resolving conflicts between plan steps that were initially believed to be independent. The TMM is constructed to notice when the assumptions made during prediction and planning are violated by subsequent modifications to the data base. This provides the basis for a style of reasoning that alternates between prediction and debugging using a number of shallow deductive steps. The TMM supports a computational framework called ***shallow temporal reasoning*** that facilitates this style of reasoning.

3.1.1 Shallow temporal reasoning and plausible inference

In this introductory section, I want to look at shallow temporal reasoning by breaking it down into three separate but interrelated components: (1) a data base or time map, (2) a strategy for interpreting the information in the time map, and (3) a process for using that information for planning, diagnosis, text comprehension or whatever.

The time map represents a commitment to how the world was, is, and likely will be. It attempts to capture the events that are believed to occur and the persistence of their effects. These beliefs are quite often complexly intertwined, one belief depending on another that depends upon others in turn, until finally we have beliefs that stand alone, requiring no justification. The TMM uses these dependencies to maintain a coherent picture of the world. An application program makes predictions using TMM utilities, and the time map keeps track of which of those predictions are warranted. Of course, the time map may not always accurately reflect the external world. It may also present a picture of the world that conflicts with whatever goals and expectations the program using the time map might be said to have. Shallow temporal reasoning provides a framework for constructing accurate and, to the extent that they are attainable, desirable pictures of the world.

The rules used in constructing time maps represent little packets of information about how processes behave and the conditions under which plans are likely to achieve certain desired effects. These rules contain sufficient information for the TMM to set up dependencies that ensure a prediction only in the presence of certain antecedent conditions. It is this dependency information, coupled with a strategy for noticing and resolving apparent contradictions, that enables the TMM to present a coherent picture of the world. Unfortunately, even if we assume that the information contained in these rules is essentially correct, we still have to deal with the fact that our knowledge of actual conditions in the world is incomplete. A program reasoning about the world must make guesses: commitments as to how the world actually is and choices about what knowledge to apply in a given set of circumstances. If the program makes the wrong commitments or applies the wrong rules, then it will paint an inaccurate and/or undesirable picture of the world. If the program notices a discrepancy between what it expects or observes and what the time map shows, then it can take steps to remove the discrepancy by retracting commitments and trying other rules. The time map is designed to represent the consequences of believing in certain events occurring in the world.

The preceding discussion points to two specific functions that the TMM should support. First, in order to maintain a coherent picture of the world, the TMM must be able to keep track of which facts in the time map are licensed by the dependencies set up in the course of applying rules. Second, it would be helpful if the TMM could indicate when additional commitments are necessary in arriving at a desired conclusion. To support these functions the TMM engages in two forms of plausible inference. The first involves the use of *default assumptions* and the second involves the generation of what I call *abductive premises*.

3.1.2 Default assumptions and abductive premises

A default assumption in the time map is generally something of the form,"As long as it's consistent to believe P you are licensed to believe Q." What this really means is, if I ever have sufficient reasons for believing (not P), then I will cease believing in Q. I have specified in advance the precise conditions for suspending belief in Q. In the time map, P typically corresponds to some fact or conjunction of facts being true throughout an interval of time.

An abductive premise in the TMM is something of the form,"If it is consistent to believe P (where P is an ordering relationship between a pair of points in the time map) then go ahead and do so." This is much stronger than a default assumption; an abductive premise constitutes a commitment. In data base terms, I am actually asserting P. Generally there exists no deductive warrant for making such an assertion. The application program is often simply engaging in wishful thinking. In diagnosis, you might have an explanation that fits the facts and you want to ask the time map what additional constraints you'll have to commit to in order for the explanation to actually apply in the current circumstances. In planning, you often want to know what it'll take for a plan to succeed. Suppose that a planner has a task that requires manufacturing a shaft of a specified diameter and length. The planner might have a rule that states that a particular plan pl-435 will work for manufacturing such a shaft, just in case there is a lathe available for a 15 minute interval satisfying whatever deadlines the planner is subject to. To come up with such an interval the TMM might have to impose some restrictions or additional constraints on the partially ordered time map. So, for instance, the TMM might state that pl-435 will work, if the planner is willing to wait until after 3:00. The additional constraint is referred to as an *abductive premise.* The process of gathering these additional constraints is built into the TMM query processing machinery. Whenever the TMM notices, in the midst of processing a query, that the computation involved in pursuing a particular answer to the query can proceed only if a certain additional constraint is added to the time map, it asks the program making the query whether or not the addition of the given constraint is acceptable, and if so, tentatively accepts it and proceeds with the deduction. If, having added P as an abductive premise, there is ever reason to believe (not P), then there is cause for alarm. Unlike the situation involving a default assumption, the TMM has no clear direction for dealing with the contradiction arising from the addition of (not P). If the calling program ever attempts to add (not P), the TMM will not allow it. The TMM will suggest methods for resolving the contradiction, but it will not take responsibility for resolving the contradiction on its own initiative.

Many sorts of temporal reasoning (*e.g.*, that involved in planning, diagnosis, or story comprehension) can be described in terms of an abductive process [Pople 73] [Charniak 85]. In Section 5.1, I will discuss the connection between abduction and planning in more detail. For now it is sufficient to understand that in order for planning to proceed, the program is going to have to jump to conclusions for which it has no deductive warrant. The TMM assists in this process by suggesting ordering constraints (abductive premises) that enable the program to apply a given rule (plan, explanation schema, etc.) in a particular time map.

The information in a time map is organized and represented using a set of special techniques that facilitate answering questions and maintaining a coherent picture of the world. The programs that use the time map require some strategy for interpreting its contents. This strategy consists of a set of conventions for reasoning about and extending the information stored in the time map. The conventions tell the time-map routines how to deal with events whose relative ordering is imprecisely known and persistences representing default information about how long certain effects will endure. A good deal of the computation involved in planning and problem solving consists of manipulating beliefs: essentially editing a set of formulae and directing the course of computations used in deriving new formulae. Some of this editing is performed routinely by the TMM's temporal reason maintenance system, and some of it requires the assistance of the program employing the TMM. The interpretation strategy enables both the TMM and the application programs that employ it to take into account the contingent or default nature of the facts stored in the time map.

The actual process involved in planning, diagnosis, etc. can be described in terms of making use of what you currently know in order to extend your knowledge. Extension might take the form of proposing details concerning how to achieve a task, elaborating upon a given explanation, or projecting the consequences of some hypothesis. This process is tricky because it requires that the extensions, in addition to being warranted by some of our prior beliefs, also fit comfortably with the rest of what we believe. We might try a particular extension only to discover that it conflicts with other beliefs or leads to an unpleasant state of affairs. In order to explore the repercussions of a given extension I have to make commitments; without commitment further exploration is impeded. Sometimes these commitments are tentative: hypotheses entertained for only a brief period of time. At other times the commitments become deeply entrenched in our view of the world. A simple example should serve to introduce the reader to some of the issues I'll be addressing in the rest of this chapter.

Suppose that it's Wednesday afternoon; I've just finished shooting a roll of film, and I'm anxious to see the developed pictures. I could take the film down to the corner drug store before it closes and have the pictures back by Saturday. In terms of representing facts in a time map, the commitment in this case involves adding the constraint that the task of delivering the film must precede 6:00 PM. I might decide to reject this option given that I had hoped to have the pictures ready for a party Friday evening. Instead I might decide that since I'm planning to go downtown this afternoon, I will take the negatives directly to the film processor and have them back in an hour. In this case, I constrain the delivery task to begin sometime after I arrive downtown. I could extend this commitment still further and constrain the task of retrieving the film to occur before I leave town, thus ensuring the success of my plan to recover the prints. This seems reasonable given that I want my plan to succeed. However, there is an alternative that will enable me to keep track of whether or not my plan is succeeding, while at the same time entertaining other plans (possibly more important) that might conflict with getting the prints home for dinner. To monitor my plan, I would have the TMM set up certain dependencies that would determine whether or not I will successfully return from the city with the prints. So, for example, I would represent in the time map the fact that my plan for returning home with the finished prints depends upon my remaining in the city at least one hour after dropping the film off at the processor.

Setting up these dependencies involves establishing default assumptions. These defaults explicitly establish the conditions for my suspending belief in the success of my plan for getting the prints. The advantage of setting up defaults is that it makes it easy to consider exactly what consequences are involved in considering other possibly conflicting plans and predictions. Suppose that the trip downtown gets delayed, and I decide to leave before 4:30 to avoid getting caught in traffic. Adding the fact representing my leaving the city before 4:30 would cause the TMM to notify me that my expectation of having the prints home for dinner is in danger. In addition, the TMM can inform me of what additional beliefs and commitments are responsible for endangering my plan. It's quite possible that I'd rather return to town tomorrow than endure rush hour traffic. The default approach makes it easy to explore the repercussions of adding new predictions to the time map.

In this chapter I am primarily interested in describing the basic functionality supported by the time map management system and in providing examples of how this functionality can be put to work in actual programs. I will show how temporal data dependencies are set up in the course of querying the data base (constructing hypotheses) and adding new information to the data base (making predictions). The result is a natural extension of the data dependency techniques developed for static data bases that enables one to reason effectively about temporalized information. To begin with, however, I want to present an overview of the deductive retrieval system upon which the time map management system is built

3.2 Deductive Retrieval Systems

The techniques described in this dissertation extend traditional deductive retrieval systems and their associated predicate-calculus data bases to handle a wide class of temporal inference. The TMM is implemented as an extension (or temporalized version) of the deductive retrieval system DUCK [McDermott 85] which follows in the tradition of MICRO-PLANNER [Sussman 71] CONNIVER [McDermott 73] and AMORD [deKleer 78]. In order to understand how to use the TMM, it is important that the reader understand certain things about the underlying deductive retrieval system. This section consists of a short tutorial on DUCK to provide the reader with the necessary background material for understanding the rest of the chapter. I will assume some familiarity with deductive-retrieval and logic-programming issues such as unification and backward chaining. Since Prolog is perhaps the most accessible logic-programming system, I will make frequent comparisons with the Prolog language [Bowen 81].

The discussion of deductive retrieval issues can be roughly partitioned as follows:

1. a set of techniques for constructing and maintaining a data base of predicate calculus formulae corresponding to *facts* or ground assertions (predications without variables) and *rules* (quantified formulae) used for deducing additional facts

2. a set of deductive methods for answering questions and noticing consequences of the information contained in the data base

Much of the discussion concerning these issues will be cursory, as it deals with issues that should be familiar to most readers. Readers who need more background are urged to consult the DUCK manual [McDermott 85] or any of a number of good introductory AI texts (*e.g.*, [Charniak 80] [Nilsson 80]). My objective is to move quickly through the fundamentals, pointing out certain syntactic conventions used by DUCK, and raising issues that will arise in later discussions concerning temporal data bases. Section 3.2.3 provides a relatively detailed description of the techniques employed in DUCK for keeping track of the reasons for believing items stored in the data base. These techniques are incorporated in a general deductive strategy for noticing and responding to changes in the data base. Understanding this strategy is important for effectively using the TMM and making sense of the implementation issues described in Chapter 4. I'll begin with some notational preliminaries.

3.2.1 Notation

A LISP-style notation will be used for predicate calculus formulae. All formulae are of the form (q ...), where q is a function, predicate, or connective, and the "..." corresponds to the arguments of a predicate or function or the subparts of a composite formed with a connective. Variables (notated ?name) are universally quantified unless otherwise stated. A substitution (or set of variable bindings) is notated as $\{(variablename_1\ value_1) \ldots (variablename_n\ value_n)\}$ (*e.g.*, unifying (foo ?x ?y) with (foo Fred Sally) results in the substitution {(x Fred)(y Sally)}). The connectives and and or have their standard interpretation. Instead of using not, the negation-as-failure operator thnot will be used (*i.e.*, (thnot ?p) succeeds just in case ?p fails). In places where one would normally use if, one of <- or -> will be substituted, where the former indicates a standard backward-chaining rule and the latter a forward-chaining rule. A formula such as (<- P (and Q_1 ... Q_n)) (equivalently in Prolog "P :- Q_1, ... Q_n.") is interpreted as saying: to prove P, prove Q_1 through Q_n with appropriate substitutions made via unification. The formula (-> P Q) is interpreted as: if P is added to the database, add Q as well. Variables appearing in formulae are declared to be of a given type using the constructs define-predicate and define-function. Types will appear in upper case, all else in lower case. The initial types include: FIXNUM (either an integer or a special symbol like *pos-inf* and *neg-inf* (for plus or minus infinity)), PROP (basically any finite non-circular list structure), and OBJ (all types are a subtype of OBJ). New types will be introduced as needed. Predicates are defined using a schema in which the arguments are replaced by a type. For instance (define-predicate (< FIXNUM FIXNUM)) declares < to be a predicate of arity 2, both of whose arguments are of type FIXNUM. Functions are defined similarly, except that the first element in the schema following the function name is the type which the function returns and the second element is a list of the types of the function arguments. So (define-function (+ FIXNUM (FIXNUM FIXNUM))) defines + to be a function of two FIXNUM arguments that returns a FIXNUM.

Lists are notated using the syntax !<*items-in-the-list-separated-by-spaces*> (equivalent to the Prolog [*items-in-the-list-separated-by-commas*]). So the list consisting of the terms fred and sally becomes !<fred sally> (or [fred,sally] in Prolog). The empty list is just `!<>`. There are also means of decomposing lists using unification. One can break a list consisting of one or more elements into its first element and the list consisting of the rest of the elements using `!<?x !& ?y>`, which is equivalent to `[X|Y]` in Prolog. That is to say ?x (`X` in Prolog) will unify with the first item in the list and ?y (`Y` in Prolog) will unify with the list consisting of everything else in the list. For example, `!<?x !& ?y>` unifies with `!<fred mary sally>` with substitution {`(x fred) (y !<mary sally>)`}, and it unifies with `!<fred>` with substitution {`(x fred) (y !<>)`}, but it doesn't unify at all with `!<>`.

3.2.2 Deduction involving forward and backward chaining

Most deductive retrieval systems separate the deductive process into two distinct operations on the data base: processing queries and performing forward inference. The former is often referred to as *backward chaining*. Coupled with unification, backward chaining is the primary instrument of procedure invocation (Horn clause resolution) in systems like Prolog [Clocksin 84]. The latter is called *forward chaining* and is often associated with production systems or pattern directed inference systems. OPS5 [Forgy 81] encourages a style of programming that employs this sort of deduction. In many systems (*e.g.*, [McDermott 73] [Sussman 71] [deKleer 78]) forward and backward chaining are combined in some sort of a deductive strategy. Such strategies are necessary to ensure that certain routine deductions are performed quickly using a minimum of storage [Moore 75].

Utilities for performing forward inference can be further divided into two classes: those that support logical implication and those that support a more general sort of inference sometimes given the unwieldy title of *pattern directed procedure invocation*. The former allows one to say, for instance, that whenever one asserts `P` one can assume `Q` (*i.e.*, `(-> P Q)`). The latter is used to instigate a computation in the event that a proposition is added to or removed from the data base. For example, one might have a rule that says, whenever a new employee is entered into the data base, check to see if there are any other employees with whom he can share a ride to and from work (*i.e.*, `(-> (employee-record ?name ?parameters) (call (check-car-pool ?name ?parameters)))` where `call` is a pseudo predicate that informs the deductive retrieval system to invoke LISP and `check-car-pool` is a LISP function that performs some complicated search).

Neither forward chaining nor backward chaining are adequate alone. It is clear that forward chaining alone would not suffice. For a simple rule like `(-> P Q)` where assertions of the form P were the only sort added by the user, you would double the space requirements. For a transitive rule you would increase storage by a factor of n^2. And in general, relying upon forward chaining alone would increase storage requirements beyond the capacity of any finite store.

One reason that backward chaining rules would not suffice stems from the sort of computation required for dealing with events in the real world. Information comes in bits and pieces. The implications of that information may only become apparent when some last crucial item comes in. Suppose that you have a system for monitoring a nuclear reactor. The system has a rule that says something to the effect, "if the coolant in the containment vessel is below a certain threshold, and the control rods are raised, then the reactor is in a potentially dangerous state." This rule might have the following form as a backward-chaining rule:

```
(<- (critical-situation ?reactor)
    (and (< (coolant-level (containment-vessel ?reactor)) threshold14)
         (control-rods-raised ?reactor)))
```

In order to detect a potentially dangerous situation the system would have to repeatedly make the queries (critical-situation reactor1), (critical-situation reactor2), etc. There may be many rules for deducing that a reactor is in danger and each of these would have to be repeatedly tried. In general, the cost of monitoring a complex system using backward chaining alone would be exorbitant. One reasonable alternative is to monitor certain key factors. These factors would serve as triggers for instigating further investigation. The "instigation" is accomplished using forward chaining; the "investigation" is carried out by backward chaining. Combining the two techniques of backward and forward chaining can result in powerful deductive strategies for integrating new information and noticing important developments.

A computation in a logic-programming system like Prolog can be described in terms of answering questions [Clocksin 84] (or, as I frequently put it, processing queries). In Prolog, you ask a question by typing a goal (generally a form containing variables but no connectives) or a conjunction of goals to the Prolog interpreter. The system tries to satisfy a goal by searching in the data base for a fact that unifies with the goal, or, failing that, by backward chaining (goal reduction) using the rules in the data base. An answer consists of a substitution for the variables in the query. A given query may have no answers or it may have many, depending upon the facts and rules in the data base. A query is said to "succeed" just in case it has at least one answer; otherwise it is said to "fail". DUCK

```
If the data base consists of:
  (string-quartet Juilliard)
  (string-quartet Guarneri)
  (cellist Joel-Krosnick Juilliard)
  (cellist David-Soyer Guarneri)

 then:
   (fetch '(and (string-quartet ?q)
                (cellist ?c ?q)))

 evaluates to the list of answers:
   ({(q Juilliard) (c Joel-Krosnick)}
    {(q Guarneri) (c David-Soyer)})
```

Figure 3.1: Invoking backward chaining from LISP

also has an interpreter that you can invoke to process queries, but DUCK encourages a style of programming that blends logic-based computation (or deductive retrieval) and the applicative style of programming common to LISP. The LISP function **fetch** takes a single argument that evaluates to a goal or conjunction of goals[1]. This function returns a list[2] of answers. Figure 3.1 provides a simple example. DUCK answers can be manipulated just like any other LISP data object, and we'll make use of this fact to control deduction in the TMM. The function **fetch** allows us to take advantage of Prolog-like deductive retrieval capabilites while in LISP.

There are computations involving deductive retrieval systems in which the data base does not change over the course of the computation. But in many applications it is convenient or even necessary to modify the set of facts and rules as part of the computation. This might be done for efficiency or it might be done simply because new information becomes available while the program is running. Prolog handles this by having predicates (**asserta** and **assertz**) that can be used to add facts and rules to the data base during backward chaining. DUCK has a predicate **assert** which serves the same basic function and a LISP function add that allows one to add facts and rules from LISP. **Assert** and add provide

[1] Both DUCK and Prolog queries can contain conjunctions and disjunctions of goals, but I've ignored this to simplify the discussion.

[2] It's actually what is called a *stream* or *generated list* of answers. In simplified terms, this means that only the first item on the list is actually computed and the rest are generated only on demand, as in "lazy evaluation".

the user with another method for performing forward chaining. Facts and rules can also be removed or *erased* from the data base. In Prolog this is accomplished by using `retract` and in DUCK by using `erase`.

In both Prolog and DUCK, exactly what is *asserted* or added to the data base will "depend" on the *current answer* or state of the computation. In Prolog, this dependency can be stated totally in terms of the variable bindings (substitution) in force at the time the assertion is encountered in the computation. As an example let's assume that the data base consists of just the fact `(brother-of Cain Abel)`. In the following query:

```
(and (brother-of ?x ?y) (assert (sibling ?x ?y)))
```

the assertion will occur in the context of the substitution {`(x Cain)(y Abel)`}, resulting in `(sibling Cain Abel)` being added to the data base. There is actually a more complicated implicit dependency in this; it would seem that the assertion `(sibling Cain Abel)` should somehow depend upon the assertion `(brother-of Cain Abel)`. If the latter assertion is removed from the data base, then there seems to be no warrant for believing former. In DUCK, this dependency is made explicit in what is called a *data dependency network*. Assertions that are no longer justified are marked as such and in certain circumstances actually removed from the data base by what is referred to as a *reason maintenance system* (RMS).

3.2.3 Reason maintenance and data dependency networks

The technique of recording data dependencies to keep track of the connection between inferences has been around for a long time. It has been used to assist in reconfiguring deductive retrieval data bases [Davis 82], to provide explanations of a program's behavior [Swartout 83], and to monitor the continued validity of abductive hypotheses to direct search in reasoning about physical devices [Stallman 79] [deKleer 84] [Forbus 84]. Some systems use the dependencies simply to record a trace of the deduction. Various programs can then use this trace to answer questions, make further inferences, debug the data base, or whatever. Data dependency records can also provide the basis for some general deductive strategies. In such strategies the data base and the dependency records are quite closely tied to one another. The content of the data base is often a matter of interpretation in that the data base may contain a record of a proposition, but only to note that the proposition is currently not believed and that some other proposition depends upon it not being believed.

Usually the data base, the dependency records, and the deductive strategy are tied together in a system for managing the items stored in the data base. Such systems are said to perform "reason maintenance" in that they maintain some useful invariant (or set of invariants) with respect to the contents of the data base. An example of an invariant might be that for all propositions for which there exists a record in the data base, either the proposition or its negation is marked as "true in the world". Now obviously by itself this invariant wouldn't be of much use, but coupled with some sort of deductive invariant (*e.g.*, enforcing modus ponens: if the beliefs corresponding to `P` and `(if P then Q)` are both true in the world then `Q` is true in the world) it might form the basis for performing propositional deduction [McAllester 80].

There are a number of reason maintenance systems in use today [deKleer 84] [Doyle 79] [McAllester 80] [McDermott 83]. The system described here is similar to that of Doyle [Doyle 79]. The system manipulates a data structure referred to as a *data dependency network*. This network is (at least conceptually) separate from the data structure used by the data base system to store and index data (*e.g.*, a discrimination net). The network consists of nodes called *ddnodes* (for data dependency node): one node for each datum of interest to the user.

The ddnode and the item it refers to are generally spoken of interchangeably. The paradigmatic case is the ddnode associated with each assertion in the database, but, in general, any explicit belief of the program must be associated with a ddnode, and any ddnode must have a *propositional content* that the program either believes or does not at any given time. Connecting all the program's beliefs in a data dependency network allows the RMS to enforce consistency among the beliefs. The connections between various items of data in the network are described in terms of *justifications*. A justification for a ddnode n_0 consists of a conjunction of other ddnodes in the following form: $(\{n_1, ...n_i\}\{n_{i+1}, ...n_j\})$ where n_1 through n_i are referred to as *in-justifiers* and n_{i+1} through n_j are called *out-justifiers*. An assertion corresponding to a ddnode with the empty justification (*i.e.*, $(\{\}\{\})$) is said to be a *premise*; it's believed unconditionally. Justifications are used to capture conditions for belief and thus the node n_0 is said to depend upon belief in $n_1, ..., n_i$ and absence of belief in $n_{i+1}, ..., n_j$. As an example suppose that n_1 corresponds to the belief that the house at 129 Rochambeau Avenue is located within the area known as East Side Providence, n_2 to the belief that houses in East Side Providence generally increase in value, n_3 to the belief that the house at 129 Rochambeau is structurally unsound, and n_0 to the

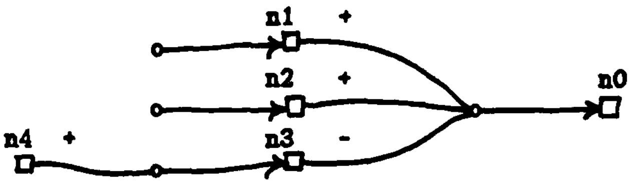


Figure 3.2: Example dependency diagram

belief that the house at 129 Rochambeau is a good investment. If n_0 has the justification $(\{n_1, n_2\}\{n_3\})$, then this would be interpreted as saying: If I believe that houses on the East Side have high resale value, and the house at 129 Rochambeau is in the East Side and as far as I know has no structural defects, then I believe that the house is a good investment.

In the graphical representation of dependency networks, ddnodes are shown as boxes and a justification is depicted as circle with lines drawn from the circle to the justifiers (the lines are labeled + and - for in-justifiers and out-justifiers respectively) and an arrow pointing to ddnode being justified. To complicate the example involving the house at 129 Rochambeau, suppose that n_1 and n_2 are premises and n_3 has the justification $(\{n_4\}\{\})$ where n_4 has no justifications and corresponds to the belief that the house at 129 Rochambeau has severe termite damage. Figure 3.2 shows the resulting dependency diagram for the network involving the house at 129 Rochambeau.

In addition to recording the reasons for believing something, it is also important to keep track of which things are currently believed. Deductions (such as forward and backward chaining) are generally performed using only items (rules and ground assertions) that are believed. Also it is often critical that the program using the reason maintenance system be notified of specific changes in the status of selected beliefs. Each ddnode has associated with it a *label* which is used to keep track of the status of the corresponding datum. In a Doyle-type TMS, the label for a ddnode is a boolean value, one of IN or OUT. IN meant that the corresponding datum is believed and OUT meant that it's not believed. In the latter case the datum associated with the ddnode was essentially hidden from the deductive retrieval machinery; it was, from the user's point of view, not present in the data base. All premises are given the label IN. A ddnode with no justifications is said to be an *assumption* and is given the label OUT. The labels for all other ddnodes have to be computed. The primary objective of a reason maintenance system is to find a *consistent* and *well-founded* assignment of statuses (IN or OUT) to all the ddnodes in the network which are neither premises nor assumptions. Intuitively a status assignment for a node is consistent if it follows from its justifications. In Doyle's system this requires that the label for a ddnode be IN iff at least one of its justifications is composed of in-justifiers with IN labels and out-justifiers with OUT labels. Intuitively a status assignment is well-founded if every IN ddnode can be shown to be grounded in premises. A ddnode with label IN is grounded (in premises) if it is a premise or it has at least one justification such that each in-justifier is either grounded or a premise and all the out-justifiers are labeled OUT.

When a justification is added or removed, the consequences of the change have to be propagated throughout the data dependency network. This involves recomputing the labels of some subset of the set of all ddnodes. The algorithm used in the Doyle RMS is discussed in [Charniak 80] and I won't repeat the discussion here. Suffice it to say that this algorithm finds a consistent and well-founded assignment of statuses (IN or OUT) for all ddnodes in the network, and does so in an efficient manner. In the dependency network shown in Figure 3.2, the nodes n_0 and n_3 have to be computed. As I have described the situation they would be assigned IN and OUT respectively. If n_4 ever became IN for some reason, then n_3 would become IN and n_0 would become OUT. The algorithm for updating the dependency network also allows us to determine which ddnodes have changed status as a result of the most recent modification to the dependency network. The indexing machinery is responsible for seeing to it that all justifications for believing a given datum refer to a unique ddnode. One reason for uniquely identifying beliefs is to simplify responding to changes.

Each data dependency node serves as a location in which to store responses to specific changes in the status of that node. These responses, called *signal functions*, are executable objects which are "run" whenever the label for the ddnode changes in some predefined way. The ddnode gives you a handle on a given belief. If the dependency network is modified in any way, the RMS can easily determine the set of ddnodes that might possibly have changed status. It can then check to see which of those ddnodes have changed status and what the nature of those changes are by comparing the newly computed label with the previous one. Signal functions are used for implementing what are called *change-driven interrupts* to support a style of programming that has been quite useful in developing the TMM. I'll return to this shortly.

The reason maintenance system actually employed by the TMM was developed specifically for reasoning about alternatives. It is based on deKleer's assumption-based approach [deKleer 84] and McDermott's update algorithm for handling contexts [McDermott 83]. This hybrid RMS behaves identically to Doyle's in the absence of what are called *gating objects*: special data structures used for reasoning about alternatives and disjunctions. In this preliminary discussion, I have chosen to focus on the functionality of the simpler Doyle-type system. In the hybrid RMS, labels are somewhat more complicated. I will introduce further complications as they are required.

There is no way to directly force disbelief in a datum in the face of justifications to the contrary. That is to say, the only way to guarantee not believing in something is to remove the justifications supporting it; you can't override an existing justification. One can, however, force belief in the negation of some datum. The system being discussed here does not keep track of truth values (as does McAllester's system [McAllester 80]). It is quite content to have two ddnodes, one for P and a second for (not P), and record that both are currently believed. The system can be designed to notice and respond to such logical contradictions [Doyle 79] but that is not its primary purpose. If you are interested in performing propositional deduction, then McAllester's system is the better choice as it was designed expressly for this purpose. The need for such a deductive capability will come up in our discussion of the implementation of the hybrid RMS (see Section 4.6 in Chapter 4), and I'll speak a bit more about propositional deduction at that time.

Data dependency systems of the sort we are discussing here are used for incrementally updating a set of beliefs upon the addition or removal of new beliefs or justifications, and then noticing and responding to specific changes in those beliefs. In principle there is no need to remove ddnodes as their corresponding beliefs can be rendered ineffectual by simply manipulating justifications. For example, if you want to ensure that you don't believe P, then remove all the justifications from the ddnode corresponding to P. As was evident from the example in Figure 3.2, it is often necessary to keep around nodes corresponding to assertions that are OUT just in case later deductions cause those assertions to become IN. In practice most systems employ some strategy for getting rid of ddnodes that have no bearing on beliefs the user is concerned with. This is done, however, to reclaim storage and not as a means of manipulating belief.

To review:

- The data dependency system can be thought of as a means of maintaining a set of beliefs.

- One is generally interested in detecting and responding to specific changes in a set of beliefs.

To make this concrete we have to be clear about:

- how the connections between beliefs (justifications) are set up
- how the "belief" status of a datum might change
- how you go about detecting and responding to such changes

The next few subsections will attempt to answer these questions, thus paving the way for an example illustrating how this functionality might be put to use. The discussion will be somewhat informal. The object is to motivate. For more detailed discussion about data dependencies in general see [Doyle 79] or [Charniak 80].

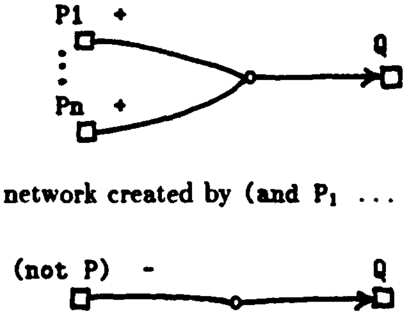
3.2.4 Establishing inferential connectivity

I've already mentioned two methods for performing deduction in deductive retrieval systems: forward and backward chaining. In addition to these, DUCK allows a third type of deduction which I'll refer to as *program-mediated* deduction. As I suggested above, Forward and backward chaining always occur in some sort of "context", including a set of variable bindings. Program-mediated deduction involves the use of programs that directly manipulate this context, setting up data dependency justifications, modifying variable bindings, and controlling forward inference. Before I explain program-mediated deduction, I want to explain what these "contexts" are and how they are created and employed during backward and forward chaining.

In the TMM, the context of a deduction is an object of data type ANS (for "answer") that consists of a set of variable bindings and a set of ddnode/support-type pairs associated with the steps in the deduction thus far. For our purposes, a support-type is just one of the set {+,-}, where + means that the deduction depends upon the ddnode being believed, and - means that the deduction depends upon the ddnode not being believed. This set of ddnode/support-type pairs attempts to capture why or under what conditions the current answer should be believed to be true. Figure 3.3 shows how ddnode/support-type pairs are incorporated into the answers returned by the function `fetch`. The machinery responsible

```
If the data base consists of:

 ddnode:  corresponding datum:
   n1        (string-quartet Juilliard)
   n2        (string-quartet Guarneri)
   n3        (cellist Joel-Krosnick Juilliard)
   n4        (cellist David-Soyer Guarneri)

then:
  (fetch '(and (string-quartet ?q)
               (cellist ?c ?q)))

returns:
  (({(q Juilliard) (c Joel-Krosnick)} {(+ n1) (+ n3)})
   ({(q Guarneri) (c David-Soyer)} {(+ n2) (+ n4)}))
```

Figure 3.3: Answers containing data dependency information



a. Dependency network created by (and P_1 ... P_n (assert Q))

b. Dependency network created by (and (consistent (not P)) (assert Q))

Figure 3.4: Dependency diagrams for backward chaining examples

for forward and backward chaining sees to it that the *current answer* (the value of ans*, a global variable of data type ANS) is modified to reflect the current state of the deduction. When an assertion occurs, the system finds or creates the unique ddnode associated with that assertion. It then installs in that ddnode a new justification formed using the set of ddnodes in the current answer (in-justifiers consisting of ddnodes of support type + and out-justifiers those of support type -). Assertions are allowed to occur during backward chaining. So in backward chaining on the conjunctive goal (and P_1 ... P_n (assert Q)), a system handling data dependencies should ensure that the ddnode corresponding to Q has a justification that depends upon on the ddnodes associated with deducing P_1 ... P_n (Figure 3.4.a shows the dependency network constructed assuming that each of P_1 through P_n correspond to ground assertions in the data base). There is also the need to handle queries that make use of the "consistent" (nonmonotonic negation-as-failure) operator. So assuming that P cannot be deduced from the current contents of the data base, a query of the form (and (consistent (not P)) (assert Q)) should succeed, resulting in the ddnode associated with Q having a justification with an out-justifier corresponding to the ddnode associated with P (see Figure 3.4.b). Figure 3.5 provides a somewhat frivolous example[3] illustrating how the system sets up dependencies during backward chaining. Forward chaining behaves similarly. If P is asserted and there is a rule (-> P Q), then the ddnode associated with Q should have a justification which includes the ddnodes for P and (-> P Q).

"Program-mediated deduction" refers to deduction done inside code using language constructs for forward and backward chaining (*e.g.*, fetch and add). When an assertion is made, it is justified, intuitively, because the process has reached a certain point in the program, with the variables bound in a certain way. Often, it is possible to identify explicit reasons for the process to have reached this point, such that an assertion made at this point ought to depend upon those reasons. Consider the following code executed with the data base of Figure 3.6:

```
(cond ((fetch '(and (string-quartet ?q)
                    (list-of-members ?q ?l)
                    (member Robert-Mann ?l)))
       (add '(classical-musician Robert-Mann))))
```

[3] This example was inspired by a maxim which I first heard from Ken Forbus: namely, that the true test of sophistication for an urban center is an eatery that sells cappuccino at reasonable hours and lots of readily available computing power. It could be that his statement was made in jest but its predictions often agree with those made by other independent measures of sophistication.

```
Suppose that the data base contains just the following:

 ddnode:  corresponding datum:
  n1       (<- (civilized ?urban-area)
               (and (instance-of ?establishment restaurant)
                    (located ?establishment ?urban-area)
                    (serves ?establishment cappuccino)))
  n2       (instance-of Consiglios restaurant)
  n3       (located Consiglios NewHaven)
  n4       (serves Consiglios cappuccino)

If the following conjunctive goal is encountered in backward chaining:
  (and (civilized ?place)
       (consistent (available ?place significant-computing-power))
       (assert (sophisticated ?place)))
 then the system should create a new ddnode n5 corresponding to:
  (sophisticated NewHaven)
 with justification:
  ({n1, n2, n3, n4}{n6})
 where n6 corresponds to the ddnode:
  (not (available NewHaven significant-computing-power))

The resulting dependency network is:
```

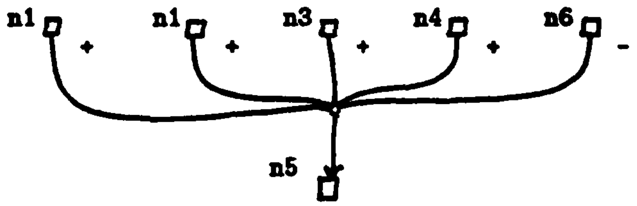


Figure 3.5: Constructing a justification during backward chaining

```
Data base contents:

 ddnode:    corresponding datum:
   n1         (string-quartet Juilliard)
   n2         (list-of-members Juilliard !<Earl-Carlyss Robert-Mann
                                          Samuel-Rhodes Joel-Krosnick>)
   n3         (<- (member ?x !<?y !& ?z>) (or (:= ?x ?y) (member ?x ?z)))
```

Figure 3.6: Data base for demonstrating program-mediated deduction techniques

The assertion (classical-musician Robert-Mann) is added to the data base *because* Robert Mann is known to be a member of the Juilliard string quartet. It should be made dependent upon the assertions (string-quartet Juilliard) and (list-of-members Juilliard !<Earl-Carlyss Robert-Mann Samuel-Rhodes Joel-Krosnick>) and the rule used by the deductive system to determine that one term is a member of a list of terms. Objects of data type ANS provide us with the means for keeping track of the reasons why a process has reached a certain point in a program. The following:

```
(fetch '(and (string-quartet ?q)
             (list-of-members ?q ?l)
             (member Robert-Mann ?l)))
```

would return:

```
(({(q Juilliard)
   (l !<Earl-Carlyss Robert-Mann Samuel-Rhodes Joel-Krosnick>)}
  {(+ n1) (+ n2) (+ n3)}))
```

in the data base of Figure 3.6.

All deductions take place in the context of the current answer ans*. The answers returned by a "fetch" are said to *augment* the current answer. If we want an assertion to be dependent upon a particular "fetch", then we will have to bind ans* to be the value of some augmented answer returned by the "fetch". The *scope* of an answer refers to the time during which that answer is the value of ans*. For our example involving the Juilliard quartet, the dependencies would be handled correctly by the following:

```
(let ((answers (fetch '(and (string-quartet ?q)
                            (list-of-members ?q ?l)
                            (member Robert-Mann ?l)))))
  (cond (answers
         (bind ((ans* (car answers)))
               (add '(classical-musician Robert-Mann))))))
```

That is to say, the system will create a ddnode corresponding to (classical-musician Robert-Mann) with justification ({n1, n2, n3}{}).

The above code can be simplified using various LISP macros designed for that purpose. In a call of the form (for-first-answer (fetch *query*) *code*), if the list returned by the fetch is empty, the call returns with (); if the list is not empty, then ans* is bound (locally) to the first augmented answer in the list and the code is executed returning whatever the code returns. In (for-each-answer (fetch *query*) *code*), the current answer is repeatedly bound to as many augmented answers as the fetch will allow, and the code is executed for each answer. The for-each-answer macro is generally used as follows:

```
(for-each-answer (fetch  some-query)
     assorted-LISP-code
    (add   some-assertion)
     more-assorted-LISP-code)
```

Each answer in the list returned by fetch is bound to ans* in turn, and add is called. Add makes the necessary substitutions and sets up the requisite dependencies. Suppose that the data base contains just (string-quartet Juilliard) and (string-quartet Guarneri) and the following code is executed:

```
(for-each-answer (fetch '(string-quartet ?q))
        (add '(repertoire ?q Mozart)))
```

Following execution, the data base will also contain (repertoire Juilliard Mozart) and (repertoire Guarneri Mozart), such that (repertoire Juilliard Mozart) depends upon (string-quartet Juilliard), and (repertoire Guarneri Mozart) depends upon (string-quartet Guarneri).

Notice that under certain conditions, the above code performs the same function as the forward chaining rule (-> (string-quartet ?q) (repertoire ?q Mozart)). The main difference concerns timing. In the case of the for-each-answer version, if after executing the code you add (string-quartet Melos), then (repertoire Melos Mozart) will not be added to the data base. There are many cases in which this is exactly what you want. For example, suppose that the data base is consistent with several hypothetical situations, only one of which can actually occur. In such a case, the program will have to be careful to add only those assertions that follow from the hypothesis selected as best[4]. As we'll see, this sort of carefully controlled deduction plays an important role in temporal reasoning. I will reserve the term *controlled forward inference* to refer to that class of deductions characterized by the pattern (for-each-answer (fetch *antecedent-conditions*) (add *consequent-predictions*)).

In the data base shown in Figure 3.5, the following two LISP fragments result in setting up exactly the same dependencies.

[4] The program might also choose to explore several alternatives at the same time using a context mechanism [McDermott 83], but the deductions still have to be tightly controlled in selecting a set of candidate hypotheses and keeping their corresponding consequences separate.

```
(fetch '(and (civilized ?place)
             (consistent (available ?place significant-computing-power))
             (assert (sophisticated ?place))))

(for-first-answer
  (fetch '(and (civilized ?place)
               (consistent (available ?place significant-computing-power))))
  (add '(sophisticated ?place)))
```

In this case, there really is no advantage to using one rather than the other. If, on the other hand, a significant amount of computation is more conveniently or efficiently carried out in LISP, then it is often advantageous to use constructs like `for-each-answer` to jump back and forth between LISP and the deductive retrieval routines provided by DUCK. In general, program-mediated deduction provides a clean and versatile means for carefully controlling deduction, while the logic-programming notation is more concise in relatively simple situations. For temporal reasoning, such simple situations are rare, so most of our examples will employ program-mediated deduction.

There are also ways of selectively modifying just the set of ddnode/support-type pairs in the current answer. The forms `(answer-support (+ '(foo bar))` *code*`)` and `(answer-support (- '(foo bar))` *code*`)` are used to construct a new ANS formed from the bindings of the current answer and the ddnode/support-type pairs of the current answer, plus a new pair consisting of the ddnode associated with `(foo bar)` along with the support types `+` and `-` respectively. The current answer is then bound locally to this augmented answer and the code executed.

These techniques and functions constitute the primary method for the user to construct new justifications (of course one can also get down and grub around in the data structures of the underlying data dependency system, but that is generally discouraged). The user still needs some means of retracting previous justifications. As I am not aware of a particularly clean interface for selectively removing justifications, we will rely upon a single rather gross but effective method for modifying dependency structures: erasure. Erasing an assertion (and indirectly its associated ddnode) is accomplished by removing all of that ddnode's justifications; effectively removing any warrant for belief in the assertion. Erasure is the normal means for forcing disbelief in a particular assertion. A call of the form `(erase '(foo bar))` will find the ddnode corresponding to `(foo bar)` and remove all of its associated justifications.

Whenever forward chaining occurs, either by using rules of the form `(-> P Q)` or via other means of making assertions, the RMS is invoked to (re)compute the status of all possibly affected ddnodes. Now we can consider how to go about detecting and responding to changes in the status of ddnodes that occur during this process of updating.

3.2.5 Detecting and responding to changes

There are two main issues concerning the detection and response to changes in the status of ddnodes. The first concerns what sort of changes one has to be aware of. The second issue concerns the order in which the responses should be made. If there are several changes to be dealt with the order can be quite critical.

In a Doyle-type system, the sort of changes that can occur are rather simple. A ddnode label can toggle between IN and OUT and that's the extent of it. (In reasoning about alternatives things get considerably more complicated but that can wait until Chapter 4).

The other issue in responding to changes concerns the order in which responses are made. It is often the case that certain operations on the data base cannot be reliably or efficiently performed unless the data base satisfies some property. It is generally quite difficult to predict the extent of the changes that will result from making modifications to a dependency network. The signal functions for ddnodes whose labels changed are fired nondeterminisitically. Suppose that one ddnode has a signal function responsible for sounding an alarm in the event that the ddnode becomes IN. Another ddnode has a signal function which checks for equipment malfunctions and corrects the malfunction if possible. Suppose that during an update both ddnodes come IN but the execution of the second signal function would serve to detect an equipment malfunction, fix it, and update the network in such a way that the alarm ddnode would go OUT. If the malfunction checking signal function is called first, and the alarm signal function checks to see if it is still warranted, then the alarm will never sound. Assuming that you don't want to be bothered with alarms caused by faulty (but repairable) equipment, the order in which signal functions are processed is critical.

One way of getting the timing to work out involves the use of priority queues. Each priority level is associated with a set of executable objects (*e.g.*, closures) such that when an object at one level is executed it can assume that all objects with higher priority have already been executed. The priority levels can be associated with data base properties, called *invariants* (see David McAllester's RUP system [McAllester 82] for a discussion of the use of invariants in reason maintenance). In the example above "all repairable equipment malfunctions accounted for" might be a useful invariant to maintain. And obviously it should be of a higher priority than "alarm sounded if malfunction outstanding". In describing the time map management system, I will introduce a number of invariants. In addition, the user is also given a range of priorities that are for his sole use. The time map invariants are of higher priority than any user invariants in order to discourage the user from making deductions on the basis of a partially updated time map.

Each signal function has associated with it a priority level and a condition which must be true in order to warrant the execution of the signal function. In the RMS update algorithm, once the status assignments have been recomputed, all ddnodes that have changed status have their signal functions checked. Any signal functions whose conditions are satisfied are placed in the priority queue at the appropriate level. When the update is completed all the objects in the queue are executed in such a way that an object can assume that all objects of higher priority have been executed. Just before execution the condition is checked once more to ensure that execution is still warranted. I'm not going to describe how one sets up and installs signal functions for invariants. The details are not important to our discussion. The temporal reason maintenance algorithm is based upon the maintenance of a number of specific invariants. The necessary signal functions and their corresponding priorities will be detailed in Chapter 4.

DUCK provides utilities to enable the user to tell the system exactly what to do when an assertion toggles from IN to OUT or alternatively OUT to IN. The resulting programs are called *change-driven interrupts* and are implemented by attaching signal functions to the ddnodes corresponding to the assertions of interest. The simplest form of change-driven interrupt can be implemented using what are called *if-erased demons* [Sussman 71] [McDermott 85]. An if-erased demon is just a piece of code that is executed whenever an assertion becomes OUT. The following illustrates how to set up an if-erased demon designed to perform the delicate operation of editing a corporation's christmas card mailing list.

```
(if-erased '(employee-record ?person ?job-description)
  (for-first-answer
   (fetch '(and (thnot (employee-record ?person ?alternate-job-description))
                (christmas-card-recipient ?person)))
   (erase '(christmas-card-recipient ?person))))
```

If any assertion corresponding to an employee's job record ever becomes OUT, then a signal function attached to the assertion is executed. This function checks to see that the employee is currently on the christmas mailing list and doesn't have have another job with the company, and if so, effectively removes the employee from the list. If-erased demons will suffice for dealing with any changes we will have to handle in this chapter. You can assume that the signal functions associated with if-erased demons have lower priority than any of the temporal reason maintenance invariants (and hence are executed only after all temporal invariants are satisfied).

3.2.6 Example application

It's probably worth considering a simple example illustrating how one might put to use the functionality of data dependencies and signal functions. Suppose that we need a program for detecting and diagnosing problems in a chemical plant. A dependency network could be designed to model production at each stage in the chemical process. The dependencies would capture certain expected states and their implications, and the system would rely upon outside sensors to assist in updating the current state of the process. Signal functions would be attached to ddnodes representing certain selected states. These states might signify potentially dangerous situations warranting action of some sort. The signal functions would enable the system to alert the authorities, or perhaps even to instigate corrective measures on its own initiative.

To keep the program up to date there would be some number of routines responsible for sampling production parameters and updating the dependency network to reflect new values. Quantitative measures could be approximated using qualitative states [Forbus 84] reflected in the status of dependency nodes. Instead of saying `(level contact-reactor4 41)`, you might have a ddnode corresponding to `(>= (level contact-reactor4) safety-threshold)`. In the event that the level in `contact-reactor4` went from 34 cm to 45 cm, the ddnode `(>= (level contact-reactor4) safety-threshold)` might toggle from OUT to IN. The dependency network would perform like a simulation of the physical system. Signal functions would be attached to various nodes with instructions about how to respond in certain situations.

Suppose that the chemical process involves the generation of synthetic ammonia. Then the dependency network might encode a rule like: If the pressure in `contact-reactor4` is in the normal operating range and the backpressure at `circulation-pump17` is below 2 atm., then suspect inadvertent venting of ammonia gas to the atmosphere. If the node corresponding to "inadvertent venting of ammonia gas" ever came IN, then a signal function could be used to issue a warning, close a valve, or instigate some other response.

3.2.7 Review

We now have a deductive retrieval system capable of keeping track of what is believed and why. A fact is believed if it is labeled IN and not believed if it is labeled OUT. "Why" is recorded in terms of justifications. DUCK provides a number of mechanisms for setting up and installing justifications, and the RMS efficiently recomputes the labels of facts affected by the new justifications. We also explored techniques for responding to changes in the status of facts in the data base. The idea of a priority queue mechanism for maintaining a set of invariants was adopted from McAllester. This idea will figure prominently in the algorithms for temporal reason maintenance described in Chapter 4. In the following sections I will show how the functionality described in this section can be extended to reason about events and their effects changing over time.

3.3 Basic temporal notions

As mentioned in the previous chapter the principal items in our ontology are points, intervals, and time tokens. Of these only the first and the last are data types (respectively POINT and TOKEN) in the TMM. An object of data type POINT designates an instant in time. That instant needn't be completely specified in any particular frame of reference. That is to say the system needn't be able to specify the precise location of a particular point on the time line laid down by a given clock. Every point is itself a frame of reference. I may know that exactly five minutes after starting the coffee maker this morning I sat down at the kitchen table to read the paper. I also know that I started the coffee maker sometime after being disturbed from a sound sleep by a wrong number. The time between waking and sitting down at the table to read the paper is only approximately known. This is represented as a fuzzy number [McDermott 84]: an upper and lower bound on the distance separating two points in the time map. The time map consists of points linked by *constraints* which bound the distance (in some units of time) separating pairs of points. The following function is used to denote the distance between two points.

```
(define-function (distance FIXNUM (POINT POINT)))
```

We want to be able to add new constraints and determine what is known about the distance separating two points in the time map. The function **distance** allows us to refer to the distance separating two points in the time map. We now need a set of predicates so that we can retrieve and modify information about such distances. The time map is designed to efficiently compute the best bounds on the distance separating a pair of points in the time map (see Chapter 4 for details). I'll refer to those bounds as the GLB (greatest lower bound) and LUB (least upper bound). For our purposes a constraint between two points can be viewed as a directed edge labeled with an upper and lower bound on the distance between the points. A path from **pt1** to **pt2** through the time map consists of a sequence of points and constraints starting at **pt1** and ending at **pt2**. Each path determines a pair of bounds (lower and upper) on the distance between the two points. The lower (upper) bound of a path is computed by taking the sum of the lower (upper) bounds of the individual constraints in the path. The GLB (LUB) on the distance separating two points is equal to the greatest (least) of the lower (upper) path bounds given all paths connecting the two points. The predicate **strict-elt** (for "strictly an element of") is used to determine the best known (strictest) bounds on the value of terms denoting integers (**FIXNUMs**). In general, (**strict-elt** *trm*1 *trm*2 *trm*3) means that *trm*1 is bounded from below by *trm*2 and from above by *trm*3, and that there are no better bounds known. The predicate declaration is simply:

```
(define-predicate (strict-elt FIXNUM FIXNUM FIXNUM))
```

So, for instance, the query (**strict-elt (distance pt1 pt2) ?low ?high**) will return with **?low** and **?high** bound to the GLB and LUB respectively. As was mentioned in Section 3.2.1, a **FIXNUM** is either an integer or one of a small set of special symbols. In the TMM, there are four symbolic **FIXNUMs**. The symbols ***pos-inf*** and ***neg-inf*** denote numbers (respectively) larger and smaller than any other number. In addition, ***pos-tiny*** (***neg-tiny***) denotes a number greater (less) than 0 but less (greater) than any positive (negative) number. The only arithmetic operations used by the TMM involving symbolic **FIXNUMs** are addition and subtraction and these are quite straightforward. If **pt1** and **pt2** are unconstrained and **?low** and **?high** are unbound, (**strict-elt (distance pt1 pt2) ?low ?high**) will succeed with substitution {(**low *neg-inf***) (**high *pos-inf***)}.

Next I'll introduce a similar (though less restrictive) predicate called simply **elt**.

```
(define-predicate (elt FIXNUM FIXNUM FIXNUM))
```

The predication (elt *trm*1 *trm*2 *trm*3) means that *trm*1 is bounded from below by *trm*2 and from above by *trm*3. A query of the form (elt (distance pt1 pt2) 6 9) will succeed just in case there exists a path from pt1 to pt2 with lower bound greater than or equal to 6, and upper bound less than or equal to 9. Elt reports on whether or not a set of bounds are supported by the current set of constraints in the time map. It turns out that this is not all that useful. Most of the time what we want to know is whether or not a set of bounds is consistent with the constraints in the time map. In the next section I'll discuss this in greater detail.

For the sake of convenience, I'll define a set of predicates, pt=<, pt<, pt>, pt>=, and pt=, which serve as shorthand for various uses of elt.

```
(define-predicate (pt< POINT POINT))

(<- (pt< pt1 pt2)
    (elt (distance pt1 pt2) *pos-tiny* *pos-inf*))

(define-predicate (pt=< POINT POINT))

(<- (pt=< pt1 pt2)
    (elt (distance pt1 pt2) 0 *pos-inf*))
```

The other definitions are quite similar, and I won't bother with them.

So far we only know how to ask questions about the distance separating points in the time map. We need to be able to add information as well. You can add a new constraint to the time map by asserting something of the form (elt (distance pt1 pt2) *low high*) where *low* and *high* are FIXNUMs. This supplies an upper and lower bound (not necessarily the least upper or greatest lower) on the distance separating pt1 and pt2. It may also change the strict bounds (GLBs and LUBs) on distances separating other pairs of points in the time map, since the new constraint can be used to compute new paths between pairs of points besides those that it directly refers to.

An assertion of the form (elt (distance pt1 pt2) *low high*) does not affect existing constraints. Indeed, it may add no new information to the time map if there are already better bounds on the distance between the points. The same constraint can be added several times, each time with a different justification. Only by undermining all justifications for a given constraint can its influence be removed from the time map.

Now that we can add constraints between points and determine information about the distance separating points, it would be nice to be able to create new points. What we're really interested in is the ability to create entities corresponding to events and facts. These entities will refer to time points, but it is the entities themselves that are of primary interest.

An interval is just a pair of points. Certain intervals, however, are of more interest than others, and for these we generally supply some name or description. Quite often such intervals correspond to the occurrence of events or a span of time separating two events. An interval might also be associated with a particular instance of a fact becoming true and enduring over a period of time. All such "distinguished" intervals share the data type[5] TOKEN. TIME-TOKEN might be more appropriate but the shorter version is more convenient and there should be no ambiguity in these pages. There are two obvious functions associated with TOKENs, namely their beginning and end points.

```
(define-function (begin POINT (TOKEN)))
```

```
(define-function (end POINT (TOKEN)))
```

In addition, we define a predicate for going from TOKENs to their descriptions (PROPs) and back.

```
(define-predicate (time-token PROP TOKEN))
```

Given a query of the form (time-token (ingest-caffeine self) ?tok), the TMM would return a list of answers in which ?tok is bound to a TOKEN with a description (ingest-caffeine self) (also referred to as the token's type). Asserting (time-token (ingest-caffeine self) tok1) would add to the time map a new object tok1 of data type TOKEN with the given schema. In a conjunctive assertion such as:

```
(assert  (and (time-token (lunch self cafeteria) ?tok)
              (elt (distance (begin ?tok) (end ?tok)) 15 20)))
```

in which the variable ?tok is unbound, the TMM will create a new object corresponding to a TOKEN with schema (lunch self cafeteria), and bind ?tok to that symbol. When (elt (distance (begin ?tok) (end ?tok)) 15 20) is encountered, the constraint will be added between the beginning and ending of the newly created TOKEN[6].

[5] In this chapter when we speak of types we are referring to data types. This is not to be confused with the distinction made in logic between types and tokens. An object of data type TOKEN is meant to denote a particular instance of a class of events or facts described by that object's schema. Said in another way, an object of data type TOKEN denotes a (logical) token which is an instance of the (logical) type characterized by its schema.

[6] The correct way to handle the variables in this case would be to use an existential quantifier (*e.g.*, (exists (tok) (and (time-token P tok) (elt (distance (begin tok) (end tok)) 15 20)))) and have

The previous chapter discussed at some length the intuitions surrounding distinctions between facts and events, and I won't belabor the point here. Suffice it to say that these distinctions don't carry over into the data types in the implementation. The time map makes no internal distinction between TOKENs interpreted as events and those interpreted as facts. A distinction, however, is implied in the tokens and rules that the user adds to the data base. Certain fact types can be defined to contradict other fact types. Two tokens are said to be contradictory if their corresponding types are determined to be contradictory. Such tokens are treated as fact tokens (*i.e.*, persistences) by the TMM. Contradiction criteria are defined using the `contradict` predicate.

```
(define-predicate (contradict PROP PROP))
```

Example criteria are:

```
(contradicts (not ?p) ?p)
and
(<- (contradicts (color-of ?obj ?color1) (color-of ?obj ?color2))
    (thnot (= ?color1 ?color2)))
```

As far as the time map is concerned, any token found to be in contradiction with another token is treated as a fact. The TMM performs no additional interpretation. It does, however, perform an important service involving contradictory tokens.

If one token is determined to be in contradiction with a second, and the first is constrained to begin before the second begins, then the first is constrained to end before the second (assuming such a constraint is consistent with the existing constraints). This is referred to as the *rule of persistence*, and while the rule is quite simple in form, it figures prominently in a number of important assumptions that are deeply woven into the fabric of time maps as used in the TMM.

The rule of persistence is used to modify the duration of fact tokens. If I assert that the light in the attic is on, then, lacking information to the contrary, I am quite willing to believe it will stay on indefinitely. This willingness on my part to believe that certain facts persist indefinitely if unmolested, corresponds to there being no upper bound on the duration of the associated token (or perhaps an upper bound equal to the estimated life of the light bulb). I am also open to the possibility that subsequent information will lower

the deductive retrieval machinery Skolemize the formula at the time the assertion is made. I apologize to any purists offended by the above hack.

that upper bound (*e.g.*, someone fetches something from the attic and turns the light off). This readiness to terminate the duration of a fact in the face of evidence of its extinction corresponds to the lower bound on the token's duration being zero (or at least very small). The rule of persistence is used to remove apparent but resolvable contradictions.

```
(<- (apparent-contradiction ?tok1 ?tok2)
    (and (time-token ?p ?tok1)
         (time-token ?q ?tok2)
         (contradicts ?p ?q)
         (strict-elt (distance (begin ?tok1) (begin ?tok2)) ?blow ?bhigh)
         (=< 0 ?blow)
         (strict-elt (distance (begin ?tok2) (end ?tok1)) ?elow ?ehigh)
         (=< 0 ?ehigh)))
```

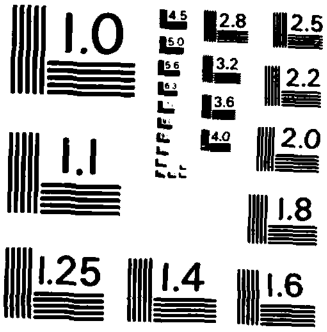
Two apparently contradictory tokens can be resolved if it is possible that the one beginning earlier can end before the later.

```
(<- (resolvable-contradiction ?tok1 ?tok2)
    (and (apparent-contradiction ?tok1 ?tok2)
         (strict-elt (distance (begin ?tok2) (end ?tok1)) ?low ?high)
         (< ?low 0)))
```

Whenever the TMM discovers a resolvable contradiction, it removes it by adding a constraint to ensure that the earlier token ends before the later (the actual mechanism will be discussed in Chapter 4). Unfortunately, not all apparent contradictions are resolvable. There is no way to resolve apparently contradictory tokens with coincident beginning points. Also, there are situations in which it is reasonable to give a token describing a fact a duration with an upper bound greater than 0. If I notice the light being on for some time and later am told that it was not on during the period I observed it, then I should be disturbed. It would be wrong for the TMM to resolve the inconsistency without notifying someone of a problem. In such a situation the TMM will describe the problem and then prompt the calling program to remove one or more constraints in order to resolve the contradiction. The important decisions are left up to the calling program.

In applications using the TMM, the durations for event tokens are generally tightly constrained. Fact tokens are typically tied quite closely to the events that they are believed to be effects of. The end points of fact tokens are generally constrained only by contradictory tokens via the rule of persistence. There are exceptions of course (such as the example of having been witness to the light shining over an interval), but in the main it is reasonable to constrain the lower bound on the distance between the begin and end of a fact token

AD-A161 242 TEMPORAL IMAGERY AN APPROACH TO REASONING ABOUT TIME 2/3
FOR PLANNING AND PRO..(U) YALE UNIV NEW HAVEN CT DEPT
OF COMPUTER SCIENCE T DEAN OCT 85 YALEU/CSD/RR-433
UNCLASSIFIED N00014-83-K-0281 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

to be 0. In later sections we'll see how this default lower bound assists in reasoning about change in a temporal database.

Most of the rest of this chapter is dedicated to explaining how the information stored in time maps is to be interpreted and made use of in a principled manner. This task will be easier if the text is supplemented with diagrams displaying the information in a given time map. To this end I will devote the next section to describing some conventions for drawing and interpreting time-map diagrams.

3.4 Conventions for displaying time maps

There are no privileged frames of reference in a time map. Some, such as the frame of reference associated with local clock time, are more convenient than others for certain types of communication, but generally the context establishes what a good frame of reference is for a given reasoning task. While the time-map routines can use any frame of reference and jump between different frames of reference as easily as you jump between words, it is necessary for display purposes to settle upon a single frame of reference. If we're talking about a football game, then the beginning of the game or perhaps halftime is preferable to noon eastern standard time on the day the game was played. If a chemist is reasoning about chemical reactions, then the point where a catalytic agent is added to a solution might provide a good frame of reference. Unfortunately, there are times when a single frame of reference fails to provide adequate resolution for viewing a large number of events and effects. In such situations it is often necessary to display different sets of points with respect to different frames of reference. The various frames of reference often constitute some sort of a hierarchy, such as the refinement hierarchies discussed in the previous chapter. To make understanding time-map displays eas'^r for the reader, I have adopted a number of conventions that are observed by the TMM in generating "pictures" of the data base. These conventions are listed below and immediately followed by an example illustrating their use.

1. Displays will be separated into sections with each section beginning with a frame-of-reference line.

2. The frame of reference line includes the name of the point being used as a frame of reference (*e.g.*, `pt1` or `(begin tok1)`), a scale factor, and an arrow indicating where

the point is positioned in the display.

3. A point, pt1, is displayed using the frame of reference, pt2, as follows:

   (a) If pt1 is known with precision relative to pt2 (the lower bound on the distance separating the two points is equal to the upper bound), then it will be shown as a vertical bar "|" in the appropriate position[7], unless that position is off the page, in which case one of "<" or ">" will be used to indicate where it lies.

   (b) If the bounds on the distance separating pt1 and pt2 are distinct, then both the upper and lower bound are displayed separately using the conventions for precisely determined points. The resulting two markers are then linked using a squiggly line (*i.e.*, "~~~~~").

4. A token, tok1, is displayed using the frame of reference, pt2, as follows:

   (a) The token symbol is printed (in this case tok1) followed by the token's schema (*e.g.*, (puton cup1 shelf1)) on one line.

   (b) On a second line, the point (begin tok1) is displayed in the frame of reference of pt2 as above.

   (c) On the same line, the point (end tok1) is displayed using (begin tok1) as a frame of reference[8].

   (d) The markers corresponding to the upper bound of (begin tok1) with respect to pt2 and the lower bound of (end tok1) with respect to (begin tok1) are linked using a dashed line ("---")

   (e) If the upper bound on the distance from (begin tok1) to (end tok1) would extend the print line off the display, then a squiggly line is extended from the marker for the lower bound to the edge of the display and terminated with a ">". This makes clear in many cases the scope of a persistence.

The display generated by the above algorithm is generally quite simple to interpret. The more constrained the time map, the less ambiguity present in the display. To demonstrate

[7] The appropriate position is calculated as an offset from the position of the reference point in the display equal to the product of the scale factor and the distance separating the two points in the time map.

[8] The marker for the upper bound of (begin tok1) with respect to pt2 serves as the position of (begin tok1) in the display. This position is used to calculate the offsets for the marks displaying (end tok1)

we can use the predicates defined in the previous section to construct a time map for a simple scenario.

Suppose that it's now noon and I'm considering the events which transpired earlier in the morning. At 8:00, give or take 15 minutes, I entered room 301 of Dunham lab to discuss our planning project with three others working on the project. The meeting was to last between 2.5 and 3.5 hours. An hour into the meeting someone called in an order to a local restaurant for coffee and pastries to be delivered. The order arrived 20 or 30 minutes later. Five minutes after the coffee arrived someone bumped the table, spilling coffee the length of the table, and thereby soaking my pants which are still wet even as I sit contemplating the accident.

In the following, unknowns (uninstantiated parameters) appear as skolem constants (they should actually be skolem functions but ..), `*now*` is a `POINT` referring to the present moment, 12:00 AM, and minutes are the basic unit of time (`minc` is a conversion function that converts notations of the form hours:minutes into minutes). The steps or assertions shown for constructing this time map are exactly those used in the actual TMM. I have included them here (and elsewhere) because they provide a clear account of how one goes about using the TMM and, precisely what information one has to supply in order to get particular results.

The following assertions are added to the database:

```
(time-token (meet !<Mikhail Hubert Louise self> rm301dunham) meeting17)
(time-token (place-order sk42 order31) ordering12)
(time-token (deliver sk43 order31 rm301dunham) delivery45)
(time-token (shake sk44 table13) disaster3)
(time-token (spill coffee table13) spill72)
(time-token (wet self coffee) wet33)
(elt (distance (begin meeting17) *now*) (minc 3:45) (minc 4:15))
(elt (distance (begin meeting17) (end meeting17)) (minc 2:30) (minc 3:30))
(elt (distance (begin meeting17) (begin ordering12)) 60 60)
(elt (distance (begin ordering12) (end ordering12)) 2 4)
(elt (distance (end ordering12) (begin delivery45)) 20 30)
(elt (distance (begin delivery45) (end delivery45)) 3 5)
(elt (distance (end delivery45) (begin disaster3)) 5 5)
(elt (distance (begin disaster3) (end disaster3)) 1 2)
(elt (distance (end disaster3) (begin spill72)) 0 1)
(elt (distance (begin spill72) (end spill72)) 1 1)
(elt (distance (end spill72) (begin wet33)) 0 0)
(elt (distance (begin wet33) (end wet33)) 0 *pos-inf*)
```

```
Frame of reference: *now*  Scale: 0.3

meeting17 (meet !<Mikhail Hubert Louise self> rm301Dunham)
|‾‾‾‾‾‾‾‾|------------------------------------------|‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾|
ordering12 (place-order sk42 order31)
                  |‾‾‾‾‾‾‾‾|||
delivery45 (deliver sk43 order31 rm301Dunham)
                        |‾‾‾‾‾‾‾‾‾‾‾|||
disaster3 (shake sk44 table13)
                           |‾‾‾‾‾‾‾‾‾‾‾||
spill172 (spill coffee table13)
                           |‾‾‾‾‾‾‾‾‾‾‾‾||
wet33 (wet self coffee)
                           |‾‾‾‾‾‾‾‾‾‾‾‾||‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>

Frame of reference: (begin delivery45)  Scale: 1.6

ordering12 (place-order sk42 order31)
<‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾|--|‾‾|
delivery45 (deliver sk43 order31 rm301Dunham)
                                               |---|‾‾‾|
disaster3 (shake sk44 table13)
                                                           |‾‾‾||‾|
spill172 (spill coffee table13)
                                                             |‾‾‾‾‾||
wet33 (wet self coffee)
                                                               |‾‾‾‾‾‾||‾‾‾>
```

Figure 3.7: Example time-map display

Figure 3.7 shows two views of the resulting time map. The first displays all the tokens with respect to the current moment. The second uses the beginning of the delivery event as a frame of reference and shows just those events and facts concerned with the incident in which I was splashed with coffee. The difference in resolution is quite apparent in the two views. Unless the case being made specifically requires it, most of the displays to follow will deal in more precise point-to-point distances. However, it is important to note that reasoning about time invariably requires the ability to deal with inexact and ambiguous information. Interpreting time-map displays is difficult precisely because much of the information is hidden. While we are seldom aware of all of the ramifications of our temporal knowledge, we do manage to keep track of many important orderings and approximate durations. If the order of two events becomes crucial to our plans, we are quite adept at making do with whatever information we have, or gathering more if need be. The next two sections will show how the TMM manages similar feats.

3.5 Default reasoning in partially ordered time maps

The information in a time map is generally incomplete. Event tokens are partially ordered, constraints on their occurrence are specified as fuzzy intervals, and the only information typically available concerning the duration of persistences is an upper bound. In order to make use of time maps, it is important to adopt a consistent strategy for interpreting the information they contain. The interpretation strategy built into the operation of the TMM is quite simple. Events occur as early as they can (the lower bound on the duration of an event token is most indicative of its start time), and fact tokens persist as long as possible (the duration of a persistence is best estimated by the upper bound on the distance separating its begin and end points). According to this strategy a fact P is true throughout an interval just in case there exists a fact token of type P that begins before, or is coincident with, the beginning of the interval, and it's consistent to believe that the token persists at least as long as the interval. In order to define a predicate that captures this, we first need to be able to reason about whether it's consistent to believe in certain relationships involving the distance separating points in the time map.

Earlier we defined a predicate `elt` and a function `distance` that were useful in reasoning about the duration of intervals separating points in the time map. A query of the form `(elt (distance` pt_1 pt_2`)` *low high*`)` allows us to determine whether or not a set of bounds (*low* and *high*) on the distance separating a pair of points are supported by the current set of constraints (*i.e.*, the lower bound *low* is not greater than the GLB and the upper bound *high* is not less than the LUB). Now I want to consider an operator M such that a query of the form (M (elt (distance pt_1 pt_2) *low high*)) will succeed just in case the bounds on (distance pt_1 pt_2) are consistent with the current set of constraints.

Consider the following definition:

```
(<- (M (elt (distance ?pt1 ?pt2) ?low ?high))
    (and (strict-elt (distance ?pt1 ?pt2) ?glb ?lub)
         (=< ?low ?high)
         (=< ?low ?lub)
         (=< ?glb ?high)))
```

Suppose that the GLB and LUB on the distance separating pt1 and pt2 are 4 and 8 respectively (*i.e.*, (strict-elt (distance pt1 pt2) 4 8) succeeds). In this case the query (M (elt (distance pt1 pt2) 3 5)) should succeed, but (M (elt (distance pt1 pt2) 2 3)) should fail. This is obviously a nonmonotonic inference. The inference (M (elt (distance pt1 pt2) 3 5)) is valid as the example was given. However, if I subsequently add the contraint (elt (distance pt1 pt2) 7 *pos-inf*), then the inference is no longer valid. Noticing when the status of such nonmonotonic inferences change (and what the consequences are) is an important part of temporal reason maintenance. In what follows, I'll assume that it is clear how the M operator behaves with regard to shorthand expressions for uses of elt involving point-to-point disances (*e.g.*, pt=< and pt>).

Now we can proceed to define a predicate tt (for "true throughout" an interval) that will help us reason about facts that change over time.

```
(define-predicate (tt POINT POINT PROP))
```

A query of the form (tt pt1 pt2 Q) is taken to mean, "Is it possible that Q is true throughout the interval from pt1 to pt2?". The definition is:

```
(<- (tt ?pt1 ?pt2 ?q)
    (and (time-token ?q ?tok)
         (pt=< (begin ?tok) ?pt1)
         (M (pt=< ?pt2 (end ?tok)))))
```

We can also define a predicate t which is intended to stand for "true at" a point:

```
(define-predicate (t POINT PROP))
```

Its definition is trivial:

```
(<- (t ?pt ?q)  (tt ?pt ?pt ?q))
```

Describing exactly how the predicate tt works in various situations will occupy most of this and the next section. In the remainder of this section I want to look at two very basic aspects of its behavior. First, how are queries involving tt handled by the TMM? The exposition of temporal query processing should serve to familiarize the reader with some of the routine operations on the time map. The second issue I'm interested in involves what happens when default assumptions involving persistences are undermined. How does the TMM respond to update the data base, and how is the user alerted to the fact that assertions made on the basis of some earlier state of the time map are no longer warranted? Section 3.5.2 provides the reader with an introduction to the role temporal reason maintenance plays in the TMM.

3.5.1 Processing temporal queries

Understanding how the TMM handles temporal queries is best done in the context of an example. The example I'll be using is drawn from the machine shop domain [Firby 85] and deals with scheduling machines to perform various manufacturing tasks. In this simple example there are four machines lathe14, lathe17, lathe34, and lathe9, each of which is available for use only over certain intervals (a machine is available for use at a particular time just in case its production-status is free). The machines can also have various attachments installed over certain intervals. There is one event token corresponding to the task to complete (manufacture) a certain order (lot427). We're interested in determining which machines are available for assisting in the manufacturing process. First we have to set up the initial situation in the time map.

In this example, the reference point *ref* corresponds to 12:00 noon, the base temporal unit is a minute, and the function minc converts hours:minutes to minutes. The manufacturing task can't begin until after 10:30 AM (assume that some prerequisite operation won't be completed until that time) and must be finished by 1:00 PM. The task is estimated to take between an hour and an hour and fifteen minutes. The assertions below set up the situation shown in Figure 3.8:

```
(time-token (production-status lathe14 free) pstatus1)
```

```
Frame of reference: *ref*  Scale: 0.2

|          |          |          |          |          |          |  **
pstatus1  (production-status lathe14 free)
||------------------------------------------------------------------->
pstatus2  (production-status lathe17 free)
                                ||----------------------------------->
pstatus3  (production-status lathe34 free)
          ||---------------------|
pstatus4  (production-status lathe9 free)
                ||--------------------------------|
installed1 (installed milling-attachment lathe14)
    ||--------------------------------------------------------------->
order721 (manufacture lot427)
                            |---------------|----------|--|
```

Note: For this example I've provided a rule (**) along the top of the time map marked off in hour increments.

Figure 3.8: Time map for demonstrating temporal query processing

```
(time-token (production-status lathe17 free) pstatus2)
(time-token (production-status lathe34 free) pstatus3)
(time-token (production-status lathe9 free) pstatus4)
(time-token (installed milling-attachment lathe14) installed1)
(time-token (manufacture lot427) order721)
(elt (distance (begin pstatus1) *ref*) (minc 4:00) (minc 4:00))
(elt (distance (begin pstatus1) (end pstatus1)) 0 *pos-inf*)
(elt (distance (begin pstatus2) *ref*) (minc 1:00) (minc 1:00))
(elt (distance (begin pstatus2) (end pstatus2)) 0 *pos-inf*)
(elt (distance (begin pstatus3) *ref*) (minc 3:00) (minc 3:00))
(elt (distance (begin pstatus3) (end pstatus3)) 0 (minc 2:00))
(elt (distance (begin pstatus4) *ref*) (minc 2:30) (minc 2:30))
(elt (distance (begin pstatus4) (end pstatus4)) 0 (minc 3:00))
(elt (distance (begin installed1) *ref*) (minc 3:30) (minc 3:30))
(elt (distance (begin installed1) (end installed1)) 0 *pos-inf*)
(elt (distance (begin order721) *ref*) (minc 1:30) *pos-inf*)
(elt (distance (begin order721) (end order721)) (minc 1:00) (minc 1:15))
(elt (distance *ref* (end order721)) *neg-inf* (minc 1:00))
```

Now, consider the following query:

```
(and (time-token (manufacture lot427) ?tok)
     (tt (begin ?tok) (end ?tok) (production-status ?machine free))
     (instance-of ?machine lathe))
```

If we assume that each of lathe14, lathe17, lathe34, and lathe9 are provably instances of lathes, then this query should return with exactly two answers, one with substitution {(machine lathe14)(tok order721)}, and a second with substitution {(machine lathe9)(tok order721)}. The query will fail for {(machine lathe17)(tok order721)} (the corresponding token pstatus2 doesn't begin earlier enough) and {(machine lathe34)(tok order721)} (pstatus3 doesn't persist long enough).

For the slightly more complicated query:

```
(and (time-token (manufacture lot427) ?tok)
     (tt (begin ?tok) (end ?tok) (production-status ?machine free))
     (instance-of ?machine lathe)
     (tt (begin ?tok) (end ?tok) (installed milling-attachment ?machine)))
```

there is only one answer (with substitution {(machine lathe14)(tok order721)}), since there is only one lathe that is both free and has the necessary attachment throughout the required interval.

You might have noticed that the first query would have succeeded with lathe17 if we had made the additional assertion:

```
(elt (distance (begin pstatus2) (begin order721)) 0 *pos-inf*)
```

In Section 3.6 we will consider how to go about noticing that certain additional constraints (referred to as abductive premises) will allow a deduction to succeed where otherwise it would not. This sort of inference is critical in reasoning with incomplete information.

The above examples only begin to show off the system's capabilities. The TMM can handle a wide range of temporal queries. For an example illustrating a bit more versatility, let's suppose that there is a conveyor system conveyor2 located in the area designated as working-bay31 that has been threatening to break down for some time. Cognizant of this fact, the planner tries to inspect the conveyor following periods of heavy use. The manufacturing task (manufacture lot721) will make frequent use of conveyor2. In order to handle the inspection of conveyor2 efficiently, the planner might look for a time following the production of lot427 when someone is servicing a machine located in working-bay31 such that the conveyor is idle (and hence accessible for inspection) for at least fifteen minutes following the service task (the amount of time required for a detailed inspection).

Consider the following query:

```
(and (time-token (routine-service ?machine) ?tok1)
     (location ?machine working-bay31)
     (time-token (manufacture lot427) ?tok2)
     (M (pt< (end ?tok2) (begin ?tok1)))
     (M (elt (distance (end ?tok1) ?some-later-point) 15 *pos-inf*))
     (tt (end ?tok1) ?some-later-point (production-status conveyor2 idle)))
```

This will require a little explanation. It is assumed that predications involving the locations of massive objects like machine tools are timelessly true. If the TMM encounters a conjunct involving (elt (distance ?pt1 ?pt2) ?low ?high) or (tt ?pt1 ?pt2 P) in which either one or both of ?pt1 and ?pt2 are unbound, the system creates new points, binds the variables appropriately, and then continues as though the points were bound all along.

The above query can be interpreted as saying, find a token corresponding to a task to perform routine service on some machine located in working-bay31, such that it is possible that the task occurs after the manufacturing task involving lot427, and it is true immediately following and for at least 15 minutes after the service task that conveyor2 is not being used. In later sections, we'll see other examples demonstrating more complicated query processing capabilities.

So far I haven't mentioned anything about the ddnode/support-type pairs returned in an answer involving a temporal query. I'll get to this in the next subsection.

3.5.2 Noticing and responding to assumption failures

The answers to the queries in the examples of the previous section are based on default assumptions concerning the persistence of fact tokens in the time map. It is often the case that steps in deductions involving the predicate `tt` are invalidated by subsequent additions and erasures from the time map. There are three ways in which this can happen corresponding to the three conjuncts in the definition of `tt` (page 82):

1. `(time-token ?q ?tok)`
2. `(pt=< (begin ?tok) ?pt1)`
3. `(M (pt=< ?pt2 (end ?tok)))`

With regard to the first conjunct, the ddnode associated with the time token (bound to `?tok`) used in making the deduction can become OUT. Relative to the second conjunct it is possible that certain constraints will become OUT so that it is no longer possible to conclude `(pt=< (begin ?tok) ?pt1)`. The third conjunct involves a nonmonotonic inference. By adding constraints it may become possible to conclude that `(pt> ?pt2 (end ?tok))`, thereby invalidating the conjunct `(M (pt=< ?pt2 (end ?tok)))`. Generally the only time you are interested in the continued validity of conclusions extracted from the time map is when you have used those conclusions as a basis for deriving further consequences. Whenever the TMM successfully processes a query involving `tt`, it augments the current answer (in particular the ddnode/support-type pairs) in such a way that assertions occurring in the context of that answer will be believed, just in case the above three conditions continue to hold. The details of how this is accomplished will be presented in Chapter 4; in this chapter we're only concerned with how we can put this to use.

The addition of new fact tokens and constraints on their occurrence can result in changes in the duration of existing time tokens. This change may serve to violate default assumptions concerning the persistence of fact tokens. I'll refer to such violations as *interactions*. An interaction occurs when a belief that was previously warranted is suddenly threatened by the addition of new information.

Suppose that I had planned on the use of the faculty conference room for a colloquium I am responsible for organizing. At some point in the morning I am told that a minor funding potentate will be visiting in the afternoon and the conference room will be used for an impromptu reception during just those hours I had scheduled the colloquium. I should realize that my colloquium is threatened and arrange for an alternative meeting place.

Not all interactions are unpleasant or a cause for alarm. As another example, say that I've noticed the kitchen sink is draining slowly, and I resolve to call the landlord. Before I get around to making the call, I notice that the sink is now draining with no problem. The task to call the landlord should now evaporate without much fuss.

The consequences of making certain assumptions can be arbitrarily complex. Suppose that instead of bothering the landlord (who wouldn't have done anything anyway), I consider doing the job myself and begin to formulate an elaborate plan that involves buying tools from the local hardware and rearranging all my appointments for the day. If I then notice that the problem has disappeared, the partially expanded plan should evaporate as well, but in this case dealing with the consequences may require considerable replanning.

Now let's be a bit more specific about what we mean by an interaction. In the TMM an assertion is said to be contingently believed if that assertion is believed just in case some other fact is believed to be true at an instant or throughout some interval. The stipulation that a fact is believed to be true at an instant or throughout some interval is said to be a (temporally dependent) condition for belief. An interaction occurs when something is determined to happen that undermines the conditions for belief in some assertion. For example, suppose that I was counting on the department office being open late enough for me to use the copier after completing the most recent draft of my thesis. An interaction could arise in a number of alternative ways:

1. I am told that the secretary will be locking up early today.
2. the draft takes longer than I anticipated and I'm still working when 5:00 rolls around
3. I discover that the office was never opened in the first place; it's Grover Cleveland's birthday.
4. I hear from an office mate that the copier broke a belt and there is no replacement part

Figure 3.9 shows a time map describing the situation. The reference point, *ref*, is 12:00 noon, the base temporal unit is a minute, minc converts hours:minutes to minutes, the CS office opens at 8:00 AM and usually stays open until 5:00 PM, and the copier has been up (functional) for at least the last 3 hours. The requisite assertions are:

```
(time-token (operational-status CScopier in-service) in-service31)
(time-token (open CSoffice) open32)
(time-token (finish draft49) finish42)
(time-token (copy draft49 3) copy-task12)
(elt (distance (begin open32) (end open32)) 0 (minc 9:00))
(elt (distance (begin in-service31) (end in-service31)) 0 *pos-inf*)
(elt (distance (begin open32) *ref*) (minc 4:00) (minc 4:00))
(elt (distance (begin in-service31) *ref*) (minc 3:00) (minc 3:00))
(elt (distance *ref* (begin finish42)) (minc 1:00) *pos-inf*)
(elt (distance (begin finish42) (end finish42)) 0 0)
(elt (distance (end finish42) (begin copy-task12)) 0 *pos-inf*)
(elt (distance (begin copy-task12) (end copy-task12)) 20 30)
```

What I want to demonstrate with this example is how the rule of persistence (Section 3.3) and the query mechanism work in concert to detect and annotate interactions. I'll have to be more precise about how *the "true throughout" predicate* tt *works*, but before I get to that it will be instructive to see what functionality is lacking.

The rule of persistence has already been briefly described in this chapter (page 75). Its mandate is carried out automatically by the time-map machinery. To reiterate: the rule of persistence detects when two tokens, T1 and T2, asserting contradictory propositions, are ordered such that (begin T1) precedes (begin T2) and it's not the case that (end T1) precedes (begin T2). Whenever the TMM finds a pair of tokens satisfying this criteria it determines if it is possible that (end t1) precedes (begin T2) (*i.e.*, the least upper bound on the distance separating the two points is greater than 0). If this last condition cannot be met then the data base is said to be inconsistent and some action must be taken to remove the inconsistency. If the condition is met, then the TMM adds the constraint (elt (distance (end T1) (begin T2)) *pos-tiny* *pos-inf*)[9]. The operation of adding constraints via the rule of persistence is referred to as *persistence clipping*.

[9] Recall that *pos-tiny* denotes a number greater than zero but less than any positive number (except itself). This constraint in conjunction with the way we handle tt implies that persistences are closed on the left and open on the right. That is to say the fact associated with a persistence is believed at the beginnning of the persistence interval and up to but not including the end of that interval.

```
Frame of reference: *ref*  Scale: 0.2

open32 (open CSoffice)
||---------------------------------------------------------------------->
in-service31 (operational-status CScopier in-service)
           ||---------------------------------------------------------->
finish42 (finish draft49)
                                                           |>|
copy-task12 (copy draft49 3)
                                                           |>----|~|
```

Figure 3.9: Initial time map for the copier example

To illustrate, suppose we added the following to the time map in Figure 3.9 (this is case 4 of the interactions listed on page 88):

```
(time-token (broke belt34 CScopier) malfunction34)
(time-token (operational-status CScopier down) down128)
(elt (distance (begin malfunction34) (end malfunction34)) 0 1)
(elt (distance *ref* (begin malfunction34)) (minc 0:30) (minc 1:00))
(elt (distance (begin down128) (end down128)) 0 *pos-inf*)
(elt (distance (end malfunction34) (begin down128)) 0 0)
```

That is to say a belt breaks in the department copier sometime between 12:30 and 1:00 PM resulting in the copier being down for some indeterminate period of time. In order for the TMM to understand that being "down" contradicts being "in-service", the data base must contain a rule such as the following:

```
(<- (contradicts (operational-status ?machine ?status1)
                 (operational-status ?machine ?status2))
    (thnot (:= ?status1 ?status2)))
```

Figure 3.10 shows the result of having made the above assertions. Notice that the only thing that has changed, other than the addition of down128 and malfunction34, is the upper bound on in-service31; the persistence in-service31 has been clipped by the persistence down128 by the TMM in resolving the apparent contradiction between the two tokens.

Now for a moment let's return to my initial problem and the time map of Figure 3.9. I'm counting on using the department copier for making three copies of my thesis draft. If the machine is operational and the office is open, then I'm willing to believe that I can

```
Frame of reference: *ref*  Scale: 0.2

open32 (open CSoffice)
||----------------------------------------------------------------------->
in-service31 (operational-status CScopier in-service)
          ||-------------------------------------------------|
malfunction34 (broke belt34 CScopier)
                                                      |-----||
down128 (operational-status CScopier down)
                                                      |-----||----------->
finish42 (finish draft49)
                                                            |>|
copy-task12 (copy draft49 3)
                                                            |>----|-|
```

Figure 3.10: Demonstration of persistence clipping

execute a simple set of steps, the result of which is that I'll be in possession of three copies of my thesis draft. In the TMM one way of setting this up would be to execute the following *LISP fragment with the time map of Figure 3.9:*

```
(for-first-answer
   (fetch '(and (time-token (copy draft49 3) ?tok)
                (tt (begin ?tok) (end ?tok)
                    (operational-status CScopier in-service))
                (tt (begin ?tok) (end ?tok) (open CSoffice))))
   (add '(and (time-token (possess copies14) possess31)
              (elt (distance (end ?tok) (begin possess31)) 0 0))))
```

You can think of the query as establishing conditions warranting the plan of using the department copier for making my three copies. The assertion is a prediction of what will happen if those conditions are maintained and I actually execute a plan to make three copies on the department copier. The time map in Figure 3.11 shows the result of this operation. Note that apart from the addition of the new token **possess31** nothing has changed from the time map in Figure 3.9.

Notice that the query establishing the conditions for the plan to use the office copier would not have succeeded in the time map of Figure 3.10, since in this time map the department copier broke down between 12:30 and 1:00. However, if, having already executed the above code fragment, we add the assertions involving the malfunction of the department copier, then we won't notice that the conditions for our plan are no longer met. So far I

```
Frame of reference: *ref*  Scale: 0.2

open32 (open CSoffice)
||------------------------------------------------------------------->
in-service31 (operational-status CScopier in-service)
           ||-------------------------------------------------------->
finish42 (finish draft49)
                                                          |>|
copy-task12 (copy draft49 3)
                                                          |>----|¯|
possess31 (possess copies14)
                                                              |>|----->
```

Figure 3.11: Result of the hypothesis generation and projection steps

have said very little about what is done after a query succeeds with respect to the end points of persistences involved in satisfying a "true throughout" query. If an application program makes assertions on the basis of a answer returned by the TMM, then those assertions are dependent upon the state of the time map reflected in the query answer. We would like to ensure that the consequent assertions are believed just in case the antecedent conditions in the time map continue to hold. There are two ways that this might be accomplished. First, we could just add constraints that ensure the antecedent conditions. Second, we could keep track of the validity of the antecedent conditions and update the status of the consequent assertions accordingly. In the following pages I want to explore the advantages and disadvantages of both methods.

We would like to be able to keep track of the continued validity of conditional assertions. If believing in X relies on believing in P being true throughout an interval, then we want to be able to either guarantee the condition or be able to detect when it is no longer met. P is said to be *protected* throughout an interval pt1 to pt2 just in case there exists a persistence T1 with fact type P such that T1 begins before pt1 and T1 cannot be shown to end before pt2. A protection is *violated* if its corresponding fact is ever false during the interval.

One way we might guarantee a protection is by adding the constraint (pt=< ?pt2 (end ?tok)) to ensure the continued validity of the nonmonotonic assumption (M (pt=< ?pt2 (end ?tok))) in the definition of tt. Suppose that we get the TMM to do this for us whenever we make an assertion in the context of an answer containing a nonmonotonic temporal assumption. Now if we execute the code fragment for establishing the conditions

```
Frame of reference: *ref*  Scale: 0.2

open32 (open CSoffice)
|------------------------------------------------------------|~~~~~~~~~~>
in-service31 (operational-status CScopier in-service)
            |------------------------------------------------|~~~~~~~~~~>
finish42 (finish draft49)
                                                           |>|
copy-task12 (copy draft49 3)
                                                           |>----|~|
possess31 (possess copies14)
                                                                |>|~~~~~>
```

Figure 3.12: Result of the hypothesis generation and projection steps

for the plan to use the office copier, we get the time map in Figure 3.12.

The important thing to notice here is the change in the lower bound of the in-service31 and open32 persistences. The conditions for the plan to use the office copier are now guaranteed. In fact the TMM would not allow you to add the assertions concerning the malfunction of the department copier. These assertions are inconsistent with the time map of Figure 3.12.

This technique of "stretching" persistences is one method of implementing protections. The only way that a protection violation can occur is by relaxing the constraints (i.e., removing one or more constraints). This means that the system is monotonic: adding new facts can never cause a protection violation. A protection in this scheme is equivalent to a pair of constraints: that a token with the required fact type begins before and ends after the interval over which the fact is being protected.

This persistence-stretching method has certain advantages. It is employed in the TMM as the method of choice for handling certain types of resource management [Miller 85a]. It can also be used to implement a variant of Vere's DEVISER planner [Vere 83] that uses the point-and-token-based representation of time presented in this dissertation. One of its main advantages is that the constraints added in constructing protections act to enforce deadlines in a neat parsimonious way. For instance, the plan to use the department copier in Figure 3.12 after the hypothesize/projection step is constrained to begin at least 20 minutes before 5:00 PM.

The disadvantages, however, make it worth our while to consider alternatives. One disadvantage is flexibility. It is often convenient in debugging plans to determine what the repercussions are of modifying certain constraints in the time map. For instance, in building a house one might ask what problems would be encountered if the concrete basement floor was poured (installed) in the early stages of construction. Using the persistence-stretching method you would simply be informed (assuming that there are potential problems in pouring the concrete early) that such a modification cannot be made (*i.e.*, it is inconsistent with the current data base). What you want, however, is a blow-by-blow account of what things might go wrong, so you can assess what might be done to make an early pouring feasible.

Another disadvantage concerns the need to accurately annotate assumption failures (of which protection violations are an example) in time maps. As an example, suppose that I am delayed in finishing the draft of my dissertation and I won't get a chance to make copies until after 5:00 PM. I represent this by adding the constraint `(elt (distance *ref* (begin copy-task12)) (minc 5:00) *pos-inf*)` to the time map of Figure 3.12. This constraint is inconsistent with the time map as it is now, and the reason is due to the constraint added by the protection. Now the reason for the delay might be a squash game, or it might be something slightly more critical like a forgotten appointment with one's advisor. In any case you'd like to be aware of what is involved so you can react appropriately. Determining the consequences of adding or removing a constraint using the protection scheme outlined above is nontrivial.

Ideally what we would like is a data base that, given some additional information, would do its best to accommodate the new data, reorganize itself to suit, and then spit out a list of beliefs that had to be modified in the revised data base in order to include the new information. Unfortunately this sort of magic data base in not likely to materialize. It is possible, however, to use data-dependency methods in a highly controlled manner to get surprising flexibility out of a temporal data base.

By "highly controlled manner" I mean that we stipulate in advance what sort of behavior we expect of the system. The `M` operator serves as annotation to tell the deductive retrieval machinery what sort of default assumptions to incorporate into the current answer. The predication `(M P)` is said to indicate a default in that it effectively says believe `P` if you can, but be prepared to give it up (and anything that depends upon it as well) if you ever have reason to believe `(not P)`. By using defaults what we are really doing is stating a priority on beliefs. Rules containing the **M** operator specify the conditions under which they can be overridden. Such rules are said to be nonmonotonic because the addition of new information can result in the falsification of old. In general the behavior of nonmonotonic systems is difficult to predict or analyze [McDermott 80], but in cases where the defaults are few and of a simple form the behavior can be quite straightforward. In the TMM defaults are used to give precedence to the rule of persistence.

The dependencies set up in processing queries involving tt don't really serve to "protect" anything. They establish what might be more appropriately called "persistence assumptions". Nevertheless, because of the way in which these assumptions are used in planning systems (*e.g.*, [Miller 85a]), I continue to refer to them as protections. As we'll see they are used to accomplish very much the same effect.

Suppose that I am told that the secretary leaves at noon and locks the department office for the day. This would be added to the time map of Figure 3.11 as:

```
(time-token (lockup sk34 CSoffice) lockup17)
(time-token (closed CSoffice) closed33)
(elt (distance (begin lockup17) (end lockup17)) 1 2)
(elt (distance *ref* (begin lockup17)) 0 0)
(elt (distance (end lockup17) (begin closed33)) 0 0)
(elt (distance (begin closed33) (end closed33)) 0 *pos-inf*)
```

In response to these assertions, the system reconfigures the data base and notes that something that was formerly believed is no longer so; the time token with fact type (possess copies14) is now OUT. The result is shown in Figure 3.13. The token possess31 is labeled as OUT and is shown as unconstrained relative to the other tokens in the time map. Something similar would have happened if we had added the assertions about the department copier malfunctioning.

Simply labeling tokens as OUT won't be sufficient for handling complex hypothetical reasoning of the sort envisioned in Chapter 1. Some interactions are inconsequential; others are of minor interest, requiring at most a bit of bookkeeping; and still others are critical, demanding our immediate attention and, potentially, the expenditure of significant computational resources for their resolution. The TMM provides utilities to enable the user to tell the system exactly what to do when an assertion fails or, for that matter, becomes true. These are called *change-driven interrupts*, and they are implemented using data dependencies (see Section 3.2 for details). Such interrupts can also be prioritized so the more

```
Frame of reference: *ref*  Scale: 0.2

possess31 (possess copies14) OUT
<>|--------------------------------------------------------------------->
open32 (open CSoffice)
||----------------------------------------------|
in-service31 (operational-status CScopier in-service)
           ||---------------------------------------------------------->
lockup17 (lockup sk34 CSoffice)
                                                ||
closed33 (closed CSoffice)
                                                 ||-------------------->
finish42 (finish draft49)
                                                            |>|
copy-task12 (copy draft49 3)
                                                            |>----|-|
```

Figure 3.13: Result of the hypothesis generation and projection steps

critical changes are handled first. The use of change-driven interrupts helps to pinpoint what problems in the data base require attention. It also helps in debugging by allowing the user to make tentative changes to the data base and then consider their repercussions: "What if I put off that game of squash until the afternoon — could I get the draft done in time to make the copies by noon?" Interrupts can range from simple annotations to complex programs which add all sorts of information to the time map.

The following illustrates how to set up an if-erased demon (Section 3.2) designed to alert the calling program to tasks that appear to fail.

```
(if-erased '(time-token ?schema ?tok)
    (for-first-answer (fetch '(and (inst ?schema task-schema)
                                   (pt=< *now* (begin ?tok))))
          code-to-debug-failing-task))
```

What this says is, if an assertion denoting a time token ever becomes OUT, then check to see if the token corresponds to a task that has yet to be achieved, and if so, execute some code to try to see what went wrong.

3.5.3 Alternative methods for reasoning about protections

Two approaches to handling protections have been presented in this section. One involves the use of defaults to keep track of whether or not a fact continues to be protected throughout an interval, and the other requires imposing constraints to ensure the protection. In the first, we stipulate what conditions must hold for a proposition to be believed, and when those conditions are no longer met we retract our belief in that proposition. The extent of our belief is built into the dependencies constructed by the TMM at the time the belief is asserted. In the time map, persistence clipping is given a high priority. Conflicts involving a protection and a pair of overlapping contradictory tokens are always resolved in favor of eliminating the apparent contradiction. If this results in a protection violation, then the TMM notes which additional beliefs are affected. These beliefs are marked as OUT in the data base. The TMM provides a form of temporal reason maintenance via its treatment of protections and persistences. The second approach, which I'll refer to as the *relaxation approach*, handles protections by adding constraints to extend persistences. This means that when two contradictory tokens are constrained to overlap, the rule of persistence might not be able to carry out its mandate. To do so would make the set of constraints inconsistent. It is also possible that the rule of persistence has already ordered two tokens and some subsequent constraint is determined to be inconsistent with existing constraints. The system is then forced to find those constraints implicated in the inconsistency and then trace them to determine which ones can be relaxed in order to proceed. If the constraints chosen for relaxation are always those added in the course of setting up protections, then a system based on the relaxation method will behave similar to one based on the default approach. Given any functionality supported by one, I have little doubt that a suitably augmented version of the other could not support the same functionality. I will focus upon the default approach because I think that it is the cleaner of the two and more directly supports the sort of computation required of temporal reasoning.

Temporal databases of the sort being considered in this thesis are unavoidably nonmonotonic. It is inevitable that if we predict anything we will occasionally be wrong and be forced to revise our beliefs. The question is how and in what manner should the revision occur.

David McAllester [McAllester 80] has claimed that nonmonotonic logics and the data dependency systems that most closely characterize them introduce nonmonotonicity too early, and at the cost of much confusion. He espouses instead the idea of monitoring a monotonic deductive system and handling the potential inconsistencies that arise using external rules for establishing the precedence of beliefs. The system's behavior as a whole is nonmonotonic, but the core deductive component is not. The relaxation method of handling protections and persistences can be seen as such an approach. In McAllester's system, dependencies between propositions are recorded to assist in resolving contradictions. I believe that it would be fairly straightforward to implement a system such as the TMM using the relaxation approach described above and McAllester's reason maintenance system. However, I claim that in the case of the TMM the defaults are well enough understood that some of the deductive machinery can be effectively short circuited to expedite the reasoning process.

In the end, it is the functionality of the TMM that I am most interested in communicating. What sort of computation is required in order to support reasoning about time? Issues involving computational complexity are critical in separating out a set of functions that are both useful and feasible. However, the decision to use the default approach over the relaxation approach will not result in any significant reduction in complexity. What we want is a computational framework that is pared down to the essentials required for supporting temporal reasoning.

3.6 Hypothesis generation and abductive inference

Section 1.3 discussed an approach to reasoning about time referred to as *shallow temporal reasoning*. The basic idea is that, beginning from an initial data base of facts and rules, one attempts to extend that data base by a cycle of shallow inferences (*i.e.*, inferences involving few steps). The extensions generally correspond to explanations in the case of diagnosis or more detailed descriptions of plans for achieving tasks in the case of planning. In Chapter 5, I will demonstrate how shallow temporal reasoning plays a role in planning. This section describes how the TMM assists in one aspect of shallow temporal reasoning: the generation of competing hypotheses. In the following, I will use the general term "planner" to refer to an application program using the TMM.

The first step in any reasoning process involves choosing something to work on (*i.e.*, reason about). This choice is obviously crucial, but I will have very little to say about it here. In the TMM, this something to work on is generally an event token corresponding to a task or interesting phenomenon.

Once a planner has decided what to work on, it tries to generate a set of competing hypotheses that in some way place the event in a temporal and causal context. In planning the objective is to reduce a complex task to a series of simpler subtasks. So the planner would be trying to find a set of plans (or task reductions) whose applicability criteria are met. For example, in the machine-shop domain, you might have a rule stating that, whenever the big engine lathe is tied up and you get a rush job, the best thing to do is to find a production turret lathe and interrupt whatever job is being performed on that lathe so you can get the rush job out in time. In diagnosis one might want to account for some unexplained fact (a symptom or malfunctioning part). As an example of a diagnostic task, the planner might be trying to explain why `assembly-unit34` has broken down. The planner might have a rule that states, if an assembly unit breaks down, and it is found to be low on hydraulic fluid, and it is not overdue for routine service, then suspect O-ring deterioration and schedule an overhaul of the hydraulic system. It's quite likely that there will be more than one hypothesis for a given event. The planner might also have a rule that states, if an assembly unit breaks down, and it's found to be low on hydraulic fluid, and it's using a new brand of hydraulic fluid, then suspect that the new fluid is more volatile and has to replaced more frequently or cooled more thoroughly. Each hypothesis from the TMM's point of view consists of a set of facts that must be true over certain intervals and a set of additional constraints to be added to the time map (a restriction on the current partial order). In the case of the hypothesis concerning the new brand of hydraulic fluid, one additional constraint might be that the planner believes that `assembly-unit34` had its most recent routine service after the last of the Garbosol Lo-V hydraulic fluid ran out. The TMM can keep track of several of these hypotheses (and their corresponding restrictions) at once in order to separate generating hypotheses and choosing between them.

For hypothesis generation we need a method of extracting possibilities from the time map in such a way that assumptions made in the course of the extraction do not permanently change its content. In other words, we want to construct a hypothetical situation that satisfies some criteria (*e.g.*, of a good explanation or a suitable plan) and then set the description of that situation aside for later comparison with other situations meeting the same criteria. This could be accomplished by using some sort of context mechanism [McDermott 83] to split the database, but given that the deductions involved are assumed to be shallow and only one hypothesis will surface as a winner, this is hardly necessary.

In addition to the need for keeping competing hypotheses separate, it is also often convenient to allow the system to jump to certain conclusions in the course of generating hypotheses. The advantage of this is that exploration can proceed despite certain gaps in our knowledge of the situation. The TMM employs a general operator A (for abductive) that licences the deductive retrieval system to jump to a conclusion given that it is consistent to believe that conclusion. This license is granted despite the fact that there is (currently) no deductive warrant for believing that the conclusion is valid. Note that the A operator is distinctly different from the nonmonotonic operator M. In a query of the form (A P), P will actually be added to the data base given that (M P) succeeds. In such a situation P is said to be an *abductive premise.*

Abductive premises generated during backward chaining have restricted scope. An abductive premise can participate in a deduction only within the scope of the answer (*i.e.*, object of data type ANS) in which it was generated. As long as ans* is bound to an ANS containing an abductive premise, it is (effectively) an assertion in the data base. If for any reason the current answer is reset, either by LISP code or by backtracking through (A P), the premise is (effectively) removed from the data base (but not from the answer; so, if one retains a pointer to the answer, its associated abductive premises can be restored by binding it to ans*). An ANS containing one or more abductive premises is called an *abductive answer.*

The TMM can generate any number of abductive answers. Taken together, the abductive premises contained in these answers may be inconsistent, but each answer in isolation is assumed to correspond to a consistent data base. Given a set of abductive answers corresponding to alternative hypotheses, the user can jump back and forth between answers in an effort to choose the best such hypothesis. Examination of the underlying abductive premises will likely play a significant role in this selection process. You can extract the abductive premises from an ANS using the function extract-abductive-premises. It is also possible to have the TMM consult the application program at the point in a deduction that an expression involving A is encountered. The TMM then asks for permission to incorporate an abductive premise into the current answer. This has the advantage of enabling the application program to cut off obviously fruitless searches and thereby direct searches with some degree of precision. These techniques aren't really of central importance here; they ultimately depend primarily on domain specific issues that are inappropriate to pursue in this dissertation.

Once the application program has selected a given hypothesis, it is necessary to make the underlying abductive premises a permanent part of the data base. This raises the issue of providing some sort of justification for abductive premises that are permanently asserted. Consider a simple scenario involving the following rule:

```
(<- (engine-status ?vehicle flooded)
    (and (engine-compression ?vehicle adequate)
         (A (status (carburetor-float-valve ?vehicle) stuck-open))
         (fuel-available-at-intake-manifold ?vehicle excessive)))
```

The above rule states that, you can determine that a vehicle is flooded, if you can prove that the engine compression is adequate (something you should probably already know), and, by consistently assuming that the carburetor float valve is stuck, you can prove that too much fuel is being delivered to the intake manifold. If the above rule succeeds in the query **(engine-status pontiac379 flooded)** and the corresponding answer is selected as the best explanation for how the engine became flooded, then the abductive premise **(status (carburetor-float-valve pontiac379) stuck-open)** should be added to the data base on a permanent basis. What should its justification be? It shouldn't come from **ans***, since it doesn't really depend upon the immediately prior deductions. I may be willing to jump to the conclusion that the float valve is stuck, simply because I've been told that this is a common ailment of older pontiacs. In general, characterizing abductive support is quite difficult. Abductive premises often correspond to good hunches about how to proceed in lacking more detailed information. The TMM relies on the application program to supply suitable support for an abductive premise. If an assertion (other than one made by the abductive machinery) occurs in the scope of an answer containing an abductive premise, then that premise is asserted to the data base. The justification for an abductive premise is extracted from a set of ddnode/support-type pairs specified using the LISP macro **abductive-support**. **Abductive-support** works just like the **answer-support** macro described in Section 3.2.4. If **selected-hypothesis** is bound to an **ANS** containing the abductive premise **(pt=< (end routine-service-on-lathe41) (begin order437))** and the warrant for jumping to this conclusion is **(on-schedule maintenance-crew)**, then this dependency would be installed as a result of:

```
(abductive-support (+ '(on-schedule maintenance-crew))
   (bind (ANS (ans* selected-hypothesis))
         (add  consequences-of-believing-selected-hypothesis)))
```

The default abductive support corresponds to the ddnode/support-type pair **(abductive-premise +)** where **abductive-premise** corresponds to a ddnode that is always IN. This support just serves as annotation to distinguish assertions generated by **A** from "real" premises. There are other mechanisms for setting up justifications for abductive premises, but abductive-support will serve all of our needs in this dissertation.

Steps one through three of our temporal reasoning paradigm can be restated as: extract a number of answers (hypothetical situations consistent with the current state of the database), choose one of those answers and make it current, and then assert whatever new predictions you can justify using that answer. Translated into pseudo-code this might look like:

```
(let ((best-answer-so-far ()))
     (for-each-answer (fetch criteria-for-explanation-of-event)
       (if (new-answer-is-better-than-best-answer-so-far
                                          ans* best-answer-so-far)
           (:= best-answer-so-far ans*)))
     (abductive-support (+ reasons-for-believing-abductive-premises)
         (bind ((ans* best-answer-so-far))
               (add consequences-of-the-best-explanation))))
```

In the TMM, the only conclusions we are interested in jumping to involve the order in which events occur. All of the abductive premises I will be discussing involve predications of the form (elt (distance pt_1 pt_2) *low high*) (or shorthand versions (*e.g.*, (pt< pt_1 pt_2))). An abductive premise in this case constitutes a constraint or restriction on the partially ordered time map. The question we have to ask ourselves is, "When is it reasonable to add a constraint to the time map?" A time map is a partial order on points and hence on tokens representing the occurrence of events and persistence of their effects. If it were totally ordered, there would never be any need to jump to conclusions. That is to say, if we had perfect knowledge of the past, present, and future, we wouldn't need to assume anything. But then there would be little incentive for doing much of anything, and certainly no point in planning, except as idle speculation about how things might possibly be, were there a chance they could be otherwise. Fortunately, we can influence the future. There are events that we have control over. We can decide when, and in what order to perform a set of actions. By our actions, we can both cause and prevent other events. This means that we can decide to order certain events, in the sense that we can choose to believe that they will occur in a given order. But indecision about the order in which to execute actions is not the only source of uncertainty. Some orderings are not consistent with the rest of my beliefs. There are events I have no power to influence. And there are events about whose precise occurrence or, for that matter, possibility of occurrence I am ignorant. If I commit to the wrong ordering, my predictions about the future will be suspect, and I will have to resolve the contradictions that arise when my observations conflict with my beliefs.

```
Frame of reference: *ref* = 12:00 AM  Scale: 5.2

proxlocation45 (proximity John Cambridge)
<<------------------------------------------------|
bectonhours3 (open Becton)
 |-----------------------------------------------------|
sterlinghours7 (open Sterling)
 |--------------------------------------------------------------------|
beineckehours14 (open Beinecke)
      |--------------------------|-----------------------------------------|
scholarlypursuit46 (access-scholarly-reference John MedievalArt421)
                           |>---------------|----|
arrival14 (arrive John NewHaven)
                                            |----||

proxlocation31 (proximity John NewHaven)
                                            |----||------------------------>
```

Figure 3.14: Time map for the Yale libraries example

If on the other hand I fail to commit to an ordering, then my predictions are likely to be inadequate. This tradeoff between commitment and procrastination is a recurring theme in temporal reasoning. Often the only justification for certain abductive premises is, "wishful thinking sometimes works". I have very little to say about justifying abductive premises. The main issue addressed here concerns how one goes about extracting possibilities from a partially ordered time map.

One of the primary methods for reasoning about possibilities involving restrictions on the time map involves the use of an alternative, abductive definition of the "true throughout" predicate (see page 82 for the previous definition).

A query of the form `(tt pt1 pt2 Q)` is now taken to mean, "Is there a restriction on the current time map such that it is possible that `Q` is true throughout the interval from `pt1` to `pt2`?". The new definition is:

```
(<- (tt ?pt1 ?pt2 ?q)
    (and (time-token ?q ?tok)
         (A (pt=< (begin ?tok) ?pt1))
         (M (pt=< ?pt2 (end ?tok)))))
```

To see how this works in processing temporal queries, consider the following example.

John, a graduate student attending Harvard, is interested in making some notes from a rather obscure treatise on medieval art. Having made inquiries, he knows that there are only three libraries in the world that have copies of this work, and all three are located at Yale University in New Haven, Connecticut. The three libraries are called Beinecke, Sterling, and Becton (Beinecke houses the original work). John has arranged to take the train from Cambridge to New Haven in order to get the information that he needs for his research. It is currently 1:00 PM, and John won't be arriving in town until sometime between 4:00 and 5:00. He has estimated that his work will take three to four hours, and he's worried that the libraries won't be open long enough for him to finish the job and get back to Cambridge for classes tomorrow. Becton closes at 6:00 PM, and Sterling at 9:00. Beinecke is open eight hours a day, but John is not sure what those hours are. He does know, however, that they begin after 9:00 AM and end before 10:00 PM. The situation is shown in the time map display in Figure 3.14.

Consider the following query:

```
(and (time-token (access-scholarly-reference John MedievalArt421) ?tok)
     (tt (begin ?tok) (end ?tok) (proximity John NewHaven))
     (tt (begin ?tok) (end ?tok) (open ?lib)))
```

The TMM will return with two answers for this query. The first answer has substitution {(lib Sterling) (tok scholarlypursuit46)} and abductive premises {(pt=< (begin proxlocation31) (begin scholarlypursuit46))}. The second answer has substitution {(lib Beinecke) (tok scholarlypursuit46)} and abductive premises {(pt=< (begin proxlocation31) (begin scholarlypursuit46)) (pt=< (begin beineckehours14) (begin scholarlypursuit46))}.

Notice that there is no answer returned such that ?lib gets bound to Becton. This is because the conjunct (tt (begin ?tok) (end ?tok) (proximity John NewHaven)) succeeds only if the beginning of scholarlypursuits46 is constrained to follow the beginning of proxlocation31, and given this constraint bectonhours3 doesn't persist long enough.

I'd like to embellish this example somewhat in order to provide the reader with some more exposure to code employing TMM routines. At the same time, I'll introduce some notation and techniques for reasoning about plans. The basic ideas presented here might be used for text comprehension or robot problem solving. In Chapter 5, I'll extend the methods shown here to assist in robot problem solving tasks. The first thing we need is a predicate to-do:

```
(let ((1st ANS) (possible-plans ()))
     (for-each-answer
        (fetch '(to-do ?tsk ?type ?plan-description))
        (:= possible-plans (cons ans* possible-plans)))
     (bind (ANS (ans* (choose-best-plan-answer possible-plans)))
           (add '(plan-chosen-for ?tsk ?plan-description))
           code-to-perform-plan-expansion))
```

Figure 3.15: Code fragment for reasoning about plans

```
(define-predicate (to-do TOKEN PROP PROP))
```

In a predication of the form (to-do ?tok ?type ?plan-description), ?tok is a token corresponding to a task of type ?type such that ?plan-description provides a detailed account of how to achieve such a task given the temporal constraints on ?tok. Notation for plan descriptions will have to wait until Chapter 5, but the following backward chaining rule still serves to illustrate how the to-do predicate might be used in deriving plans for specific situations.

```
(<- (to-do ?tsk (access-scholarly-reference ?agent ?ref)
           schema-describing-detailed-plan)
    (tt (begin ?tsk) (end ?tsk) (and (has-copy ?ref ?lib)
                                      (instance-of ?lib library)
                                      (location ?lib ?loc)
                                      (proximity ?agent ?loc)
                                      (open ?lib))))
```

The above rule states that *schema-describing-detailed-plan* describes a plan that ?agent might use to gain access to ?ref just in case it is true throughout the time allocated for ?tsk that some open library ?lib in the proximity of ?agent possesses a copy of ?ref[10]. We'll assume that facts like (has-copy MedievalArt421 Beinecke), (instance-of Sterling library), and (location Becton NewHaven) are timelessly true.

Now consider the code fragment shown in Figure 3.15. The for-each-answer code collects a list of answers corresponding to possible ways of achieving a task of type ?type given the temporal constraints on the token ?tsk. I am assuming that there is a set of rules involving the to-do predicate that serve as an index into a library of plans for achieving various tasks (see Section 5.2). The function choose-best-plan-answer takes a list of

[10] Note that (tt ?b ?e (and P_1 ... P_n)) is equivalent to (and (tt ?b ?e P_1) ... (tt ?b ?e P_n)).

```
Frame of reference: *ref* = 12:00 AM  Scale: 5.2

proxlocation45 (proximity John Cambridge)
<<------------------------------------------------|
bectonhours3 (open Becton)
 |------------------------------------------------------|
sterlinghours7 (open Sterling)
 |---------------------------------------------------------------------|
beineckehours14 (open Beinecke)
      |---------------------------|-----------------------------------------|
arrival14 (arrive John NewHaven)
                                             |----||
proxlocation31 (proximity John NewHaven)
                                             |----||------------------------>
scholarlypursuit46 (access-scholarly-reference John MedievalArt421)
                                              |>---------------|----|
scholarlysubtask1 (locate John MedievalArt421 (stacks Sterling))
                                               |>----|--|
scholarlysubtask2 (extract-information  John  MedievalArt421)
                                                     |>--|--|
```

Figure 3.16: John chooses to do his research in Sterling

answers (objects of data type ANS) describing plans and selects the best such answer on the basis of some criteria. In the last part of the code fragment in Figure 3.15 the current answer **ans*** is bound to the best answer as specified by **choose-best-plan-answer**, and some number of assertions are made in the context of that answer in the course of expanding the chosen plan according to the value for the variable **?plan-description** in the selected answer. The assertions in the context of an answer containing abductive premises result in these premises being added to the data base. Since no abductive support was supplied, these abductive premises are given the above mentioned default abductive support.

If the code in Figure 3.15 is executed in the context of answer containing the following bindings {**(tsk scholarlypursuit46) (type (access-scholarly-reference John MedievalArt421))**}, then the variable **possible-plans** will be bound to a list of two answers differentiable from one another only in that one commits John to using Sterling and the other to his using Bienecke. Also, the one involving Bienecke requires two abductive premises. If we assume that the function **choose-best-plan-answer** uses the criteria of fewest abductive premises (obviously a poor choice in general), then the remainder of the code in Figure 3.15 will result in the time map of Figure 3.14 Note that the only interesting difference between this time map and that of Figure 3.16 is that the task corresponding to `scholarlypursuit46` is now constrained to follow John's arrival in New Haven. In this case, the abductive premise is a plausible one since John presumably has control over the start time of a task he is responsible for executing. At other times the decision can be more complex.

Suppose that John also wanted to make sure that his visit to Sterling coincided with a time during which his Yale girl friend was working in the library. Now, even assuming that she is willing to cooperate, it will still require that he get in touch with her, inform her of his plans, and come to some accord over a convenient time. In such cases, making decisions about event orderings may require arbitrariliy complex deductions. The TMM tries to assist in such decision making, but the ultimate responsibility lies with the application program.

In this section we've explored how the TMM can be used to extract information about what is possible, given what is already known. In many situations the possibilities are abundant. A program will need to be able to look at a number of possibilities, choose between them, and then use the chosen one as a basis for making further choices and predictions. The rather complicated bookeeping procedures needed to assist in this process are facilitated by the TMM using abductive answers and the program-mediated deduction techniques described in Section 3.2.

The next section describes some special techniques akin to forward chaining in static data bases developed for reasoning about processes. These techniques will help a great deal in reasoning about the effects of actions in planning.

3.7 Facilities for automatic projection and refinement

It is often convenient to be able to specify routine inferences that are to be performed whenever certain new facts are added to or removed from the data base. Standard forward chaining rules are an example of a technique for performing such inferences. As I mentioned in Section 3.2.4, controlled forward inference of the sort characterized by the pattern `(for-each-answer (fetch` *antecedent-conditions*`) (add` *consequent-predictions*`))` are unsuitable for certain applications due to timing problems. One application in which such timing problems prove particularly irksome concerns reasoning about the effects of actions in planning.

A planner has to take into account the effects of actions proposed as steps in a plan for achieving a given task. However, the specification of a plan should not have to mention those effects of actions (performed as steps in the plan) that have no bearing on the plan achieving its stated objective. It's true that such side effects should be readily available as they are likely to figure prominently in integrating a set of steps into a larger plan. It's unreasonable to expect, however, that the plan specification for achieving an isolated task attempt to anticipate all possible interactions by providing a detailed account of the physics involved in carrying out the plan. The underlying physics should be represented elsewhere. A planner should expect that the deductive system responsible for managing its representation of actions, processes, and their effects occurring over time maintain the following invariant: the representation should reflect exactly those effects that are licensed by the rules describing the physics of the world. Relying on the deductive system to maintain a physically consistent view of the world frees the planner to concentrate on figuring out what to do when its actions fail to achieve their intended effects.

This section describes utilities for performing temporal forward chaining. I will begin with a temporal analog of (-> P Q) and then describe methods for reasoning about change, the effects of actions, and causal relationships.

3.7.1 Implications involving the intersection of overlapping time tokens

A temporalized version of (-> P Q) might be accomplished by something of the form:

```
(-> (time-token P ?tok1)
    (assert (and (time-token Q ?tok2)
                 (coincident (begin ?tok1) (begin ?tok2))
                 (coincident (end ?tok1) (end ?tok2)))))
```

But this will do what is expected only if P denotes a literal. What if we want a rule of the form (-> (and P_1 ... P_n) Q)? First of all, what do we mean by this? A reasonable interpretation might be, whenever P_1 through P_n are simultaneously true, you can infer Q. That is, given any set of n tokens such that the first has type P_1, the second P_2 and so on up to P_n, then if the intersection of their corresponding intervals is nonempty, we can construct a new time token of type Q whose endpoints are coincident with the interval of intersection. The resultant time token with schema Q is said to be a *generated token* and is given special treatment by the TMM.

To distinguish temporal forward-chaining rules from simple forward-chaining rules, the TMM requires the following form:

```
(->t PROP PROP)
```

where conjunctions in the first argument place are handled correctly by the system. In the TMM a rule of the form (->t (and $P_1 \ldots P_n$) Q) sets up a series of pattern-invoked procedures which ensure that for every set of tokens satisfying the above interpretation, a new token is constructed with the appropriate schema and restrictions on its duration and start time. These procedures are invoked only in situations where a nonempty intersection is possible, and thus potentially avoid a lot of unnecessary work. The actual mechanics of temporal forward chaining are rather complicated. Apart from avoiding work on sets of tokens that have an empty intersection, there are issues of how the duration of a generated token is to be restricted, how generated tokens should be handled by the persistence clipping machinery, and what happens when the conditions that allowed us to generate a particular token change and no longer warrant the token (for instance when some token is retracted or clipped, resulting in an empty intersection). These issues will be dealt with in detail in Chapter 4.

Temporal forward chaining-rules can also be temporally "gated". That is to say you can temporally restrict the interval of time over which a rule is to apply. An assertion of the form (->t (and $P_1 \ldots P_n$) Q) is timeless; it applies to any conjunction of appropriate tokens. An assertion of the form (time-token (->t (and $P_1 \ldots P_n$) Q) *tokenname*) applies only within the interval (begin *tokenname*) to (end *tokenname*). As an example consider the following assertion:[11]

```
(time-token (->t (and (flashing ?warning-indicator)
                      (instance-of ?warning-indicator radiation-monitor)
                      (color ?warning-indicator red))
                 (radiation-level high))
            power-plant-visit35)
```

This assertion says that throughout the interval associated with token power-plant-visit35, one can conclude that the level of radiation is high during any interval in which a radiation monitor is both flashing and red. (Assume that it could be flashing and yellow

[11] The property instance-of is normally considered to be timeless. Stipulating it as such in the TMM (*i.e.*, setting the 'tpred property of 'instance-of to be 'timeless) can save a considerable amount of storage in the time map.

to make it interesting.) In most cases this sort of temporally gated rule can be replaced with a similar rule with another conjunct (*e.g.*, (location self ThreeMileIsland)). But there are situations where the number of additional conjuncts required to limit the scope of a rule is sufficiently large, and their simultaneous occurrence sufficiently rare, that it is more efficient to use a gated rule.

3.7.2 Representing and reasoning about the physics of a domain

In addition to ->t, which supports a limited form of logical implication, there is also a method for setting up rules that can be used to model the physics of a domain or project the consequences of holding certain beliefs. Suppose you believe that every time you're irritable and hungry and get in a checkout line at the supermarket, the cash register in that line will surely go on the fritz. You could represent this inevitable fact as:

```
(pcause (and (irritable ?who)
             (hungry ?who))
        (queue-up ?who ?checkout-aisle)
        (operational-status (register ?checkout-aisle) down))
```

This rule is interpreted as saying that queueing up at a checkout line when you're irritable and hungry causes the register to be in a nonfunctioning state. Obviously, the connection between my physical and mental state and the operation of a cash register is not likely to be causal, except in my warped perception of how the world works. Predications involving pcause are not restricted to any particular notion of causality. The predicate name pcause is used primarily for historical reasons [McDermott 82] [Allen 83]; Pcause refers to persistence causation. The semantics are quite clear. The first argument is a temporal precondition, generally a conjunction of fact types. The second argument is generally taken to be an event type, the so called triggering event. The last argument is a fact type, the resulting persistence. A pcause rule states that if there exists a token of the given event type such that the temporal precondition is true throughout the event interval, then one can infer a new token of the stated fact type whose beginning is coincident with end of the event interval. Temporal reason maintenance sees to it that exactly those tokens are IN as are licensed by the rules describing the physics of the domain and the conditions (premised tokens and constraints on their occurence) specified by the user.

Pcause rules are a special case of a more general class of rules called "auto-projection" rules. Auto-projection rules perform a fairly large subset of those inferences that can be performed using controlled forward inference, but without abductive inference and without the usual timing problems associated with controlled forward inference. To begin the discussion of the general form, I'd like to point out some of the more obvious limitations of pcause rules.

The following rule (which you'll see again shortly in Section 3.7.3) might be used as a means of capturing the physics of completed parts leaving a production unit in the factory domain:

```
(pcause (and (production-status (sorter ?proc) running)
             (production-status (conveyor ?proc) running))
        (batch-process ?proc ?lot)
        (location ?lot (staging-area ?proc)))
```

This rule says that, if the sorter and the conveyor are running throughout the period of a batch production process, then following the process the production lot will be located in the production unit's staging area. If we turn off the query mechanism's ability to add abductive constraints, then the following LISP fragment would perform a service similar to the above pcause rule disregarding timing considerations:

```
(for-each-answer
  (fetch '(and (time-token (batch-process ?proc ?lot) ?trigger)
               (tt (begin ?trigger) (end ?trigger)
                   (production-status (sorter ?proc) running))
               (tt (begin ?trigger) (end ?trigger)
                   (production-status (conveyor ?proc) running))))
  (add '(and (time-token (location ?lot (staging-area ?proc)) ?result)
             (elt (distance (end ?trigger) (end ?result) 0 0)))))
```

But what if it was only necessary that the conveyor be running when the production process finished? The only thing that would have to be changed in the controlled forward inference version is to replace the second "true throughout" conjunct with a "true at an instant" conjunct: `(t (end ?trigger) (production-status (conveyor ?proc) running))`. What if there was a delay of between 5 and 10 minutes from the time when the production process finished to the time when the production lot arrived in the staging area? Again this would be trivial to change in the controlled forward inference version by simply modifying the constraint assertion. The TMM has a routine for setting up this sort of rule. The format is similar to that for controlled forward chaining, but it is in the form of a rule that is asserted to the data base:

(auto-project *time-token-trigger-schema antecedent consequent*)

The *time-token-trigger-schema* describes the triggering event type and provides a variable to refer to the corresponding token. The *antecedent* conditions consist solely of tt and t predications and the *consequent* prediction is any assertable schema including calling LISP functions using the call pseudo-predicate. The modified pcause rule with the 5 to 10 minute delay and the modified prerequisite would be:

```
(auto-project
  (time-token (batch-process ?proc ?lot) ?trigger)
  (and (tt (begin ?trigger) (end ?trigger)
           (production-status (sorter ?proc) running))
       (t (end ?trigger) (production-status (conveyor ?proc) running)))
  (and (time-token (location ?lot (staging-area ?proc)) ?result)
       (elt (distance (end ?trigger) (end ?result) 5 10))))
```

This is still quite restricted. For instance, you're not allowed to have constraints of the form (elt (distance pt1 pt2) *low high*) in the antecedent conditions of an auto-projection rule. This would preclude saying, for example, that the conveyor need only be running during the last five minutes of the production process. This sort of rule can be constructed, but there is as yet no (released) general format for doing so in the TMM. Auto-projection rules do allow for a surprising range of uses, however. As an example, consider the following scenario.

Suppose that you have a rule that says, whenever you have a shipment going out on a day that the government safety inspectors are visiting, then you should notify the customer in advance that there may be some delays, and if at all possible have the client make a specific appointment for pickup. The notification should precede the day of pickup by some factor dependent upon the customer's distance from the factory. This could be made more complex by attempting to merge this task with that of informing the customer when to pick up his order, but I'll ignore that complication. This rule is different from others discussed in that it projects backward in time. Such rules are said to set up *anachronistic* data temporal dependencies[12] since the consequent token depends upon tokens which follow it in time. The corresponding auto-projection rule would look like:

```
(auto-project
 (time-token (pick-up ?order ?client) ?trigger)
 (tt (begin ?trigger) (end ?trigger) (inspection-tour ?agency ?purpose)))
 (and (time-token (notify ?client !<(delay (loading-step ?trigger))>) ?result)
      (active-task ?result)
      (travel-time-estimate ?client factory ?low ?high)
      (elt (distance (end ?result) (begin ?trigger)) *neg-inf* ?low)))
```

[12] The term is due to Stan Letovsky.

The conjunct (active-task ?result) is used to inform the planner that it must plan out the details of this task. Rules of this sort could be used to notice the need for all sorts of tasks, including tasks whose purpose it is to prevent the occurrence of the very events that prompted them into being in the first place (see [Dean 85]).

I'll conclude this section with a demonstration of temporal forward chaining. The example will be a bit more complex than the previous examples in an effort to impress the reader with the capabilities of the machinery developed thus far.

3.7.3 A demonstration of temporal forward chaining

Consider the following scenario involving reasoning about production units in a semi-automated factory. In this factory there are two separate production lines. Batches of parts manufactured on one line can be transfered to another via transfer units. Different processes on each production line try to appropriate resources wherever they can find them. Conflicts naturally arise and have to be resolved. Figure 3.17 depicts the general layout of the factory.

The example considered in this subsection concerns the problem of taking into account the simple physics of the factory domain. In the course of routine manufacturing chores, objects get shuttled around from one location to another according to the physics of robot manipulators and automated transfer units. Much of this shuttling about occurs as side effects of the planner's actions to achieve specific tasks. Only occasionally do these side effects explicitly enter into the plans of the robot controlling production in the factory. Traditionally, modeling the physics of a domain had to be handled by special plans that took into account the prerequisites and consequences of the actions of other agents and processes. These actions had to planned for, just like any other action being contemplated by the planner itself, with all the attendant problems involving plan failure, backtracking, and debugging. In the example that follows we'll see that a certain amount of this drudgework can be eliminated (*i.e.*, carried out by the TMM without the supervision of the planner). The underlying physics of robot arms and automated material handlers is modeled by asserting a set of forward chaining rules.

The following rule says that if both the sorter and the conveyor for a given batch processor are in operation during a manufacturing run of the batch processor then the finished batch, called a "lot", will be located in the batch processor's staging area when the

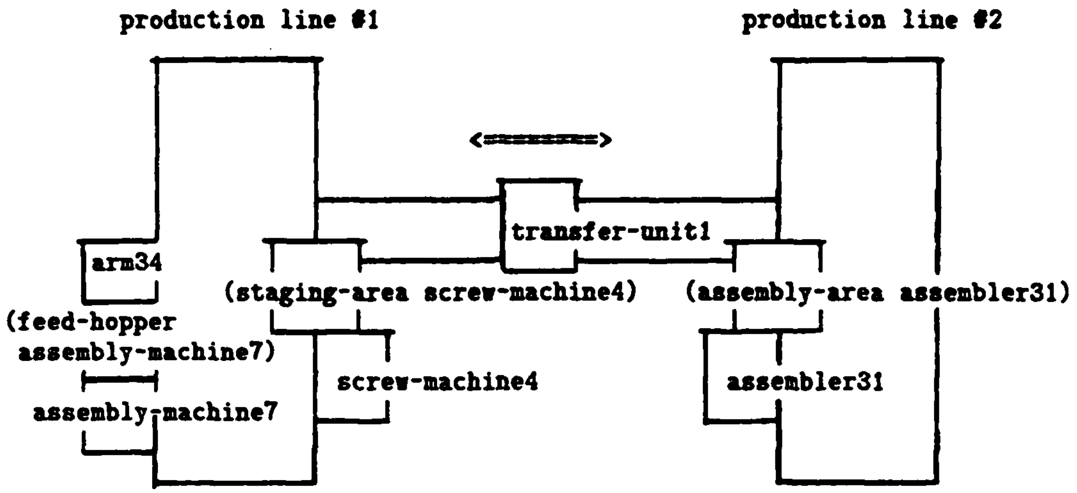


Figure 3.17: General layout of the factory domain

run is complete (if one of the preconditions is not met, then the finished batch will likely be scattered on the floor at the base of the batch processor).

```
(pcause (and (production-status (sorter ?processor) running)
             (production-status (conveyor ?processor) running))
        (batch-process ?processor ?lot)
        (location ?lot (staging-area ?processor)))          rule #1
```

In order to make use of a given object, one prerequisite is that it be accessible from the location where it is to be used. An object is accessible from a location, loc1, if the desired object is either at loc1 or there is a running transfer unit connected[13] from loc1 to a second location, loc2, and the object is at loc2.

```
(->t (and (location ?obj ?loc1)
          (production-status ?transfer-unit running)
          (connects ?transfer-unit ?loc1 ?loc2))
     (accessible-from ?obj ?loc2))                          rule #2
```

If an event of the form (mass-transfer ?transfer-unit ?loc1 ?loc2) occurs, then all the objects which prior to the transfer were at ?loc1 are moved to ?loc2. In a mass-transfer the transfer unit just sweeps everything at the initial location into a bag and then deposits them all at the new location.

[13] The predicate connects is declared timeless for this exercise.

```
(pcause (and (production-status ?transfer-unit running)
             (connects ?transfer-unit ?loc1 ?loc2)
             (location ?obj ?loc1))
        (mass-transfer ?transfer-unit ?loc1 ?loc2)
        (location ?obj ?loc2))                              rule #3
```

There is also a rule that captures the effect of a robot manipulator moving a single object from one location to another.

```
(pcause (location ?obj ?loc1)
        (item-transfer ?manipulator ?obj ?loc1 ?loc2)
        (location ?obj ?loc2))                              rule #4
```

The following rule states that an object cannot be in two places at once.

```
(<- (contradicts (location ?obj ?loc1)
                 (location ?obj ?loc2))
    (thnot (:= ?loc1 ?loc2)))                               rule #5
```

All the machines have been running for the last 24 hours and no down time is expected in the immediate future. The initial conditions are:

```
(time-token (production-status transfer-unit1 running) running1)
(time-token (production-status (sorter screw-machine4) running) running2)
(time-token (production-status (conveyor screw-machine4) running) running3)
(elt (distance (begin running1) *ref*) (minc 24:00) (minc 24:00))
(elt (distance (begin running2) *ref*) (minc 24:00) (minc 24:00))
(elt (distance (begin running3) *ref*) (minc 24:00) (minc 24:00))
```

Approximately two hours before the current reference point the stock room ordered a batch of set screws made. The set screws are manufactured on the first of the two production lines. We assert:

```
(time-token (batch-process screw-machine4 set-screw-lot47) batch69)
(inst set-screw-lot47 (lot set-screw))
(elt (distance (begin batch69) (end batch69)) 15 20)
(elt (distance (begin batch69) *ref*) (minc 2:00) (minc 2:00))
```

Additionally, there are two outstanding tasks to be completed. The first task **task1** requires that the staging area for the screw machine (an area shared by several machines) be cleared in preparation for an assembly task. The second task **task2** involves appropriating an additional supply of set screws for a job already in progress.

```
Frame of reference: *ref*  Scale: 0.5

running1 (production-status transfer-unit1 running)
<|‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>
running2 (production-status (sorter screw-machine4) running)
<|‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>
running3 (production-status (conveyor screw-machine4) running)
<|‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>
batch69 (batch-process screw-machine4 set-screw-lot47)
   |--------|‾|
TOKEN1 (location set-screw-lot47 (staging-area screw-machine4))
          |‾‾||‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>
TOKEN2 (accessible-from set-screw-lot47 (assembly-area assembler31))
          |‾‾||‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>
task2 (appropriate (lot set-screw) job45)
                                   |--|‾‾|
task1 (clear-working-surface (staging-area screw-machine4))
                                                    |--------|‾|
```

Figure 3.18: Initial time map for the wandering-set-screws example

```
(time-token (clear-working-surface (staging-area screw-machine4)) task1)
(time-token (appropriate (lot set-screw) job45) task2)
(elt (distance (begin task1) (end task1)) 15 20)
(elt (distance (begin task2) (end task2)) 5 10)
(elt (distance (begin task1) *ref*) (minc 0:30) (minc 0:30))
(elt (distance (begin task2) *ref*) (minc 1:00) (minc 1:00))
```

The resulting time map is shown in Figure 3.18. Notice that two new tokens, TOKEN1 and TOKEN2, have been generated by the TMM in addition to the ones we have asserted. These tokens correspond to the results of our first two rules firing (rule #1 determines where the result of a batch process ends up, and rule #2 determines if an object is accessible from a location neighboring the location where it is currently situated).

Now we expand task1: the task to clear everything from (staging-area screw-machine4) in preparation for an assembly task. To use the mass-transfer primitive all that is required is that there be a working transfer unit connected from the staging area of the screw machine to some other location. The expansion is carried out by the following code:

```
(for-first-answer
 (fetch
  (and (time-token (clear-working-surface (staging-area screw-machine4)) ?tok)
       (connects ?transfer-unit (staging-area screw-machine4) ?some-other-loc)
       (thnot (:= (staging-area screw-machine4) ?some-other-loc))
       (tt (begin ?tok) (end ?tok)
           (production-status ?transfer-unit running))))
 (add '(and (time-token (mass-transfer transfer-unit1
                                            (staging-area screw-machine4)
                                            ?some-other-loc)
                        transfer54)
            (elt (distance (begin transfer54) (end transfer54)) 5 5)
            (elt (distance (end transfer54) (end ?tok) 0 0)))))
```

Figure 3.19 shows the time map after the expansion. Notice that rule #3 has done its job, and `set-screw-lot47` which was in `(staging-area screw-machine4)` previous to the `mass-transfer` action is now in `(assembly-area asembler31)`. The system has generated a new token `TOKEN3` denoting the fact of the set screws being at the new location, and clipped the persistence of `TOKEN1` (and indirectly `TOKEN2` as well) which indicated the fact of their being at the old location.

Next, the appropriation task (`task2`) is expanded. The preconditions for the plan of using a fixed manipulator to simply reach out and grasp a bunch of set screws requires that there be a suitable bunch of set screws in the nearby staging area of `screw-machine34`. If these conditions are met, executing the `item-transfer` primitive will effect the required appropriation. The following code sees to this:

```
(for-first-answer
 (fetch
  (and (time-token (appropriate (lot set-screw) job45) ?tok)
       (inst ?lot (lot set-screw))
       (tt (begin ?tok) (end ?tok)
           (location ?lot (staging-area screw-machine4)))))
 (add '(and (time-token (item-transfer arm34 ?lot
                                       (staging-area screw-machine4)
                                       (feed-hopper assembly-machine7))
                        transfer32)
            (elt (distance (begin transfer32) (end transfer32)) 2 2)
            (elt (distance (end transfer54) (end ?tok) 0 0)))))
```

The results of expanding `task2` are shown in the time map of Figure 3.20. The system-generated token, `TOKEN3`, is now labeled as OUT. This token was generated by the rule for projecting the effects of a `mass-transfer` action, but the conditions under which that rule was originally fired have changed. Before we expanded `task2`, `set-screw-lot47` could be

```
Frame of reference: *ref*  Scale: 0.5

running1 (production-status transfer-unit1 running)
<|‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>
running2 (production-status (sorter screw-machine4) running)
<|‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>
running3 (production-status (conveyor screw-machine4) running)
<|‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>
batch69 (batch-process screw-machine4 set-screw-lot47)
   |--------|‾|
TOKEN1 (location set-screw-lot47 (staging-area screw-machine4))
          |‾‾||‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾|
TOKEN2 (accessible-from set-screw-lot47 (assembly-area assembler31))
          |‾‾||‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾|
task2 (appropriate (lot set-screw) job45)
                                   |--|‾‾|
task1 (clear-working-surface (staging-area screw-machine4))
                                                   |--------|‾|
transfer54 (mass-transfer transfer-unit1 (staging-area screw-machine4)
                          (assembly-area assembler31))
                                                        |‾‾|--|
TOKEN3 (location set-screw-lot47 (assembly-area assembler31))
                                                           |‾||‾‾‾‾‾‾‾‾‾‾‾>
```

Figure 3.19: The set-screws example after the clear-working-surface task is expanded

predicted to be in the staging area of `screw-machine4` right up until the `mass-transfer` action was executed. Hence, the mass transfer was predicted to result in the set screws being tranported to the assembly area of `assembler31`. After the expansion of `task2`, `set-screw-lot47` is no longer predicted to be in the staging area of `screw-machine4` at the time the `mass-transfer` action is executed; the set screws were moved to the feed hopper of `assembly-machine7` by the `item-transfer` primitive. If we had expanded `task2` before expanding `task1`, then `TOKEN3` would never have been generated.

It would now appear that the mass transfer action was executed to no purpose. It would seem that one prerequisite warranting the use of a mass transfer action be the fact that there is something at the location from which the transfer originates (*i.e.*, this location is not clear). If the working surface to be cleared is already cleared, then there is nothing to do. In a situation in which a location can have any number of objects sitting on it, the task of determining that an area is clear throughout an interval is more difficult than one might imagine. Conceptually it is quite easy; computationally it is rather messy. In Chapter 4 (Section 4.7.1) we'll see how the time map manages it.

Now in the example above, no one really cared whether `set-screw-lot47` was shuttled around from the staging area of `screw-machine4` to the assembly area of `assembler31`. The shuttling was just a side effect of the `mass-transfer` action. One can easily imagine, however, a situation in which it would make a difference. Suppose that another machine located on the second production line also needed some set screws. Suppose further that the corresponding task for appropriating them, call it `task3`, is expanded after `task1` and before `task2`. Now `task3` will surely take advantage of the fact (predicted when `task3` is expanded) that `set-screw-lot47` is located in `(assembly-area assembler31)` at the time[14] the set screws are needed. Task3 is said to depend upon this fact, and a protection is set up during plan expansion to monitor whether or not it continues to hold. Unfortunately, after `task2` is expanded this fact will no longer be true. When `task2` is finally expanded it will cause a protection violation. This in turn should alert the program to resolve the dispute over what production process has the most legitimate claim to `set-screw-lot47`.

Most of the machinery is now in place for understanding the role of temporal imagery in planning. Let's consider the overall strategy suggested by the techniques presented thus far.

When you make a change to the data base (either the addition of new information or

[14] This refers to plan execution time.

```
Frame of reference: *ref*  Scale: 0.5

TOKEN3 (location set-screw-lot47 (assembly-area assembler31)) OUT
<>|‾---------------------------------------------------------------------->
running1 (production-status transfer-unit1 running)
<|‾----------------------------------------------------------------------->
running2 (production-status (sorter screw-machine4) running)
<|‾----------------------------------------------------------------------->
running3 (production-status (conveyor screw-machine4) running)
<|‾----------------------------------------------------------------------->
batch69 (batch-process screw-machine4 set-screw-lot47)
   |--------|‾|
TOKEN1 (location set-screw-lot47 (staging-area screw-machine4))
          |‾‾||‾---------------------------|
TOKEN2 (accessible-from set-screw-lot47 (assembly-area assembler31))
          |‾‾||‾---------------------------|
task2 (appropriate (lot set-screw) job45)
                                   |--|‾‾|
transfer32 (item-transfer arm34 set-screw-lot47 (staging-area screw-machine4)
                          (feed-hopper assembly-machine7))
                                     |‾‾|-|
TOKEN4 (location set-screw-lot47 (feed-hopper assembly-machine7))
                                      |‾‾||‾----------------------------->
task1 (clear-working-surface (staging-area screw-machine4))
                                                   |--------|‾|
transfer54 (mass-transfer transfer-unit1 (staging-area screw-machine4)
                          (assembly-area assembler31))
                                                        |‾‾|--|
```

Figure 3.20: The set-screws example after the appropriation task has been expanded

the removal of old), the TMM attempts to reorganize things to accommodate the change. Temporal forward-chaining rules (*e.g.*, pcause auto-projection rules) assist in this reorganization by modeling some of the underlying physics of the domain (*e.g.*, how objects move around under the influence of various processes). When this reorganization is complete, the repercussions that matter (those that the user has indicated using the change-driven interrupts) are available for use as a summary of, or response to, the important developments in the data base (change-driven interrupts can be used in either capacity). It is the user's responsibility to decide what to do about these new developments. The system is only required to provide an accurate picture (time map) of the implications of what the user believes. A planner proceeds by providing more and more detail concerning how it intends to accomplish its current set of tasks. The "repercussions that matter" typically involve interactions between various steps in the plan proposed thus far. The time map represents how the proposed plan might turn out if it was executed as is. In planning the TMM assists in choosing more detailed accounts of individual tasks, and in detecting interactions between tasks. Chapter 5 describes how temporal imagery can be applied in the design of systems for solving robot problem solving tasks.

3.8 Reasoning about choices in planning

The problem of making choices is central to automated reasoning. Choosing the right alternative (plan, hypothesis, context, or whatever) generally requires considerable computation, foreknowledge, or some combination of the two. There are times, however, when it is convenient to delay making a choice, thereby leaving one's options open. The hope is that, in the course of subsequent planning and information gathering, the right choice will become clear. It is often important to do more than simply put off making a choice. In order to decide upon a reasonable course of action, it is useful to keep track of several alternatives simultaneously. In the course of refining these alternatives, it may be discovered that some alternatives complement one another (providing opportunities to consolidate effort and conserve resources) while others interact disadvantageously (leading to undesirable consequences of one sort or another). Such discoveries provide direction in choosing a "reasonable" set of alternatives. In general, the alternatives suggested by a given situation are abundant. Keeping track of large numbers of alternatives eats up storage and processing time, and that presents us with a bit of a dilemma: it's hard to make choices and it's expensive to put off making them. Since planning itself consumes resources (most notably time and storage) it behooves us to come up with a reasonable strategy for extricating ourselves from this dilemma. The general form of such a strategy is clear:

> If there are two or more alternatives, both of which have been demonstrated to pay off in circumstances "similar to those at hand", but choosing the "wrong" one will adversely affect the outcome of the overall plan, then split the world and consider both alternatives simultaneously. Upon expanding certain other tasks which, for example, vie for common resources, make a decision on the basis of whatever opportunities for "optimization" have presented themselves, and eliminate from consideration the alternatives that were not chosen.

As a simple example, suppose that a planner has several jobs that can be handled by either a lathe or a milling machine. In many situations, it won't be clear how to go about using the available machines efficiently without actually exploring some of the possibilites (*i.e.*, expanding the tasks in various ways). One strategy might be to expand the tasks considered first both ways (using the lathe as one alternative and the milling machine as the other). *When the other tasks are considered, it should be clear which of the initial* expansions are better. By having the alternatives laid out before you, it is obviously simpler to take advantage of opportunities for consolidating effort and conserving resources.

Of course, the details involved in making such strategies work in practice are likely to be messy. Determining when to consider more than one alternative, what makes one alternative better than another, and when to eliminate all but one competing alternative from consideration will require a considerable amount of domain specific knowledge. Here we will be concerned primarily with methods for efficiently representing and maintaining a number of alternatives simultaneously.

In addition to reasoning about alternative outcomes that a planner has control over, it is also useful to keep track of situations that the planner has little or no control over, but whose outcomes it is unsure about. By anticipating various outcomes, a planner might prepare itself to deal with such situations no matter how things turn out.

Consider the following scenario. A major manufacturer of personal computers, Mega Computers, is considering bids from semiconductor firms for memory chips for its new business computer. One of the companies submitting bids, call it C1, has faster chips than any of its competitors, but it's not clear that this young company will be able to meet the necessary production requirements. Another company C2, which you happen to own stock in, is also submitting a bid, but with run-of-the-mill chips. C2, however, will have no trouble keeping up with Mega Computer's production schedule. You've been told by insiders that Mega Computer will either buy from C1 or from C2, but that it is waiting for the results of a survey (concerning whether or not buyers are sufficiently influenced by machine speed specifications) before it makes its final decision. You want to prepare for either outcome. If C1 gets the contract, its stock will rise sharply and probably keep on climbing for a considerable period. You want to be prepared to take advantage of this so you tell your broker to liquidate some other stocks in preparation for a quick purchase. If, on the other hand, C2 gets the contract, its stock will probably experience a brief upswing and then return to previous levels. You'd just as soon have your money invested in more ambitious companies, so you tell your broker that in the event C2 gets the contract, sell your stock as soon as C2 appears to peak.

In this scenario you are unsure about the outcome of a particular event. In order to be ready to respond to either of the two anticipated outcomes, it is necessary to do some preliminary work to guarantee your preparedness (*e.g.*, liquidate some stock to provide working capital or notify your broker to sell). You might also want to put in motion certain tasks to determine which of the two outcomes will actually occur (*e.g.*, try to get hold of the results of Mega Computer's survey or even conduct your own survey). You might even try to influence the outcome (*e.g.*, educate the business community in the advantages of high speed machines).

The same techniques used for handling disjunctions of the sort illustrated in the above example have also been applied to reasoning about counterfactuals [Dean 85]. Suppose that you're trying to prevent the occurrence of an event E. In order to ensure that an action A is effective in preventing E, you have to see to it that in the situation where you actually execute A, E fails to materialize. In order to monitor the warrant for A, you want to make sure that if you fail to execute A, then E will indeed occur.

The solution proposed here for handling the above reasoning tasks revolves around maintaining several (partial) descriptions of the world simultaneously. The key to doing this efficiently involves a method of labeling assertions with the choices that they depend upon. This section should give the reader a feel for some of the issues involved and a glimpse of some of the problems that still have to be solved before these techniques are shown to have any advantage over more traditional approaches for dealing with choices (*e.g.*, backtracking or other sorts of enumeration schemes).

3.8.1 Maintaining several partial world descriptions simultaneously

A ***partial world description*** is just a set of event and fact tokens together with certain constraints on their occurrence and duration. Such a description may describe a large number of possible outcomes involving various restrictions on the partial order of tokens. The time map, as it has been presented so far, essentially maintains a single partial world description (or PWD). The only disjunctions represented in a single PWD involve the order of tokens, and a partially ordered time map provides a weak basis for projecting consequences. If there are two tokens unordered with respect to one another, one of type P and a second of type (not P), then it may be impossible to determine whether or not P is true at a point in time following both tokens.

Traditional context mechanisms, such as the one employed in the SIPE planner [Wilkins 84], provide a suitable basis for projecting consequences, but are inappropriate for effectively reasoning about alternatives. In the context-based approach, a planner is forced to construct a context corresponding to a given set of choices. Then, in order to notice interesting consequences of that set of choices, the planner must choose to reason in that context and ask the right questions. What we really want is a mechanism that suggests a suitable context (or set of choices) in response to a general question about what is possible given the options currently available. If I have the option of either eating at home or going to a resaurant this evening, and I want to know if I'll be at home to receive a long distance call between 5:30 and 6:30, then I would like to notice that this is possible only if I choose to eat at home.

Not all partial world descriptions are worthy of consideration. To begin with, mutually exclusive alternatives should be kept distinct (*e.g.*, I can't both drive to work and take the bus). In addition, I will probably want to rule out certain combinations of alternatives (*e.g.*, I can't drive to work and expect that my wife will be able to pick up her mother at the airport). A single alternative is referred to as an ***option***. A set of mutually exclusive options is referred to as a ***choice set***. A conjunction of options to be ruled out is called a ***nogood***. A ***choice assignment*** involves assigning a boolean value 1 (for choose this option) or 0 (for don't choose this option) to every option being considered.

The way in which we keep track of several partial world descriptions simultaneously involves labeling all assertions (*i.e.*, ddnodes) in the data base with boolean combinations of options. An assertion is *licensed* by a given choice assignment just in case the assertion's label evaluates to 1 under that assignment. Choice sets and nogoods constitute constraints on what is considered to be an admissible choice assignment. Exactly one option of a choice set must be assigned 1; all others must be assigned 0. At least one option of a nogood must be assigned 0. An *admissible* choice assignment is one that satisfies the constraints defined by the choice sets and nogoods.

The TMM employs a specially designed RMS (described in Section 4.6) to maintain a consistent and well-founded labeling of all assertions in the data base. Defining exactly what is meant by consistent and well-founded in this context will have to wait until Chapter 4. For our purposes, a labeling is consistent and well-founded just in case the label of each assertion in the data base accurately reflects the dependence of that assertion on the current set of options, as recorded in the network of justifications.

A simple example should help to make this a bit clearer. I'll begin with some programming preliminaries. The function `option` takes an s-expression designating an option, creates a ddnode corresponding to that option, and then simply returns the s-expression. The function `option` is similar to add except that it doesn't install any justifications. Ddnodes created by the function `option` are handled somewhat different from other ddnodes. The system sees to it that every ddnode that depends, either directly of indirectly, upon a ddnode corresponding to an option is labeled to reflect this dependence. Options can participate in justifications just like any other assertion (*e.g.*, using the `answer-support` or `for-each-answer` macros). The functions `nogood` and `mutually-exclusive` each take a list of s-expressions designating options, determine if the appropriate corresponding constraint is consistent with the constraints imposed thus far (*i.e.*, those from other choice sets and nogoods), returns `false` if this condition is not met, and returns `true` and adds the corresponding constraint if the condition is met.

Suppose that you are considering how to spend your lunch hour and there are two decisions that you have yet to settle. You're not sure whether to go to the university cafeteria or to an off-campus diner, and you are undecided about whether to go at noon or wait until after 1:00 when the lines are shorter. You also know that the cafeteria closes at 1:00. To specify these conditions:

```
Data dependency nodes:
    n1  -->  (= lunch-spot cafeteria)
    n2  -->  (= lunch-spot diner)
    n3  -->  (lunch-during prime-time)
    n4  -->  (lunch-during off-peak)
    n5  -->  (impatient (management lunch-spot))
    n6  -->  (free-refills lunch-spot)
    n7  -->  (good-company lunch-spot)
    n8  -->  (linger-over-lunch lunch-spot)

Inferential connectivity involving ddnodes other than options:
    Node:   Justifications:                Label:
     n5       (({n2}{}))                     (({n2}{}))
     n6       (({n1}{}))                     (({n1}{}))
     n7       (({n3}{}))                     (({n3}{}))
     n8       (({n6,n7}{n5}))                (({n1,n3}{n2}))

Admissible choice assignments:
      {(1 => n1) (0 => n2) (1 => n3) (0 => n4)}
      {(0 => n1) (1 => n2) (1 => n3) (0 => n4)}
      {(0 => n1) (1 => n2) (0 => n3) (1 => n4)}
```

Figure 3.21: Simple dependency structures involving options

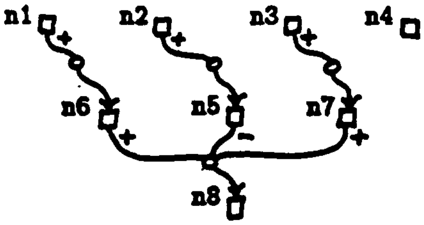


Figure 3.22: Dependency network for the lunch example

```
(mutually-exclusive (list (option '(= lunch-spot cafeteria))
                          (option '(= lunch-spot diner))))
(mutually-exclusive (list (option '(lunch-during prime-time))
                          (option '(lunch-during off-peak))))
(nogood '((= lunch-spot cafeteria))
        (lunch-during off-peak))
```

For some of the alternatives there are implications. If you go to the cafeteria, you can get free refills on coffee. If you go to the diner you won't be able to dawdle, as the management doesn't appreciate people tying up counter space. If you eat during prime time, noon until about 1:00, you're assured of meeting someone to strike up a conversation with, no matter where you go.

```
 (answer-support (+ '(= lunch-spot cafeteria))
        (add '(free-refills lunch-spot)))
 (answer-support (+ '(= lunch-spot diner))
        (add '(impatient (management lunch-spot))))
 (answer-support (+ '(lunch-during prime-time))
        (add '(good-company lunch-spot)))
```

Finally, we have the inference encoded in the data dependency network that, if you have good company and free refills on coffee, and you have no reason to believe that the management of the spot you've chosen for lunch is pushy, then you can rely upon a leisurely, lingering, meal.

```
 (for-first-answer
    (fetch '(and (free-refills lunch-spot)
                 (good-company lunch-spot)
                 (consistent (not (impatient (management lunch-spot))))))
    (add '(linger-over-lunch lunch-spot)))
```

The ddnodes, their justifications, and their labels are displayed in Figure 3.21 and the data dependency network is shown in Figure 3.22. Consider the label of n8. This can be interpreted as saying that you can linger over lunch if you choose the cafeteria and not the diner and go during the prime time hours. The label tells you what choices you must make to believe the assertion. Figure 3.21 also shows all of the admissible choice assignments.

The function **extract-options** takes an object of data type ANS and returns a boolean formula (in disjunctive normal form) of s-expressions indicating options corresponding to the conjunction of all the labels of in-justifiers indicated by the ANS and the negation of the labels of out-justifiers indicated by the ANS. In the following code fragment:

```
 (for-first-answer
    (fetch '(linger-over-lunch lunch-spot))
    (extract-options ans*))
```

the call (extract-options ans*) returns:

```
(or (and (= lunch-spot cafeteria)
         (lunch-during prime-time)
         (not (= lunch-spot diner))))
```

This sort of interchange might be used to provide a planner or diagnostician with some basis for choosing between its alternatives.

Reasoning about alternatives in problem solving is a continuous process: as we encounter uncertainty in the world we consider various alternative predictions. These alternatives tend to complicate our understanding of the world; they make reasoning more expensive and decisions more difficult, but they also provide us with options. As we gather new evidence and notice interactions between our beliefs, we occasionally rule out certain alternatives that no longer appear likely or that are less appealing than their competing options. This serves to simplify the world and reduce complexity. Once you complicate the world by introducing alternatives, you must be alert for opportunities that will allow you to make a choice between them. Ruling out alternatives is accomplished using the function **nogood**. For example (nogood '((lunch-during prime-time))) has the effect that every admissible choice assignments must assign (lunch-during prime-time) the value 0 and (lunch-during off-peak) the value 1. Notice also that this constraint, in conjunction with the constraints already in force, also implies that that every admissible choice assignments must assign (= lunch-spot diner) the value 1 and (= lunch-spot cafeteria) the value 0. The TMM can perform this service of ruling out options (and ruling in others) on the basis of the constraints provided in nogoods and choice sets, but there are a number of complications. First of all, the general problem of determining whether a set of boolean formulae (nogoods and choice sets) is satisfiable is intractable (exponential in the number of boolean variables or in this case options). A second problem concerns what to do when you discover that there are no admissible choice assignments. How do you go about considering new alternatives? The solution to the first problem is simple. The number of alternatives must be kept small. But this just serves to make the second problem more acute. The mechanism presented here provides a means for implementing certain heuristic techniques for exploring sets of alternatives efficiently. If those techniques fail, then you're still probably better off than you would have been without those techniques. In future work I hope to demonstrate how the information gathered in exploring several alternatives simultaneously can prove useful in recovering from planning failures in novel situations. In the mean time I'll just assume that you never run out of alternatives.

In maintaining and reasoning about several alternatives simultaneously, we're not likely to be interested in a solution that enumerates (either in time or space) all possible combinations of choices. Of course, this is difficult to prevent in general. In the worst case, a set of n choices involving two alternatives apiece will result in something on the order of 2^n distinct possibilities. Storing each of these separately is prohibitive. For the most part, alternatives involving separate choices interact minimally, if at all. In actual practice, the storage required to represent the possibilities involving a set of n binary choices is generally no worse than polynomial. Only in extreme cases are we forced to absorb an exponential overhead. A reasonable solution to the problem of representing several alternatives simultaneously should detect and keep records of only those differences that serve to distinguish various possibilities. For instance, if one of my options involves driving the car to work instead of riding the bus, then whether or not the car is in the driveway may have no bearing on any of my other plans. If, on the other hand, my wife is undecided about whether to pick up her mother at the airport or have her ride the limousine, then it is probably worth noting that the former is likely to fail in the event that I choose to drive to work. The TMM can save a great deal of storage by only noting what things differ between partial world descriptions. All of the assertions that remain invariant across sets of partial world descriptions are shared. Each individual PWD represents a *virtual copy* [Fahlman 79] of these shared assertions along with a set of modifications and additions peculiar to that particular PWD.

3.8.2 Temporal reasoning involving sets of alternatives

All of the basic functionality regarding temporal reasoning in time maps involving a single partial world description must now be extended to reasoning about several partial world descriptions simultaneously. The machinery for exploring hypothetical situations, reasoning with incomplete information, and noticing particular consequences must now take into account sets of alternatives licensing various inferences. In this (sub)section I want to introduce the reader to some of the basic methods for performing temporal inference involving alternatives. In Chapter 5 (Section 5.4) I will demonstrate how these methods might fit into the design of a general purpose planning system. As usual, most of the discussion will center around examples.

Suppose that a planner is undecided about whether to use a lathe or a milling machine for a particular task to be performed in the morning. So it expands two plans: one using `lathe14`, and a second using `milling-machine31`. Setting up the two exclusive plans is

```
Frame of reference: noon Scale: 1.0

idle41 (production-status lathe14 free)
||¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯>
idle43 (production-status milling-machine31 free)
||¯¯¯|
task14 (manufacture widget)
     |--------------|
inuse9 (production-status milling-machine31 in-service)
     ||¯¯¯¯¯¯¯¯¯¯¯¯¯|
idle47 (production-status milling-machine31 free)
                    ||¯|
installation-subtask76 (setup milling-machine31 widget)
     |¯¯¯¯¯¯¯¯¯|----|¯|
installed421 (installed jig34 milling-machine31)
           |----------||¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯>
task13 (manufacture gizmo)
                       |--------------|
inuse11 (production-status milling-machine31 in-service)
                       ||¯¯¯¯¯¯¯¯¯¯¯¯¯|
idle49 (production-status milling-machine31 free)
                                      ||¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯>
service35 (routine-service milling-machine31)
  |¯|-----------|
```

Figure 3.23: Time map in which the milling machine option is chosen for task14

```
Frame of reference: noon Scale: 1.0

idle41 (production-status lathe14 free)
||~~~|
task14 (manufacture widget)
     |--------------|
inuse47 (production-status lathe14 in-service)
     ||~~~~~~~~~~~~~|
idle78 (production-status lathe14 free)
                    ||~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
idle43 (production-status milling-machine31 free)
||~~~~~~~~~~~~~~~~~~~~~|
task13 (manufacture gizmo)
                       |--------------|
inuse11 (production-status milling-machine31 in-service)
                       ||~~~~~~~~~~~~~|
idle41 (production-status milling-machine31 free)
                                      ||~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
service35 (routine-service milling-machine31)
  |~|-----------|
```

Figure 3.24: Time map in which the lathe option is chosen for task14

accomplished as follows:

```
(mutually-exclusive (option '(use lathe-plan task14))
                    (option '(use milling-machine-plan task14)))
(answer-support (+ '(use lathe-plan task14))
                code-for-expanding-task14-with-the-lathe-plan)
(answer-support (+ '(use milling-machine-plan task14))
                code-for-expanding-task14-with-the-milling-machine-plan)
```

Now let's suppose that one subtask in the expansion of the plan for using the milling machine has the effect that a certain fixture jig34 is installed in milling-machine31. At some later point the planner is interested in a plan for some other task task63 that would be facilitated by having jig14 already installed. The planner might make the following query:

```
(for-each-ans (fetch (and (tt (begin task63) (end task63)
                              (installed jig34 milling-machine31))
                          additional-antecedent-conditions))
              some-decision-making-code)
```

If the query succeeds by relying upon the token corresponding to the effect in the expansion of task14 with the milling machine plan, then the planner should be aware of the fact that (use milling-machine-plan task14) is an option that has not been decided upon; the planner should obviously not blithely consent to just any set of options. If (extract-options ans*) were executed in the context of *some-decision-making-code* it would return (or (and (use milling-machine-plan task14)))[15]. If the planner really wants to take advantage of the fact that the milling machine plan for task31 installs jig34, it might at this point commit to this plan and dispense with thinking about the alternative lathe plan for the same task. Commitment of this sort corresponds to ruling out one or more of the alternatives. The system uses this as a cue to perform some house cleaning. For instance if the planner executed (nogood '((use lathe-plan task14))), the assertions corresponding to all of the projection and expansion steps involved in reducing task14 with the lathe plan would be eliminated and all references to the option (use lathe-plan task14) would be expunged. Having eliminated (use lathe-plan task14) you have, in effect, committed to ((use milling-machine-plan task14)).

In the example above, alternatives are eliminated in the course of hypothesis selection. It is also often convenient to rule out alternatives in the course of handling interactions.

[15] Disjunctive normal form is a bit awkward here but still a worthwhile convention to adhere to for handling more complicated situations.

I'll continue with the same example. Starting with a time map containing one task task13 employing milling-machine31, suppose that we expand the two different plans for achieving task14. Suppose further that the robot is told that it must schedule routine service for milling-machine31 before the day is out. There are only two time slots in which the service task service35 can be performed: early in the morning (coincidently during the time task14 is scheduled) and late in the evening. These two alternatives are set up as follows:

```
(mutually-exclusive (option '(schedule service35 (time-slot 8:00AM)))
                    (option '(schedule service35 (time-slot 9:00PM))))
(answer-support (+ '(schedule service35 (time-slot 8:00AM)))
     (elt (distance (begin service35) *ref*) (minc 4:00) (minc 4:15)))
(answer-support (+ '(schedule service35 (time-slot 9:00PM)))
     (elt (distance *ref*  (begin service35)) (minc 9:00) (minc 9:15)))
```

It's not possible to provide a clear picture of the resulting time map in a single diagram. There are actually four admissible choice assignments. To show the whole picture would involve producing one time map for each such assignment. Instead, I have provided two time maps corresponding to the two most interesting possibilities. Figure 3.23 shows a snapshot of the time map assuming that options (use milling-machine-plan task14) and (schedule service35 (time-slot 8:00AM)) are chosen. Figure 3.24 shows a snapshot assuming that (use lathe-plan task14) and (schedule service35 (time-slot 8:00AM)) are chosen. Remember, however, that the actual time map combines the information for all four possibilities in a compact form.

The routine service task is shown in both time maps, but the planner has yet to provide the details of its execution. So now let's suppose that the planner expands the routine service task. One side effect of this task is that milling-machine31 is not available for use (its production-status is offline) during the interval corresponding to service35. Unfortunately, the plan of using milling-machine31 for task14 depends upon milling-machine31 being available for use throughout task14, and service35 and task14 overlap. Noticing such interactions is handled in a manner similar to the situation not involving options. First of all, we assume that the code for expanding plans sets up certain predicates for monitoring the continued validity of the assumptions implicit in plan selection. In this situation, I'll assume that the predication (applicable-reduction task14) is dependent upon (production-status milling-machine31 free) being true throughout task14. Now we want to set up a change-driven interrupt, akin to the if-erased demons of Section 5.2, in order to detect and respond to possible interactions. In dealing with options we're not just interested in an assertion becoming OUT. Now we're interested in the choice assignments which result in its being believable or not. The analogs of if-erased demons in a system maintaining several partial world descriptions are called *if-endangered* demons. For example, the following demon ensures that the *code-to-deal-with-possible-plan-failures* is executed just in case the assertion (applicable-reduction task14) becomes "more OUT".

```
(if-endangered (applicable-reduction ?task)
          code-to-deal-with-possible-plan-failures)
```

By "more OUT" I mean that there is some admissible choice assignment (perhaps several) that (applicable-reduction task14) was believable in (*i.e.*, licensed by), but is no longer. Adding the assertions corresponding to the production-status of milling-machine31 being off-line throughout service35 results in (applicable-reduction task14) no longer being licensed by the choice assignment in which (use milling-machine-plan task14) and (schedule service35 (time-slot 8:00AM)) are chosen. The *code-to-deal-with-possible-plan-failures* can determine the "cause" of the possible plan failure using the function extract-options. In this case, executing (extract-options ans*) in the context of *code-to-deal-with-possible-plan-failures* would return:

```
(or (and (use milling-machine-plan task14)
         (not (schedule service35 (time-slot 8:00AM))))
    (and (use lathe-plan task14)))
```

This indicates that there is an applicable reduction for task14, just in case either the lathe plan is used for task14 or the milling machine plan is used and routine service for milling-machine31 is not scheduled for the 8:00 AM time slot. To simply eliminate the possibility that the routine service task interferes with the milling machine option for the task14, the planner could execute the following call to nogood:

```
(nogood '((use milling-machine-plan task14)
          (schedule service35 (time-slot 8:00AM))))
```

The planner could also simply choose to eliminate either the option to use the milling machine or the option to perform the routine service in the 8:00 AM slot by executing one of:

```
(nogood '((use milling-machine-plan task14)))

(nogood '((schedule service35 (time-slot 8:00AM))))
```

As usual, the decision concerning how to respond to a possible interaction is not simple, and the responsibility for making such decisions lies with the planner, not with the TMM.

3.9 Summary

This chapter has provided an overview of the mechanics of temporal imagery. A language for constructing time maps and querying the TMM about their contents was presented, and elementary examples of its use described. An effort was made to show the reader how the TMM supports projection and refinement in the context of a general approach to automated reasoning described as "shallow". The idea behind shallow temporal reasoning is that prediction and reflection proceed in small steps. Possible sets of modifications to the data base are proposed on the basis of what is already known or what is possible given what is known. Each set of modifications constitutes a hypothesis concerning how the world is. The relative merits of these hypotheses are considered and compared, and one is finally selected. The modifications are incorporated into the time map and their repercussions are determined and dealt with. No individual inference is expected to require a great deal of computation, and generally speaking each individual change is minor. The approach assumes that in the main your first guess is likely to be right and that correcting its minor deficiencies won't require a major overhaul of the data base. Patching is always preferred over scrapping.

The TMM is a data base management system. In designing this system a number of assumptions were made about its potential users. First, the user is not likely to be in possession of complete knowledge about all of the entities stored in the data base. The query mechanism is designed to conduct a dialogue with the user asking for assistance in cases where a step in the deduction is only consistent with the information in the database but not necessarily implied by it. Second, the user is assumed to be fallible and liable to make assertions that contradict earlier ones. He is, however, interested in the continued validity of certain of his beliefs, especially when those beliefs are critical to the success of a plan or the understanding of some phenomenon. Our third assumption, then, is that if the user is interested in the continued validity of a given proposition, then he will make use of the TMM's change-driven interrupt facility to take appropriate actions in the event that the validity of that proposition is threatened. To support this functionality, the TMM has to keep track of the reasons why a proposition was believed in the first place and under what conditions it will cease to be believed. The TMM provides the machinery for keeping track of temporal data dependencies using a generalization of what are called protections in the planning literature. Finally, if the user finds himself in a situation where some number of his beliefs are threatened, the TMM makes it possible for the user to modify the data base in order to reason about a suitable patch. The ability to efficiently modify the data base and keep track of the repercussions of changes is critical to temporal imagery.

Chapter 4

Implementation

4.1 Introduction

The time map, together with the routines that support shallow temporal reasoning about time, constitute a special-purpose data base management system. Many of the issues that are important in static data bases are also relevant here. In addition, however, there are a number of implementation issues that arise specifically due to the temporal aspect of items stored in the data base. One such issue concerns the indexing of temporalized assertions (*i.e.*, time tokens) in order to expedite fetches. This is critical in determining the "when" of fact tokens: the duration of a time token denoting a particular instance of a fact becoming true and its offset relative to other points in the time map. One of the most common operations performed by the time map management system involves finding an interval satisfying a set of temporal constraints such that some fact or conjunction of facts is believed to be true throughout that interval. This operation requires that it be cheap to determine the best bounds on the distance separating pairs of points in the time map. Another issue concerns how one deals with defeasible predictions. How can we extend the functionality of data dependencies in static data base systems [Doyle 79] in order to handle temporalized assertions?

This chapter will deal primarily with low-level implementation details, but it will be helpful to review what "higher" level functionality we are trying to support. Below are listed some of the steps involved in shallow temporal reasoning along with specific functions that the time map management system must support in order to facilitate these steps:

1. hypothesis generation: The system has to be able to handle a range of queries. In particular, it must be able to deal efficiently with conjunctive queries of the form `(tt ?pt1 (and` $P_1 \ldots P_n$`))`. The system is also responsible for setting up dependencies in order to keep track of the warrant for believing assertions made on the basis of the response to queries.

2. hypothesis selection: The selection process is largely domain-dependent, but there are certain general requirements:

   (a) The system must be capable of constructing and isolating two or more separate, and possibly contradictory, hypotheses. Recall that a hypothesis is essentially a set of constraints on the existing partial order of facts and events in the time map such that the constraints are consistent with the existing partial order.

   (b) Because a given hypothesis may commit the user to additional constraints, the system must inform the user of the added commitment and provide some means for investigating the possible implications of making such a commitment.

3. projection and refinement: These operations entail the addition of new time tokens and constraints. Of these two, the addition of new constraints is the most critical. It is the "when" of a time token that is of primary interest, and propagating new constraints is an important part of noticing and responding to certain configurations of facts and events. These operations are also likely to introduce what have been referred to as apparent contradictions: pairs of contradictory tokens that are ordered with respect to one another such that their intervals might possibly overlap. The system is reponsible for resolving such contradictions (where possible) by adding additional constraints to restrict the "when" of the earlier occuring token.

   In addition to what was termed "controlled forward inference" of the sort carried out in shallow temporal reasoning, there are also the temporal analogs of forward chaining rules in static data bases. In the previous chapter, I called these temporal forward chaining rules "auto-projection"rules. Such rules (*e.g.*, `(->t (and P Q) R)` and `(pcause (and P Q) E R))` are employed to capture certain logical and physical laws of the domain. They can assist in causal analysis and have been used to implement a restricted[1] form of envisionment [deKleer 82].

[1] This analysis is restricted in that it is (currently) not possible to reason about feedback.

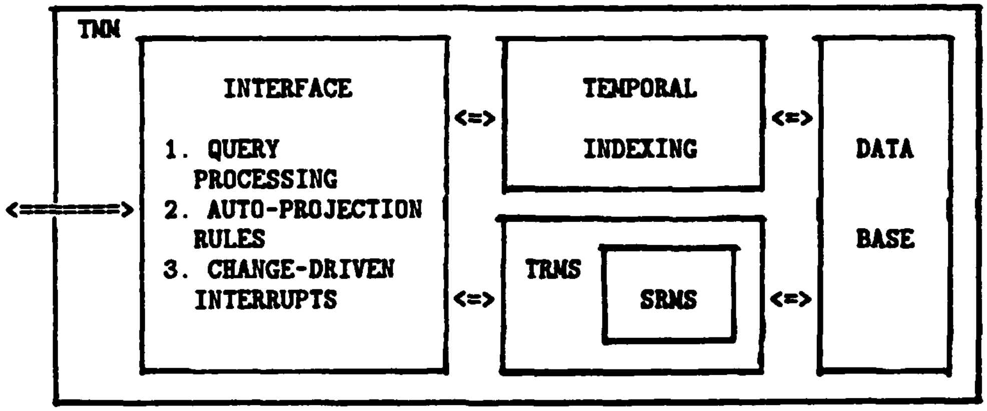


Figure 4.1: TMM architecture

4. handling interactions and assumption failures: Many assumptions may be undermined by a given projection/refinement step. How does the user deal with the incoming barrage of information? What if while dealing with one failure, additional failures occur or other pending failures are resolved thus no longer requiring attention?

In order to describe how these rather high level operations are facilitated, it is necessary to understand how a number of more primitive operations are carried out by the TMM; in particular it is important to understand how the system:

1. determines best estimates on point-to-point distances

2. detects and resolves apparent contradictions

3. monitors protections

4. propagates constraints

5. caches certain derived point-to-point distances to expedite fetches

The first operation is employed frequently in performing the deductions required for processing temporal queries. Time tokens are indexed syntactically by their types and temporally by the network of constraints. The latter index constitutes the "when" or temporal extent of a time token. Computing this index involves determining the best estimates of the distance separating pairs of points in the time map. Operations two through five are handled by a special purpose ***reason maintenance system*** (RMS) [Doyle 80] specifically designed for managing a set of temporalized beliefs. Section 3.2.3 described a general purpose reason maintenance system modeled after Jon Doyle's design. To distinguish the two types of systems, I will refer to a Doyle-type system as a static RMS (or SRMS), and the special purpose system as a temporal RMS (or TRMS). I will also continue to use the generic term "reason maintenance system" (or RMS) to refer to system's like Doyle's [Doyle 79], McAllester's [McAllester 80], and deKleer's [deKleer 84]. Reason maintenance systems are used to keep track of which facts in a data base are currently believed to be true and why[2]. Knowing which facts are believed to be true is obviously useful; knowing why a fact is believed helps in generating explanations of program behavior, debugging data bases, and any number of other handy tasks. The TRMS employs a static RMS as one part of a technique for handling temporal data dependencies. Figure 4.1 shows a rather simplified picture of the TMM architecture.

In the TRMS, a static reason maintenance system is used to maintain a set of timeless beliefs that refer to temporal instan and intervals. These beliefs will include items that you're already familiar with from previous chapters (*e.g.*, constraints, time tokens, and protections) as well as new constructs that are generally hidden from the casual TMM user. Using these basic elements, I will describe how the TMM sets up temporal data dependencies, processes queries, and handles changes to the data base. This last involves updating the status of belief of facts linked by temporal data dependencies. Whether or not P is believed at a given instant may depend upon whether or not Q_1 through Q_n are believed during certain intervals. The TRMS has built into it a method for interpreting temporal information stored in the static RMS and computing the consequences of changes made by the user. Changing a single constraint can result in radical changes to the data base. The TRMS update algorithm is responsible for making those changes. This algorithm is probably the most important idea developed in this chapter, and a considerable amount of time will be spent providing background material, describing the actual algorithm, and proving it correct.

[2] This applies to justification-based systems like Doyle's or McAllester's. It's not quite appropriate for assumption-based systems like deKleer's. DeKleer's system essentially keeps track of the assumptions under which a given fact would be true given that the assumptions were true. See [deKleer 84] for a discussion.

In the following, I will assume that the reader is familiar with the basic ideas employed in static reason maintenance systems. The TRMS used in the current implementation of the TMM is actually built on a hybrid RMS which combines features of McDermott's [McDermott 83] and deKleer's [deKleer 84] systems in order to reason about sets of alternatives simultaneously. The details of this hybrid do not affect the discussion of the TRMS in any substantial way and hence I have chosen to begin by describing the TRMS in terms of a Doyle-type RMS in order to ease the burden on the reader. In Section 4.6 I'll describe the hybrid system in detail. In particular, I will present the requisite algorithms and point out how the implementation employing the hybrid differs from the one described in earlier sections.

This chapter is a compendium of techniques and design decisions. The bulk of it will describe how time maps are implemented and the special purpose data dependency system which supports a form of temporal reason maintenance. This chapter is not meant as a recipe to follow in building a TMM. Rather, it discusses the main issues and strives to point out both the strengths and weaknesses of one particular implementation.

4.2 Time map data types and data structures

As might be expected, there is a close correspondence between the ontological commitments outlined in Chapter 2 and the data types used in the actual implementation. The time map consists of points and constraints upon the distance separating pairs of points. The time map can be viewed as a graph with points as vertices and constraints as directed edges labeled with an upper and lower bound on the distance separating the two points the edge connects. The most important data structures from the user's point of view are time tokens. The corresponding data type is `TOKEN`. An object of data type `TOKEN` has three slots that we'll be considering: `begin`, `end`, and `schema`. The first two are objects of data type `POINT`. The schema slot is an object of data type `PROP` (*i.e.*, a LISP s-expression (a finite non-circular list structure)) that denotes the (logical) type of which the `TOKEN` is a (logical) token and is employed by the system to index the `TOKEN` syntactically in a discrimination network. If the symbol `service31` does not already designate a `TOKEN`, asserting `(time-token (routine-service lathe14) service31)` will result in the creation of a new object of data type `TOKEN` with schema `(routine-service lathe14)`. The symbol `service31` is used to refer to the new object in LISP as well as in the deductive retrieval system[3]. In a query, the term (begin service31) refers to the object of data type POINT that corresponds to the begin slot of service31. The two POINTs, (begin service31) and (end service31), were created when the assertion was made. The predicates begin-token and end-token allow one to go from a POINT to a TOKEN and vice versa. The query (begin-token ?pt service31) will succeed with ?pt bound to something like POINT146 where POINT146 and (begin service31) denote the same thing. Similarly, (begin-token POINT146 ?tok) will succeed with ?tok bound to service31.

The asserting (time-token (routine-service lathe14) service31) also creates a new ddnode with propositional content corresponding to form of the assertion. The new ddnode has a justification constructed from the current answer as described in the previous chapter.

In the current implementation it is assumed that all tokens have schemata that contain no variables. The only exception to this involves the use of temporally "gated" forward chaining rules introduced in Section 3.7. If you allow variables in token schemata, or you employ criteria for determining whether or not a pair of tokens contradict that make use of inconstant properties of terms appearing in token schemata, then persistence clipping becomes considerably more complicated. I won't be considering such cases in this dissertation (but see Section 6.2).

A constraint is implemented as a pair of objects of data type CONLINK. Each CONLINK designates a directed edge between a pair of objects of data type POINT. An object of data type CONLINK has slots for the POINT object that the CONLINK object ends at and two slots low and high for the lower and upper bounds[4] on the distance separating the two points.

[3] I don't want to bring in the details of the underlying LISP system and its conventions for defining data types and creating and accessing slots in structures, so I will confine my attention to the deductive retrieval system as much as possible. For the curious, data structures like tokens and points are implemented as interned symbols with their slots accessible from their property lists. This eliminates a number of problems associated with unification and syntactic indexing that would otherwise arise had we chosen to use vectors, hunks, or some other more exotic (though possibly more efficient) means of implementing data structures.

[4] The data type of these bounds differs depending upon the implementation. In the implementation used in the examples in this dissertation, the bounds are either integers (data type FIXNUM) or elements of a small set of objects of data type SYMBOL. These symbols allow us to represent either negative or positive "unboundedness" (respectively *neg-inf* and *pos-inf*) and the notion of infinitesimally small in either the negative or positive direction (respectively *neg-tiny* and *pos-tiny*). The system employs a set of routines for adding and subtracting such objects in a consistent manner. The implementation

Objects of type POINT have a slot, constraints, which provides access to all CONLINKs that begin at that POINT object. Asserting (elt (distance pt1 pt2) 4 5) would result in the creation of two CONLINK objects. One would end at pt2 and be stored at pt1 under the constraints slot of that POINT. Its lower and upper bounds would be 4 and 5 respectively. The other CONLINK would end at pt1, be stored at pt2, and have lower and upper bounds of −5 and −4 respectively. CONLINK objects also have slots used by the system to guide search. There are two such slots both of data type FIXNUM; one is a general weighting slot and the second is a slot that records some indication of the time at which the CONLINK object was created (*i.e.*, processor time).

Every constraint has associated with it a ddnode. The propositional content of such a ddnode corresponds to the form of the associated assertion (*e.g.*, (elt (distance pt1 pt2) 3 4)). The label of this ddnode determines whether or not the corresponding directed edges in the time map (*i.e.*, CONLINKs) are traversible by the routines employed in propagating constraints and determining the "when" of time tokens. Each time you assert a constraint, the system creates a ddnode, just as it does when you assert a new time token. There is one difference, however, worth noting. The ddnode corresponding to (elt (distance pt1 pt2) 3 4) is not identified with the assertion (elt (distance pt1 pt2) 3 4). So, for example:

```
(answer-support (+ '(elt (distance pt1 pt2) 3 4)) code)
```

does not result in the ddnode created at the time (elt (distance pt1 pt2) 3 4) was asserted being added to the support of the current answer. Constraint ddnodes are not easily accessed by the user. This means that a particular ddnode corresponding to a constraint can't be incorporated into a justification using the techniques that I've described thus far. The reason is simple: it is seldom, if ever, necessary in practice to construct such justifications, and I wanted to avoid the considerable cost of indexing the (typically) large number of constraints involved in constructing time maps. If all you're worried about is that assertions occurring in *code* depend upon (elt (distance pt1 pt2) 3 4) being true, then there is a simple alternative. One can set up such a dependency using:

```
(for-first-answer (fetch '(elt (distance pt1 pt2) 3 4)) code)
```

however, the user has no control over which ddnodes corresponding to constraints are used in augmenting the current answer.

details are uninteresting and will be ignored.

Constraints establish a network of temporal connectivity used for determining relationships between the intervals associated with time tokens. For reasons that will hopefully become clear in the next section, the inferential connectivity among time tokens established in the data dependency network and the temporal connectivity between those same time tokens established in the time map can interact in ways that make it difficult to guarantee that the TRMS update algorithm will always terminate. In the next section, I'll develop the idea of temporal data dependence, describe the temporal reason maintenance system in some detail, and provide a criterion under which we can guarantee that the TRMS algorithm will terminate correctly.

4.3 Temporal reason maintenance

The previous chapter introduced a number of functions that the time map management system should support. The system was advertised as performing temporal reason maintenance. In order to understand what that might entail, I introduced the notion of an apparent contradiction and suggested that it was the responsibility of the system to resolve apparent contradictions where possible. The basic task of the TMM is the same as that for a conventional (static) reason maintenance system, namely, to keep track of the conditions for belief. In a temporalized data base, the conditions for belief are somewhat more complex than in the static case. A (temporal) condition for belief is generally something of the form: there exists a token asserting P that begins before a given interval and possibly persists throughout that interval. This sort of condition was referred to as a protection. Protections are, like Doyle's assumptions, nonmonotonic in that they can be undermined by the addition of new information. A protection can fail as a result of either adding or removing a constraint. It was shown that resolving apparent contradictions can result in protection failures. From the user's perspective the task of a temporal data dependency system is to detect when an assertion is no longer justified and assist the user in responding in an appropriate manner. The rest of this section is devoted to explaining how the system detects and resolves apparent contradictions and detects and responds to protection failures.

4.3.1 Apparent contradictions

Recall that two tokens were said to be apparently contradictory if they satisfied the following criterion:

```
(<- (apparent-contradiction ?tok1 ?tok2)
    (and (time-token ?p ?tok1)
         (time-token ?q ?tok2)
         (contradicts ?p ?q)
         (strict-elt (distance (begin ?tok1) (begin ?tok2)) ?low1 ?high1)
         (=< 0 ?low1)
         (strict-elt (distance (begin ?tok2) (end ?tok1)) ?low2 ?high2)
         (=< 0 ?high2)))
```

It is the responsiblity of the system to resolve such situations by constraining the earlier token to end before the later one. The hard part, however, is noticing that resolution is required in the first place. Adding new tokens alone does not result in apparent contradictions. It is the addition of new constraints that the system has to be alert to. Every additional constraint results in new paths through the constraint network. Some of these new paths may enable the system to deduce new apparent contradictions. Noticing this efficiently is rather tricky.

The apparent contradiction criterion can be broken down into three tests: (1) do the two tokens contradict one another? (2) does one token precede the other? and (3) assuming that one token does precede the other, could the earlier token possibly overlap the later token? The result of the first test can be established once and for all at the time new tokens are created. This is because time token schemata contain no variables and contradiction criteria do not depend upon properties of terms in schemata that change. The third test is not performed in the current implementation. If two tokens could overlap, then adding the appropriate constraint will eliminate the possibility. If the two tokens can't overlap, then adding a constraint that ensures this fact shouldn't hurt. The second test is used to determine whether or not the two tokens possibly overlap.

When a new token is asserted, the TMM finds all the tokens that contradict it (actually it only finds a limited subset of such tokens, but I'll get back to this shortly). For each pair of contradictory tokens `t1` and `t2` found in this way, the TMM creates a pair of what are called *clipping constraints*. For the tokens `t1` and `t2`, the two clipping constraints would be `(pt< (end t1) (begin t2))` and `(pt< (end t2) (begin t1))`. The purpose of the clipping constraints is to eliminate an apparent contradiction should one be detected. (When

Two situations resulting in apparent contradictions involving t1 and t2:

```
t1  P
 ||---------------------->
t2  (not P)
          ||---------------------->

t2  (not P)
 ||---------------------->
t1  P
          ||---------------------->
```

Data dependencies used for resolving apparent contradictions:

ddnode:	associated data type:	corresponding datum:
n1	TOKEN	(time-token P t1)
n2	TOKEN	(time-token (not P) t2)
n3	TCONDIT	(pt< (begin token1) (begin token2))
n4	TCONDIT	(pt< (begin token2) (begin token1))
n5	CONLINK (pair)	(pt< (end token1) (begin token2))
n6	CONLINK (pair)	(pt< (end token2) (begin token1))

n5 has the justification ({n1, n2, n3}{})
n6 has the justification ({n1, n2, n4}{})

Figure 4.2: Dependency relations for managing persistence clipping

the end of one token is constrained to precede the beginning of a second (contradictory) token the sescond token is said to *clip* the first.) To make this work, the TMM adds a special justification to each clipping constraint in accordance with the second of the three tests described above. The easiest way to understand this is by looking carefully at the structures and dependencies built by the TMM for the pair of contradictory tokens t1 and t2.

First, suppose that t1 and t2 have corresponding ddnodes n1 and n2. The system constructs two additional ddnodes n3 and n4 corresponding to the predications (pt< (begin t1) (begin t2)) and (pt< (begin t2) (begin t1)) respectively. These predications are referred to as *temporal conditions* and are associated with objects of data type TCONDIT. The system also constructs ddnodes n5 and n6 corresponding to the clipping constraints (pt< (end t1) (begin t2)) and (pt< (distance (end t2) (begin t1))). The only justification for n5 is ({n1, n2, n3}{}) and the only one for n6 is ({n1, n2, n4}{}). Figure 4.2 shows the two situations leading to apparent contradictions involving t1 and t2 along with the data dependencies set up to resolve them.

The propositional content of a ddnode associated with a constraint will have the form (elt (distance pt_1 pt_2) *low high*). The justification for this ddnode is constructed from the current answer at the time the constraint is added to the data base. The propositional content of a ddnode associated with a temporal condition also has the form (elt (distance pt_1 pt_2) *low high*), but in this case the justification for the ddnode will be provided by the system. The system tries to make sure that each ddnode corresponding to a temporal condition (such as n3 or n4) is IN just in case there is path through the network of constraints (CONLINKs) whose bounds satisfy the condition corresponding to propositional content of the ddnode. The TMM is said to manage the "virtual transitive closure" of a set of relations of which pt< is but one. We'll return to see how this is done after we see how the same sort of thing helps out in monitoring protections.

I mentioned that the TMM does not set up dependencies to monitor the possibility of contradictions involving all possible pairs of contradictory tokens. We're interested in having the system resolve any apparent contradiction that might occur, but we'd also like to avoid expending a great deal of time and storage worrying about all those contradictions that are "highly unlikely". If there are *n* tokens of type P and *n* tokens of type (not P), then there will be on the order of n^2 possible contradictions the system will have to set up dependencies in order to keep track of. Given that we allow both the addition and deletion of constraints, coming up with a strategy that is both complete and efficient presents some problems. How do you tell that two contradictory tokens will never overlap? The obvious answer is that you can't tell, but you can enlist the user's help to narrow the range of possibilities. The technique used in the current TMM places the responsibility on the user to establish a focus of attention or *kernel* that restricts attention to a limited subset of the set of all tokens in the time map. The kernel serves a function similar to that of a fast associative memory in modern computer architectures. If something is relevant to the current computation, it is assumed that it will be added to the kernel. Likewise, if something is no longer of interest, it should be swapped out. Operations on tokens within the kernel can be performed efficiently since there is less data to search through. The kernel technique is especially useful in planning applications that interleave planning and execution (see [Dean 83]). In these applications, the process of swapping tokens in and out of the kernel can easily be automated. However, as we begin to experiment with applications involving time maps of increasing size and inferential requirements of increasing complexity, it has become apparent that there are a lot of issues that still need to be addressed.

4.3.2 Protections

The notion of a protection is strongly connected with planning [Sussman 75]. The idea of monitoring a protection came from trying to make sure that a prerequisite task served its purpose, where purpose was construed narrowly to mean making a fact true over an interval spanning the main task served by the prerequisite. In the time map we have stretched this notion to mean simply making sure that some proposition is believed to be true throughout an interval. Different sorts of protections demand different techniques for handling them.

Suppose that you've planned to install an attachment in a lathe to facilitate some manufacturing process. Now, due to a new ordering constraint, it turns out that another task will remove the attachment before it is put to its intended purpose. In this case, we would use a protection to monitor whether or not a task fulfils its intended purpose. It would be inappropriate to initiate a new task for installing the attachment. On the other hand, suppose that you chose a plan on the basis of a conveyor being turned on at 8:00 AM. The plan gets scheduled for 3:00 PM, but around noon someone shuts down the conveyor briefly in order to make some minor repairs. The conveyor was only down for a few minutes and the time map even shows that it will be running at 3:00, but a different time token spans the afternoon hours. In this case it would be convenient if the time map machinery could transfer dependence to the later time token and never bother the user about a potential failure.

The TMM has three different types of protections. The first is called a simple protection, and since the other two types are just augmented simple protections I will describe simple protections in some detail before turning to the other two types.

All protections are implemented as ddnodes associated with an assertion of the form (passume P pt1 pt2) (passume stands for protection assumption).

```
(define-predicate (passume PROP POINT POINT))
```

Every protection has one or more justifications, each of which refers to a specific time token. A simple protection has only one such justification. The justification is composed of three additional ddnodes: (1) n1 - a time token, call it t1, asserting P (2) n2 - the node associated with the temporal condition (pt=< (begin t1) pt1) and (3) n3 - the node associated with the temporal condition (pt< (end t1) pt2). The data dependencies set up in a simple example of controlled forward chaining are shown in Figure 4.3. The justification for the ddnode *associated with (passume P pt1 pt2) is ({n1,n2}{n3})* and corresponds to an instantiation of the *protection criterion*:

```
 (<- (passume ?p ?pt1 ?pt2)
     (and (time-token ?p ?tt)
          (pt=< (begin ?tt) ?pt1)
          (consistent (pt=< ?pt2 (end ?tt)))))
```

Simple protections will work quite nicely for handling the example dealing with lathe attachments. You just set up a protection and have it justify something that will execute an appropriate signal function in the event that the protection fails. In fact simple protections will work for the other example involving the conveyor if the user doesn't mind being troubled with false alarms. Let's consider this in a little more detail.

Suppose that assembly-plan34 scheduled for the afternoon depends upon conveyor2 running throughout the interval associated with the plan. You might set this up as:

```
(for-first-answer
 (fetch '(tt (begin assembly-plan34) (end assembly-plan34)
          (operational-status conveyor2 running)))
 (add '(suitable-plan assembly-plan34)))
```

The following code fragment:

```
(for-first-answer (fetch '(tt pt1 pt2 P))
     (add '(and (time-token Q t2) (pt= pt2 (begin t2)))))
```

given the time map:

```
t1  P
 ||------------------------>
    pt1          pt2
     |-------------|
```

results in the augmented time map:

```
t1  P
 ||------------------------>
    pt1          pt2
     |-------------|
t2  Q
                  ||------------------------>
```

and the following data dependencies for monitoring protections:

ddnode:	associated data type:	corresponding datum:
n1	TOKEN	(time-token P t1)
n2	TCONDIT	(pt=< (begin token3) pt1)
n3	TCONDIT	(pt< (end token3) pt2)
n4	PROP	(passume P pt1 pt2)
n5	TOKEN	(time-token Q t2)

n4 has the justification ({n1, n2}{n3})
n5 has the justification ({n4}{})

Figure 4.3: Dependency relations for handling protections

The query will succeed and (suitable-plan assembly-plan34) will be added to the data base justified by a protection which in turn is justified by the persistence asserting that conveyor2 is running since early this morning.

You want to be alerted if anything happens to this plan, so you have a change-driven interrupt set up to react to possible plan failures.

```
(if-erased '(suitable-plan ?plan)
      (some-hairy-debugging-routine ?plan))
```

Now, suppose that a task is added that shuts down the conveyor (thereby clipping the persistence which asserted that conveyor2 is running in the morining) and starts it up a few minutes later after a simple repair (thereby adding a new persistence asserting that conveyor2 is running at least through the afternoon). Clipping the earlier persistence would cause (suitable-plan assembly-plan34) to become OUT, and the interrupt described above would be triggered. The function some-hairy-debugging-routine might try to reestablish (suitable-plan assembly-plan34) by executing the for-first-answer code a second time. In this case, the query would succeed and a new protection justifying (suitable-plan assembly-plan34) would be constructed using the persistence asserting the conveyor is running in the afternoon.

It would be nice in this sort of situation if we could define a special sort of protection that would would try to reestablish itself if it ever became OUT. By reestablish I mean that the protection ddnode would have a signal function that would fire whenever the ddnode toggled from IN to OUT. This signal function would try to find another token satisfying the protection criteria, and if successful, it would create a new justification and install it in the protection ddnode. This is, in fact, exactly what a type II protection is. The associated signal function has a priority level which ensures that the user will not be bothered by false alarms[5].

Type II protection will work in some situations, but will fail in others. Suppose that a type II protection fails, and its associated ddnode becomes OUT. At this point the signal function would try to reestablish the protection. Let's suppose, however, that there is no token that satisfies the protection criteria. The protection stays OUT. Now, suppose that you remove a constraint that would allow a token to persist long enough to satisfy the

[5] User change-driven interrupts have a priority level that guarantees that all TMM signal functions are run first. Changes that occur during TRMS update and are subsequently nullified by other TMM signal functions are never noticed by the user.

protection criteria. Since there's no way to notice or anticipate this change, the protection won't be updated, and it will remain OUT despite the fact that there is a token which would satisfy it. One possible remedy for this would be to set up justifications for every time token that matches the fact type to be protected. This is called a type III protection. The overhead for a type III protection is potentially exorbitant. In most cases a simple protection is adequate and the greater generality of type II and III protections not worth the additional expense.

The time map uses simple protections by default. It is quite easy to change the default to type II or type III protections if desired.

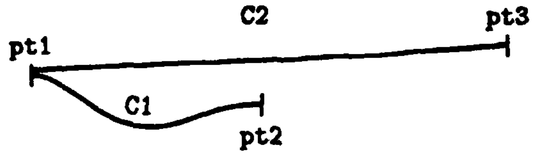
Notice that all three protection types depend upon the ability to keep track of the validity of temporal conditions such as (pt< pt1 pt2) and (pt=< pt1 pt2). In the next section we'll see how to implement the "virtual transitive closure" of temporal conditions involving pairs of points.

4.3.3 Keeping track of temporal conditions

The objective of this subsection is to describe a method for maintaining the following invariant: each ddnode corresponding to a temporal condition (*e.g.*, (pt< pt1 pt2)) is IN just in case the constraints in the time map warrant it. I'll begin by being more precise concerning what I mean by "the constraints in the time map warranting a temporal condition". Let's assume that all of the conditions that we're interested in can be defined in terms of the predicate elt described in Chapter 3.

The temporal condition (pt< pt1 pt2) is warranted in a given time map just in case there exists a path from pt1 to pt2 (through the time map) consisting of points and directed edges such that the sum of the lower bounds associated with the directed edges is greater than 0. If we just wanted to find out if (pt< pt1 pt2) was true in a given time map this would be simple. But for the TMM to work properly we want to be able to guarantee that the status (IN or OUT) of the ddnode corresponding to (pt< pt1 pt2) always accurately reflects the status of the constraints in the time map. Almost any change to the dependency network that affects a constraint can have a bearing on whether or not a ddnode corresponding to a temporal conditions should be IN or OUT.

We would like the following to occur: whenever the ddnode corresponding to a constraint becomes IN, if that constraint can participate in a path that satisfies a temporal condition,

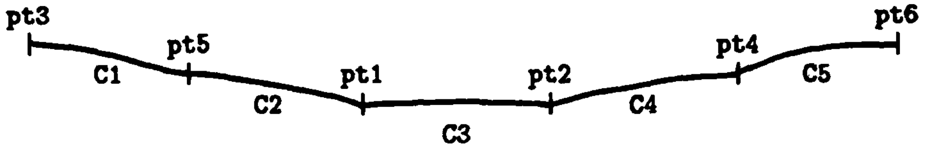


```
ddnode:                                          justifications:
 n1 --> C1 = (elt (distance pt1 pt2) 1 3)           premise
 n2 --> C2 = (elt (distance pt1 pt3) 4 5)           premise
 n3 --> (pt< pt2  pt3)                              (({n1, n2}{}))
```

Figure 4.4: Updating a temporal condition

then that fact is noticed and the system sets up a justification for the ddnode corresponding to that temporal condition. The justification is constructed from the ddnodes for the constraints used in the path. A simple example should make this clearer. Figure 4.4 illustrates a time map with three points, two constraints, and one temporal condition (pt< pt2 pt3). The path from pt2 to pt3 using constraints C1 and C2 has a lower bound of 1 and an upper bound of 4. Since the lower bound is greater than zero, this means that (pt< pt2 pt3) is satisfied. The justification ({n1,n2}{}) is added to n3, the RMS updates the data dependency network, and n3 becomes IN. If either of the two constraints ever becomes OUT, then the ddnode corresponding to (pt< pt2 pt3) will become OUT as well (assuming that it doesn't have another valid justification).

Whenever a new constraint is added (or, in general, whenever a new constraint becomes IN) the possible repercussions of that constraint have to be propagated throughout the network. The propagation must occur so that, among other things, new derivations for temporal conditions are noticed and appropriate justifications installed in the corresponding ddnodes. In the time map, propagation occurs as heuristic search. Noticing that a path satisfies a temporal condition is handled using objects of data type TCONDIT (introduced in the previous subsection) that are installed on objects of data type POINT in the time map. Each TCONDIT consists of a ddnode and a pair of functions. The propositional content of the ddnode corresponds to a temporal condition defined in terms of elt (*e.g.*, (pt< pt1 pt2)). The first function, called the *satfun*, is used to determine if the temporal condition is satisfied by a path through the constraint network. Such a path determines an upper and lower bound on the distance separating the two points referred to in the temporal condition. Recall that the time map is just a graph with POINTs for vertices and CONLINKs



```
Constraints:
  C1 = (elt (distance pt3 pt5) 4 5)
  C2 = (elt (distance pt5 pt1) 1 2)
  C3 = (elt (distance pt1 pt2) 1 1)
  C4 = (elt (distance pt2 pt4) 1 2)
  C5 = (elt (distance pt4 pt6) 3 4)
```

Points:	TMPATHs:	Temporal conditions:
pt1	{<pt1 pt1 {} 0 0>}	{}
pt2	{<pt2 pt2 {} 0 0>}	{}
pt3	{<pt1 pt3 {C1 C2} -7 -5>}	{(pt< pt3 pt4)}
pt4	{<pt2 pt4 {C4} 1 2>}	{(pt< pt3 pt4)}
pt5	{<pt1 pt5 {C2} -2 -1>}	{}
pt6	{}	{}

Figure 4.5: Updating temporal conditions during constraint propagation

for directed edges. A path is just a set of CONLINKs, and the lower (upper) bound on the distance estimate of the path is just the sum of the lower (upper) bounds of the CONLINKs. The satfun for a TCONDIT corresponding to the relation (pt< pt1 pt2) would return true if applied to a path from pt1 to pt2 with lower bound greater than 0 and false otherwise. The other function for a TCONDIT, the *assimfun*, is executed just in case the satfun returns true. The assimfun serves to install new justifications in the TCONDIT's ddnode. The new justification consists of all of the ddnodes associated with the constraints in the path (*e.g.*, in Figure 4.4 the path consists of the constraints C1 and C2 with corresponding ddnodes n1 and n2. The assimfun would install the justification ({n1, n2}{}) in n3).

In the course of propagating constraints, the ddnodes corresponding to temporal conditions are brought up to date. The actual algorithm used in the time map is rather hairy. It incorporates a number of optimizations and special techniques in order to deal efficiently with contexts [McDermott 83] and branching time [Dean 85]. The basic idea, however, is pretty straightforward. Given a constraint, you want to find each path traversing the constraint that satisfies the satfun for some TCONDIT and then use the assimfun for that

TCONDIT to install a suitable justification to bring the corresponding temporal condition up to date. Each time a constraint becomes IN, the system has to propagate that constraint. The algorithm proceeds by extending paths out from either end of the constraint in breadth-first order. The algorithm keeps temporary notes (objects of data type TMPATH) on each point visited. These notes are used to store information about the best paths found so far. Every time you extend a path, you see if you've already been there, and if so, update the note stored there. If you've never been there before, or you've found a better path, then you want to check to see if the new or improved path can be used to update any TCONDITs. An example should make this clear.

Figure 4.5 shows a simple time map involving six points and five constraints. In this figure, objects of data type TMPATH are notated *<starting-point final-point list-of-constraints-in-path lower-bound upper-bound>*. Suppose that the ddnode corresponding to the constraint C3 between pt1 and pt2 has just become IN. Suppose further that the update algorithm has already extended paths to pt5 and pt4, and it has just found a new path from pt1 to pt3. In checking to see if there are any TCONDITs relating pt3 to some other point, it finds the TCONDIT (pt< pt3 pt4) relating pt3 to pt4. To see if it can satisfy the temporal condition, it examines pt4 for a path from pt2. In this case, it finds the required path. Using the path from pt2 to pt4, the path from pt1 to pt3, and the constraint C1, it constructs a new path from pt3 to pt4. The lower bound of this new path is 7, which satisfies the TCONDIT's satfun. To complete the operation, the assimfun is used to construct a new justification for the ddnode corresponding to (pt< pt3 pt4) from the ddnodes associated with the constraints (C1, C2, and C3) in the composite path.

Each TMPATH has a score computed using the weighting information on the CONLINKs traversed in the path. The scoring function in the current implementation causes the search to be conducted in an essentially breadth-first manner, though it does tend to favor certain paths slightly. (I'll talk a little more about scoring functions when we get to the section on caching (Section 4.4.3).)

The above description of the propagation algorithm should be sufficient for most readers. The algorithm employs standard graph searching techniques with some additions to handle updating TCONDITs. For those wishing a slightly more detailed exposition, I will now present the algorithm a bit more carefully.

An object of data type CONLINK consists of a begin and end POINT, and an upper and lower bound on the distance separating the two points. In addition, each CONLINK has a pointer to the ddnode associated with its corresponding constraint. A CONLINK is traversible if and only if this ddnode is IN. An object of data type TMPATH records an estimate of the distance from one POINT in the time map, its beginning, to a second POINT, its end. A TMPATH is stored on a list referenced from the POINT corresponding to the end of the TMPATH. The distance estimate for a given TMPATH is actually computed from two (possibly different) paths though the time map: one for a lower bound and one for an upper bound. An upper (lower) bound of a path is equal to the sum of the upper (lower) bounds of the individual CONLINKs traversed in the path. In the time map, the justification for believing a distance estimate is as important as the estimate itself. The *path-support* set for the upper or lower bound of a path consists of all the ddnodes associated with CONLINKs traversed in the corresponding path. The justification constructed for a distance estimate has a set of in-justifiers equal to the union of the path-support sets for the bounds used in the distance estimate. Such a justification has no out-justifiers. A TMPATH has slots for the upper and lower bounds of its associated distance estimate and slots for the corresponding upper and lower path-support sets.

Now I want to define the function propagate-constraint. This function takes six arguments. The first five correspond to the ddnode (dn), the begin and end POINT (pt1 and pt2), and the upper and lower bounds (low and high) of the constraint to be propagated. The sixth argument is an indication of the amount of time to be spent in propagating this constraint. It can be either a fixed amount of CPU time or a depth cutoff (in the description below I refer to an allotment of CPU time). The algorithm keeps a queue of TMPATHs to extend. This queue is generally sorted according to some criterion of goodness (*e.g.*, path length or the sum of the weights on the CONLINKs in the path). In this case, the "most promising" TMPATH is always the first on the queue. The function propagate-constraint is defined as follows:

```
propagate-constraint(dn pt1 pt2 low high alloc-cpu)
 1. Shadow all constraints linking pt1 to pt2 by setting the labels of
    their corresponding ddnodes to OUT - set aside each old label and
    ddnode so they can be restored later (the reason for this is that
    we're only looking for paths that include the constraint being
    propagated).
 2. Set the queue to {}.
 3. Create a TMPATH with beginning POINT pt1 and end POINT pt2. Set
    the lower bound to low, the upper bound to high, and the lower and
    upper path-support to {dn} - put the new TMPATH on the queue.
 4. Create a TMPATH with beginning and end POINT pt1.    Set the lower
```

```
   and upper bounds to 0  and set the lower and upper path-support to
   {}  - put the new TMPATH on the queue.
5. Choose the "most promising" TMPATH on the queue in order to extend
   the search - call this chosen path the initial-segment.
6. For each CONLINK cl starting at the end POINT of the initial-segment:
   a. Call the end POINT of cl tc-begin.
   b.  Create a new TMPATH called the extended-path with the same
     beginning POINT as the initial-segment and end POINT tc-begin
     i.  set the lower (upper) bound equal to the sum of the lower
        (upper) bounds of cl and the initial-segment.
     ii. set the lower (upper) path-support equal to the lower (upper)
        path-support of initial-segment augmented with the ddnode
        associated with cl
   b. If there is no existing TMPATH at tc-begin that has
         the same beginning POINT as the extended-path,
      then place the extended-path on the list of TMPATHs of
           tc-begin and put the extended-path on the queue
      else if either the lower or upper bounds of the extended-path
               are better than those of the the existing path,
            then update the existing path (and its corresponding
                 path-support) and place the existing path on the queue.
   c. If either there was no existing TMPATH at tc-begin with
                the same beginning POINT as the extended path
             or the extended-path was better than the existing path
                in either its lower or upper bounds
       then for each TCONDIT tc at tc-begin:
            i. find the POINT corresponding to the other end of
               the TCONDIT and call it tc-end
            ii. if there is a TMPATH at tc-end that begins at the
                   POINT opposite the one extended-path begins at
                   (TMPATHs can only begin at one of pt1 or pt2),
                 then I. form a new path called the resultant-path
                        by combining this opposing path with the
                        extended-path and the constraint being
                        propagated.
                      II. if the satfun of tc applied to the
                             resultant-path returns T
                           then execute the assimfun of tc
                                with the same resultant-path
7. If there's still CPU time left and the queue is not empty
     then go to step 4
     else  a. restore the old labels to the DDNODEs shadowed in (1)
           b. clean up the visited POINTs by removing all TMPATHs
```

There are a number of complications involved in the propagation of constraints that are worth mentioning briefly. When a new justification is installed, the RMS is called and the status of a temporal condition may change. This, in turn, can result in other constraints becoming IN (*e.g.*, clipping constraints) that need to be propagated. The signal functions associated with constraints and the priority queue mechanism ensure that if a constraint becomes IN, it will eventually be propagated. Other complications arise from attempting various optimizations. The TMM does not install every possible justification for temporal conditions. The main reason is to save work. In order to ensure completeness, TCONDIT signal functions are designed to try to find a path satisfying the TCONDIT satfun if the TCONDIT ddnode ever becomes OUT. In practice, this technique and other similar dependency-mediated bookkeeping methods make a noticeable difference in the performance of the TMM. In Section 4.4.3 we'll look at some additional techniques for caching point-to-point distances that make use of special TCONDITs.

Using the constraint propagation scheme outlined above, and allowing the algorithm to run until no path extensions achieve better bounds, guarantee maintenance of the invariant described at the beginning of this subsection. Of course this sort of exhaustive search is prohibitively expensive. The complexity of the algorithm depends upon the number of constraints. Assuming a reasonable number of constraints, say n^2 where n is the total number of points, the algorithm will take time proportional to the cube of n. If n is large, as is expected in many applications, and constraints are frequently changed, then this sort of overhead cannot be absorbed. In most cases, however, you can get by with much less than exhaustive search. The easiest approach is to put an absolute limit on the search in terms of either CPU seconds or length of longest path considered. In most applications, the length of search paths required to catch all critical temporal conditions is bounded by a small integer. For a given application this bound can be determined with a little experimentation. It might also be convenient in some instances to vary the bound to suit the type of problem being worked on or the time required for a solution. Adding a constraint (or having a constraint become IN) can be handled in constant time where the constant depends upon the sort of temporal connectivity expected for the application at hand.

4.3.4 The TRMS update algorithm

We can now describe the overall algorithm for performing temporal reason maintenance in terms of a set of invariants to be maintained. The invariants are:

1. all TCONDITs are as IN as the current set of constraints warrant

2. all apparent contradictions are resolved

Priority:	**Data dependency nodes and associated signal functions:**
100	ddnodes associated with constraints (CONLINKs) - the signal function instigates constraint propagation whenever the ddnode becomes IN.
200	ddnodes associated with temporal conditions (TCONDITs) - the signal function tries to find a new path connecting the two points whenever the ddnode becomes OUT.
300	ddnodes associated with TCONDITs responsible for detecting apparent contradictions - the signal function propagates a constraint which serves to clip the persistence of the earlier occurring time token whenever the ddnode becomes IN.
400	ddnodes assocated with type II protections - the signal function tries to find an alternative time token satisfying the protection criteria whenever the ddnode becomes OUT.
500-900	ddnodes associated with if-erased demons and other sorts of change-driven interrupts - priority levels for user invariants

Figure 4.6: TRMS invariants and signal functions responsible for maintaining them

3. all protections (type II in particular) are as IN as possible

The algorithm for maintaining these invariants is implemented using the signal functions and priority levels (where the lower the number the higher the priority) shown in Figure 4.6.

Once you understand how signal functions and priority queues work in the static RMS (see Section 3.2.5), the TRMS algorithm can be completely specified in terms of the dependency structures built by the TMM and signal functions attached to various ddnodes. Previous sections have sought to describe these signal functions and dependency structures, but I doubt very much that most readers will have a clear understanding of the algorithm, or feel assured that it performs as advertised. The actual flow of control in such an algorithm is extremely difficult to follow. The important changes to the time map (those requiring significant reorganization and relabeling of ddnodes) result from adding or removing constraints and erasing time-tokens[6]. When constraint ddnodes become IN, their signal functions propagate their corresponding point-to-point distance estimates. This propagation can result in the update of TCONDIT ddnodes which participate in the justifications for protections and clipping constraints. The former can directly affect the status of ddnodes corresponding to time tokens. Either update can result in a change in the status of ddnodes corresponding to constraints used to resolve apparent contradictions. If one of these ddnodes becomes IN, then propagation occurs and the cycle repeats. In the next (sub)section I'll show that (under certain assumptions) this cycle of activity is guaranteed to terminate in a state that satisfies one interpretation of correctness for time maps.

4.3.5 Correctness of the temporal reason maintenance algorithm

The top level invariant maintained by the temporal reason maintenance system can be expressed as follows:

> Each token, T1, can be shown to clip each contradictory token, T2, just in case T1 and T2 are IN and there exists a path through the time map consisting of CONLINKs with IN ddnodes such that the beginning of T2 can be shown to precede the beginning of T1.

Of course, tokens are typically justified by protections that are IN or OUT, depending upon the status of other tokens and the extent to which these (protecting) tokens are clipped. This gives rise to a circularity that could potentially cause problems. Figure 4.7 shows how changes instigated by the user are translated into calls to the static RMS and execution of signal functions which, in turn, give rise to further calls to the static RMS. The circularity is plainly depicted in the Figure 4.7. The user modifies the time map by adding new tokens and constraints in the course of controlled forward chaining. It is the addition of constraints which is of real interest. Modification to the status of a CONLINK ddnode can cause constraint propagation which, in turn, causes modification to the status of other CONLINK ddnodes. If one is not careful, this can go on indefinitely. Ensuring that it does not, will require the cooperation of the user. In order to construct a proof of correctness, we will have to establish some conventions for using the TMM.

[6] Adding a time token is not nearly as important as constraining a time token. An unconstrained time token has no appreciable effect on the time map.

```
   external modifications to the constraint set
___||_________________________________________________________________
|  ||                                                                |
|  |/============<<==============<<============\\======<<========\\  |
|  \/                                          ||  ____________  ||  |
| changes in the status of CONLINK ddnodes     || | changes in | ||  |
|  ||                                          || | the status | ||  |
|  ||  IN => contraint propagation             /\ | of clipping| ||  |
|  \/                                          || | constraint | ||  |
| changes in the status of TCONDIT ddnodes     || |__ddnodes___| /\  |
|  ||                                          ||                ||  |
|  |\============>>==============>>============//                ||  |
|  ||                                                            ||  |
|  ||  OUT => attempt to reestablish temporal conditions         ||  |
|  ||                                                            ||  |
|  |/=================<<======================<<=================\|  |
|  \/                                                            ||  |
| changes in the status of protection ddnodes                    ||  |
|  ||                                                            ||  |
|  ||  OUT => attempt to reestablish type II protections         /\  |
|  \/                                                            ||  |
| changes in the status of TOKEN ddnodes                         ||  |
|  ||                                                            ||  |
|  \\=================>>======================>>=================//  |
|____________________________________________________________________|
```

Figure 4.7: Flow of control in temporal reason maintenance

I'll begin by describing a general criterion that should serve as the basis for constructing working sets of conventions that guarantee correctness. The situation that we're trying to avoid occurs when the action of clipping a persistence results in a chain of further actions that are ultimately responsible for undoing that clipping. The criterion is captured a bit more concretely in the following two rules:

1. clipping which results from resolving an apparent contradiction between two tokens cannot be the cause of a situation in which one of the two tokens becomes OUT.
2. clipping which results from resolving an apparent contradiction between two tokens cannot be the cause of a situation in which one of the CONLINK ddnodes involved in establishing the apparent contradiction criterion (*i.e.*, one of the CONLINKs in the path satisfying the TCONDIT used to determine that one token precedes a second) becomes OUT.

The above rules do not provide a satisfactory foundation upon which to build a proof of correctness. Neither do they provide a great deal of guidance in building systems that employ the TMM and depend upon it performing correctly. In the following, I will present 5 specific conventions for managing temporal data bases that satisfy the rules above and provide a general framework that is provably correct. These rules may at first appear rather restrictive, but I hope to show that they impose reasonable restrictions and do not undermine the basic functionality described in the previous chapters. These conventions, once their intent is understood, can be considerably relaxed if one is careful. In addition they make the proof of correctness a good deal less complicated. The TMM was used for over a year before anyone tried to carefully prove anything about its behavior. In that time, the conventions to be described were frequently abused with no noticeable detrimental effects. Where possible, I will try to motivate each convention with examples showing why violating the convention can get you into trouble, and why the convention will, in many cases, keep the user from getting into situations which on the face of it may appear natural but on closer inspection point out some underlying conceptual misunderstanding.

```
(for-first-answer
      (fetch '(and (time-token E ?tok1)
                   (tt (begin ?tok) (end ?tok) P)
                   (tt (begin ?tok) (end ?tok) Q)))
      (add '(and (time-token R ?tok2)
                 (elt (distance (end ?tok1) (begin ?tok2))
                      *pos-tiny* *pos-tiny*))))
```

Query for establishing dependency

```
Frame of reference: (begin token3)  Scale: 1.0

token1 P
 ||---------------------------------------------------------->
token2 Q
    ||------------------------------------------------------->
token3 E
       |---------|
token4 R
                 ||------------------------------------------>
```

Resulting time map

Figure 4.8: Restricted controlled forward chaining

Rule 1

Temporal data dependencies constructed in controlled forward chaining must conform to the following: All persistence tokens which depend upon protections are constrained to follow (by at least some infinitesimal amount) the intervals associated with their justifying protections.

Figure 4.8 illustrates how temporal data dependencies are generated which conform to Rule 1.

What we really want to say here is that a time token with schema P cannot depend upon (even indirectly) a time token with schema R where P and R contradict one another and the token with schema R is protected longer than the beginning of the token with schema P. Simply put, a persistence cannot undermine its own reason for being. Figure 4.9 illustrates

```
(for-first-answer
      (fetch '(tt pt1 pt2 P))
      (add '(and (time-token (not P) ?tok2)
                 (elt (distance pt1 (begin ?tok2)) 0 *pos-inf*)
                 (elt (distance (end ?tok2) pt2) 0 *pos-inf*))))

Frame of reference: pt1  Scale: 1.0

token1 P
 ||~~~~~~~~~~~|???????????????????????????????????????????????>
    pt1         pt2
    |-----------|
token2 (not P)
    |~~~~~~~~~~|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
```

Figure 4.9: Situation in which the TRMS will not terminate

a blatant disregard of this rule.

The loop in this case is quite simple. Initially P is true throughout the interval from pt1 to pt2 so token2 is IN. But if token2 is IN, then it is in apparent contradiction with token1, so the ddnode associated with the clipping constraint becomes IN and is propagated. But this should result in noticing that the end of token1 precedes pt2, which says that P is no longer protected throughout the interval from pt1 to pt2, and so token2 is OUT. But if token2 is OUT, then so is the clipping constraint ddnode so the protection is back, IN therefore token2 is IN, and so on until something external terminates the program. The problem in such cases is that the data base is essentially contradictory. The cycle of deduction that leads to a contradiction is seldom so apparent as that shown in Figure 4.9. In general it requires a good bit of care on the part of the designer to see to it that such contradictions are not possible. The rule stated above ensures that no contradictions occur, but still allows for the case in which an event can terminate one of its antecedent conditions (*e.g.*, throwing the main circuit breaker in the basement may depend upon the lights being on so I can see the breaker box; turning the main breaker off will result, however, in the light being off and my being in the dark).

It may seem somewhat clumsy to require that a persistence begin after (but not coincident with) the end of the interval(s) over which its dependent protections must span. This requirement is necessary due to the fact that persistence clipping constraints force the earlier token to end before (but not coincident with) the later token. Since protections demand that the associated token not end before the end of the protected interval it is necessary that persistences begin after the end of the protected interval. Essentially, this means that all tokens are closed intervals. Because of the way persistence clipping and protections are handled, the following code fragment will not terminate:

```
  (for-first-answer
     (fetch '(tt pt1 pt2 (on light34)))
     (add '(and (time-token (off light34) ?tok)
                (elt (distance pt1 (begin ?tok)) 0 0))))
```

However, if we're willing to look at persistence boundaries in a slightly different manner, we can set things up so the above code fragment will terminate. All we have to do is change the persistence clipping constraint so its lower bound is 0 instead of *pos-tiny*. This allows that two contradictory tokens can share boundary points. If you adopt a consistent interpretation in which persistences are considered closed on the left and open on the right, then this will not cause problems. It is important, however, to be clear about what is happening at boundary points. Rule 1 implies a consistent strategy, but other less restrictive ones are possible.

Rule 2

Having initiated the TRMS update algorithm by changing the justification for a constraint or time token, the only additional constraints to change status are those added by the system in resolving apparent contradictions.

This rule is a bit harder to swallow than the previous one, simply because it forces a (partial) separation between temporal connectivity and inferential connectivity. From an aesthetic point of view, it would be nice if we could somehow integrate the two. Ideally, all constraints should be dependent upon the tokens whose points they constrain. If a token became OUT for one reason or another, then its associated constraints would evaporate as well, leaving the data base free of "phantom" constraints (*i.e.*, constraints relating points belonging to tokens which are OUT). Figure 4.10 illustrates such a situation. Suppose that I had been planning for some time to tie one on with a few friends, but, things being as they are, we'd never pinned the date down too carefully. Then we hear of the fuse being lit (this is all weakly metaphorical) and we mutually decide to have our binge before the bomb. Now assuming that we have some idea when the bomb will explode (for instance we know that it has a fuse just as long as the patience of the most belligerent SAC commander), it makes

```
Frame of reference: (begin light41)  Scale: 1.0

lighet41 (light (fuse-of bomb32))
            |--|
burning6 (burning (fuse-of bomb32))
                ||------------------|
explosion14 (explode bomb32)
                                    |---|
binge67 (ingest !<(fluid-type alcohol) (quantity excessive)> self)
                       |---|--------|
```

Figure 4.10: Tying one on while there's still time

sense to say that we'll schedule our binge before the explosion. It also makes sense that if the threat of explosion disappears, then we should no longer feel constrained to hasten our dissipation. That is to say, the constraints that refer to a token which is OUT should be OUT as well.

Unfortunately, there are other times when it seems quite natural to constrain tokens representing real events relative to those which are not believed to occur. Figure 4.11 shows a simple example. In this case, you plan to extinguish the lighted fuse in order to prevent the explosion. The fuse quenching task is constrained to precede the predicted explosion event. Obviously in this case, if the constraints evaporate we may not execute the quenching task in time[7]. Now, you may say that it was wrong to have constrained the quenching task to come before the explosion in the first place. Instead it should've been constrained to follow the fuse lighting event by some small period of time. This will work out fine in the two examples (and the TRMS algorithm will terminate correctly), but, and the reader will have to take my word for this, there are other situations where this is not enough. Neither approach will do precisely the right thing in all cases. It is the user's responsibility to separate inferential connectivity and temporal connectivity so as to avoid situations leading to nontermination. In practice this in not at all dificult. "Phantom" constraints still have to be dealt with on occasion, but this has never presented much of a problem. In those situations that require being informed that an event is no longer predicted to occur and therefore should not constrain other events, it is simple enough to set up dependencies to monitor this fact and respond to it in an appropriate manner.

[7] It is also the case that in this situation the temporal data dependency system won't terminate.

```
Frame of reference: (begin light41)  Scale: 1.0

lighet41 (light (fuse-of bomb32))
            |--|
burning6 (burning (fuse-of bomb32))
               ||~~~~~~~~~~~~|
explosion14 (explode bomb32)
                                    |---|
extinguish35 (extinguish (fuse-of bomb32))
                         |~~~~~~~|--|
quenched17 (not (burning (fuse-of bomb32)))
                            ||~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
```

Figure 4.11: Circularities

Keeping temporal connectivity separate also makes fault annotation (for dependency directed debugging in planning) simpler. An example should illustrate how. Suppose that the robot has a task to transfer an object from one location to another, and that one way of doing this is to place the object upon a working conveyor. That is to say, the plan for transferring an object using the conveyor depends upon the conveyor being operational throughout the period of transfer. Let's also suppose that the robot has the task of performing some routine service work on the conveyor which will require shutting down the conveyor for an indeterminate period of time. Figure 4.12 shows the relevant portions of the situations (the transfer task and the routine service task are not constrained relative to one another and hence are shown from different frames of reference). One subtask of the transfer task, the subtask involved with placing the object on the conveyor, is shown. This subtask is constrained to occur during the transfer task. The question we're concerned with here is what is the justification for those constraints. Suppose for a moment that the constraints depend upon the protection that it be true throughout the tranfer task interval that the conveyor be operational. Let's see what problems this raises. Suppose that the routine service task is constrained to occur after the beginning of the transfer task and before the beginning of the task to place the object on the conveyor. Figure 4.13 shows such a situation, but the dependency network in this case is not stable. When the constraints on the routine service task are added, the system can clip the persistence corresponding to the conveyor being operational; this in turn will cause a protection violation, which will "erase" (*i.e.*, cause to become OUT) the constraints that forced the placement task to oc-

```
Frame of reference: (begin transfer14)  Scale: 0.3
                        -
running17 (operational-status conveyor34 in-service)
<~~~~~~~~||~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
transfer14 (transfer object41 loc1 loc2)
                        |--------------------|
place56 (place object41 conveyor1)
                        |~~~~~~~~~~~~~|------|

Frame of reference: (begin routine-service43)  Scale: 0.3
                        -
routine-service43 (routine-service conveyor34)
                        |----------|
shutdown44 (shutdown conveyor34)
                        |~~~~~|----|
down8 (operational-status conveyor34 out-of-service)
                             ||~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
```

Figure 4.12: Potential problems with termination

cur during the transfer task. But these constraints were used in clipping the persistence, thereby violating the protection in the first place, and so the protection becomes IN, and the constraints become IN, and the whole process repeats. In this simple example, it seems that we should be able to detect the problem and tell the user to retract certain constraints but in general the repeating cycles could be arbitrarily complex. Determining when you've been in the same state twice can be quite expensive.

But, of course, detecting the problem and simply stating that the data base cannot be reconciled with the newly added constraint is not all that informative. If the constraints in this case could've been depended upon to remain stable, the system could've told the user that the constraint on the routine service task endangered the success of the transfer task. This is the sort of information that a planner should be prepared to deal with, and a temporal management system prepared to supply.

If you are going to allow the calling program to make arbitrary ordering decisions, then you're going to have to lay down some rules about what sort of dependency networks can be constructed with impunity. Keeping temporal connectivity and inferential connectivity separate is in some cases an extreme move. The system doesn't demand it (the user has full control over the current answer and the alternate answer), but it is recommended in

```
Frame of reference: (begin transfer14)  Scale: 0.3

running17 (operational-status conveyor34 in-service)
<‾‾‾‾‾‾‾‾||‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾|
transfer14 (transfer object41 loc1 loc2)
                        |--------------------|
routine-service43 (routine-service conveyor34)
                        |‾|----------|
place56 (place object41 conveyor1)
                                   |‾|-------|
shutdown44 (shutdown conveyor34)
                        |‾‾‾‾‾‾‾|----|
down8 (operational-status conveyor34 out-of-service)
                              ||‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾>
```

Figure 4.13: Unstable time map

many applications including planning. Chapter 5 will describe a regimen that maintains the separation in a fairly natural way suitable for robot planning and problem solving.

Rule 3

> The only constraints that refer to the end of a persistence are those added by the system in resolving apparent contradictions.

This rule essentially says that the end points of persistences must "float"; drifting at the whim of the TMM. Figure 4.14 illustrates an example where violating this rule can cause problems.

The time map in Figure 4.14 shows that **token5** is clipped by **token7** and that it endures at most 5 units. The beginning of **token3** is coincident with the end of **token2**, and **token3** clips **token1**. It's not clear in Figure 4.14, but the beginning of **token2** is not completely constrained with respect to the beginning of **token1**, however the points corresponding to the *beginning* of **token1**, the beginning and end of **token6**, and the beginning of **token7** are all completely constrained with one another. Assume that **token5** depends upon **token1** persisting throughout **token4**. Now let's suppose we add the following constraint: `(elt (distance (begin token2) (end token5)) 6 6)`. This constraint will result in `Q` no longer being protected throughout **token4** so that **token5** will be OUT, so **token7** will no longer clip the persistence of **token6**, so `Q` will again be protected throughout **token4**,

```
Frame of reference: (begin token2)  Scale: 1.0

token1 Q
 ||-----------------|
token2 E1
                |---|
token3 (not Q)
                    ||---------------------------------------------->
token4 E2
       |---------|
token5 P
                 ||---[0,5]---|
token6 E3
                          |---|
token7 (not P)
                              ||------------------------------------>
```

Figure 4.14: Violating the freedom of the end of a persistence

so token5 will be IN, and so on ad infinitum.

I realize that this example requires some effort to understand. I guarantee, however, that the (insidious) bugs that result from failure to comply with rule 3 are infinitely more incomprehensible and difficult to analyze.

The lesson to be learned from this is that it's not safe to bank upon the end of a persistence. In most cases, what is really needed is either a protection or some sort of forward inference rule that alerts you to predicted events or effects. I have trouble coming up with plausible examples where one would want to constrain the end of a persistence. I'll give an example of where it might make sense, but (I think) other methods are more appropriate.

Suppose that you want to make sure that you have enough diesel fuel for the generator backup system for your computing facility so all the machines won't crash during an extended power failure. In light of this, you have a task to call your supplier not later than 10 minutes after the first indication of a power failure (*i.e.*, the persistence of the "power available" token ceases). To make this a bit more plausible, suppose that you're really cheap and you don't want to buy oil unless it's absolutely necessary, the supplier will deliver within 20 minutes, and you have 45 minutes of battery backup (so you have 15 minutes leeway). This constraint on your task won't really help you to plan, since you have no idea of the conditions in force at the time the plan actually will be carried out (being at the mercy of the weather and the utility commpanies). So all you really can expect is to have the token representing the unexpanded task constrained correctly. The constraint on the task won't even tell you when the task is imminent so you can expand it in time. There are better ways of alerting oneself of the need to plan for, and carry out, such tasks. One way would be to set up a forward chaining rule that would trigger whenever an event was added which caused a power outage. In effect, this had to be done by the TMM anyway in order to detect when the token corresponding to the power being available is clipped by some contradictory token. As with the other rules, rule 3 can be broken if you're careful. It is my contention, however, that this rule causes few hardships and any functionality lost by its enforcement is easily made up in other ways.

Rule 4

No new objects of data type TOKEN are added during TRMS update.

This just makes the proof simpler. In later sections I'll break this rule to handle certain forms of temporal forward chaining.

Rule 5

No two time tokens asserting contradictory facts can be shown to have coincident beginning points.

This is just another criterion to ensure that the data base not have any outright unresolvable contradictions.

The above 5 rules constitute the basic assumptions about how the TMM is employed necessary to guarantee termination. Before I present the proof of correctness I want to review some ideas presented in previous sections, point out some of the places where the 5 rules described above will come into play, and introduce some additional terminology that will be used in the proof.

4.3.6 Some preliminaries to a proof of correctness

In this section I've spoke about two important data dependency relations. The first concerned dependencies set up to clip persistences in keeping with the apparent contradiction criterion. In that case we had a special constraint called a clipping constraint whose ddnode was justified by a TCONDIT ddnode and the two ddnodes associated with the two contradictory tokens. The resulting data structures along with their associated ddnodes and justifications were depicted in Figure 4.3. The second type of dependency relation concerns protections and was illustrated in Figure 4.3, again with its associated data structures. In each case, ddnodes corresponding to TCONDITs and TOKENs play the critical roles. Recall that a TCONDIT has justifications which take the form of some number of CONLINK ddnodes. These CONLINKs define a path through the time map which would satisfy the TCONDIT's satfun given that all the ddnodes were IN. When a ddnode associated with a constraint (pair of CONLINKs) becomes IN, this causes the propagation of the constraint. This, in turn, may result in certain TCONDITs being updated to reflect the change in temporal connectivity. When a constraint ddnode becomes OUT no propagation occurs, but TCONDIT ddnodes can become OUT as a result of justifications that refer to the diminished constraint ddnode[8].

Rules 2 and 3 together guarantee that, after the constraint changed by the user (thus instigating the update) is propagated, the relative ordering of the beginning of the tokens in the time map will not change over the course of the rest of the update. This is because the only constraints that change (by rule 3) are *clipping constraints*, and since they only constrain the endpoints of persistences (which are otherwise unconstrained by rule 2) these constraints cannot change the relative order of the beginning of tokens[9] So once the initial propagation occurs, the beginning of all tokens will remain in the same partial order throughout the rest of the update. When I say that one token is earlier than another, or refer to the set of all tokens which occur earlier than a given token, I am speaking only about the relative ordering of their beginning points.

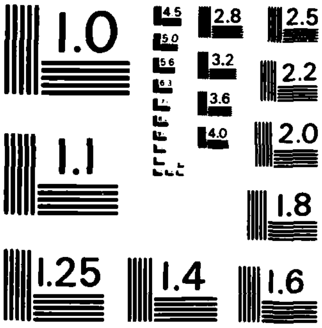
Finally, before we get on with the proof, I want to introduce two properties of tokens. A token is said to be stable with respect to its status (abbreviated S/STATUS) just in case each earlier occurring token is both stable with respect to its status and stable with

[8] When a TCONDIT becomes OUT it tries to reestablish its lost status by instigating a search. This is pictured in Figure 4.7. In discussing the correctness of the temporal reason maintenance algorithm I will ignore this. It is simply an efficiency matter that allows us to avoid adding justifications for all possible satisfying paths in the network. A new path is added as a justification only if its composite label is not subsumed by the composite label of an existing path.

[9] We're assuming here that all apparent contradictions can be resolved. This is guaranteed if the lower bound on all persistences is 0. If not and an apparent contradiction cannot be resolved the system will detect an inconsistency and prompt the user to resolve it.

AD-A161 242   TEMPORAL IMAGERY AN APPROACH TO REASONING ABOUT TIME   3/3
FOR PLANNING AND PRO..(U) YALE UNIV NEW HAVEN CT DEPT
OF COMPUTER SCIENCE T DEAN OCT 85 YALEU/CSD/RR-433
UNCLASSIFIED   N00014-83-K-0281   F/G 9/2   NL

END



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

respect to the clipping that that token is licensed to perform (abbreviated S/CLIP). If a token is OUT, then it's not licensed to perform any clipping. In general, a token, T1, is licensed to clip any earlier contradictory, T2, just in case the ddnodes corresponding to T1 and T2 are IN, and there exists a path through the time map such that the beginning of T2 can be shown to precede the beginning of T1, and the ddnodes corresponding to the CONLINKs in the path establishing this ordering relation are also IN. Of course, this is just a restatement of the top level invariant which began Section 4.3.5. To prove correctness we want to show that the algorithm terminates in a state such that every token is both S/STATUS and S/CLIP. Intuitively (and simplifying somewhat), a ddnode is IN just in case all of the protections participating in its justification are IN. A protection is IN just in case the associated token is IN and it spans (persists throughout) the required interval. Every IN token clips all earlier occuring contradictory tokens.

4.3.7 Correctness

The proof will proceed by induction. The idea is to show that as the update progresses tokens become S/STATUS and S/CLIP on a regular basis, and, once they achieve this lofty position, they do not revert to an unstable state. The only thing that sustains the update is change in the status of tokens and, since there are a finite number of tokens and these are eventually all S/STATUS, the algorithm must terminate; and, since all the tokens are both S/STATUS and S/CLIP, it terminates correctly. Let m be the total number of tokens in the data base and assume that $m > 0$.

Basis step: When the initial propagation is finished, there is a nonempty set of tokens each of which have no earlier tokens in the partial order. These tokens are necessarily S/STATUS since by rule 1 they have no protections as this would imply that there are earlier tokens. In addition, they must be S/CLIP since there are no earlier tokens and hence there are no earlier contradictory ones. The status of these tokens must have been given by fiat, and hence this status is correct and will not change during the remainder of the update.

Induction hypothesis: I assume that n tokens are both S/STATUS and S/CLIP where $0 < n < m$ and that further these tokens have the correct status and this status will not change during the remainder of the update.

Induction step: It's easy to see that there must exist some number (> 0) of tokens that are S/STATUS but not S/CLIP. By the induction hypothesis there are tokens which are not both S/STATUS and S/CLIP, so necessarily there are such tokens that are not earlier than any of the others and these must be S/STATUS by definition. Let's consider one such token, call it T1. First, I want to show that the status of T1 is correct and that it won't change. Since all earlier tokens have the correct status and have clipped just those tokens they are entitled to, the protections of T1 must be IN just in case their corresponding tokens are IN and span the required interval. If they are IN, then whether or not they span the required interval should be evident in the status of the protections' TCONDITs. The only tokens that could stop the tokens associated with the protections from persisting long enough (by rule 1) are those that are earlier than T1 and hence have done all the clipping they're entitled to. Propagation of constraints guarantees that the clipping constraints will be reflected in TCONDITs which keep track of the status of protections. So T1 has the correct status and it can't change.

Now we have to show that T1 will clip all the tokens it's supposed to, and, once it has performed this clipping (and its associated constraint propagation), it won't do any more. All clipping constraints depend upon a pair of tokens and a TCONDIT which relates the beginnings of the two tokens. We can assume that the TCONDIT does not change status after the initial constraint propagation. So, the only thing that can inititiate further constraint propagation is a change in the status of tokens. Since both T1 and any earlier tokens which it could clip will not change any further (by the induction hypothesis and the argument above), we can assume that the status of all clipping constraints that would result in T1 clipping an earlier token T1 are stable. Moreover, the signal functions associated with these clipping constraints will see to it that these constraints are reflected in the TCONDITs for protections justifying later tokens. This last may result in a change in the status of other later tokens, and hence cause further changes in clipping the next time the static RMS is called. However, T1 has done all the clipping it can do, and this will not change, since the status of T1 will not change. This shows that T1 is S/CLIP thus completing the induction argument.

In this way, we can see that the process will not stop as long as changes keep occurring in the status of tokens. However, since eventually all tokens are both S/STATUS and S/CLIP the process must terminate and terminate correctly[10].

[10] The changes in token status do not necessarily occur in the most efficient order. In the worst case, a single

4.3.8 Review

We now have a basic framework for performing temporal reason maintenance. The TMM resolves apparent contradictions and computes the conditions for belief in assertions that depend upon facts spanning intervals. The system relies upon a hybrid static RMS along with a mechanism for maintaining invariants to accomplish a form of temporal data dependency update. A proof of correctness for the update algorithm (with certain restrictions on the use of the data base) has been provided. As in the case of the static RMS, the update is incremental and performs well in practice.

4.4 Abductive temporal queries

Consider the following simple query in the time map of Figure 4.15:

```
(and (tt (begin produce31) (end produce31)
         (operational-status ?machine in-service))
     (instance-of ?machine lathe))
```

According to the time map, the token produce31 can begin before or after the beginning of the token running47. So what should one expect from such a query? In most deductive retrieval systems, a query returns a set of bindings such that the instantiation of the query form with those bindings follows in some way from the information in the data base. In the case of the above query and the time map of Figure 4.15, there is no such substitution.

If, on the other hand, we assume an additional constraint, `(elt (distance (begin running47) (begin produce31)) 0 *pos-inf*)`, the query will succeed with substitution {(machine lathe17)}. What we might want then, is for the query to return with a set of bindings and a set of additional constraints such that if the additional constraints were imposed, the instantiation of the query form with those bindings would follow from the augmented time map. This is the abductive interpretation of the true-throughout predicate

token could change status on the order of n times where n is the total number of tokens. In practice this would be hard to even purposely arrange, but less extreme situations occur frequently and can cause the TRMS to do more work than is necessary. One way in which the TRMS algorithm might be improved would be to sort the signal functions for clipping constraints so that they swept forward in time; never propagating a clipping constraint until both of the tokens associated with it were stable with respect to their status. It would take some experimentation to determine whether or not the sorting overhead would outweigh the cost of unnecessary token status toggling, but it would certainly pay off if anything like the worst case began to manifest itself regularly.

```
Frame of reference: *ref*  Scale: 1.0

running47 (operational-status lathe17 in-service)
                ||--------------------------------------------------------------->
produce31 (produce order43 <(type widget) (number 4)>)
 |-----------------------------------------|--------|
```

Figure 4.15: Simple time map

discussed in Section 3.6. The additional constraint was referred to as an abductive premise. In a sense, the query is a request to find a hypothetical situation in which the query form is true relative to a set of bindings. Hypothesizing is limited to restricting the current partial order. What we want here is for the TMM to help us see possibilities that are not immediately apparent in the time map. In planning, this essentially consists of suggesting a set of additional constraints on an existing partially specified schedule that will help the planner in carrying out a particular plan to achieve some task.

Handling conjunctions of **tt** formulae is a bit more complicated. Consider the following query in the time map of Figure 4.16:

```
(and (tt (begin produce31) (end produce31)
         (operational-status ?machine1 in-service))
     (instance-of ?machine1 conveyor)
     (tt (begin produce31) (end produce31) (attachment ?machine2 ?type))
     (instance-of ?machine2 lathe)
     (member ?type !<widget-bit gizmo-bit>))
```

In this case, there is only the answer with bindings {(machine1 conveyor7) (machine2 lathe17) (type gizmo-bit)} and additional constraints (elt (distance (begin running47) (begin produce31)) 0 *pos-inf*) and (elt (distance (begin gizmo12) (begin produce31)) 0 *pos-inf*). The other possibility, involving the tokens running47 and widget14, won't work, because the intersecting interval isn't long enough. The query mechanism recognizes this by imposing temporary constraints during the query.

The algorithm is shown in Figure 4.17. The temporary constraints added in step 4 are abductive premises (see Section 3.6). These constraints are actually added to the time map on a permanent basis if an assertion occurs in the context of an ANS referring to them. The temporary constraints added in step 6 of Figure 4.17 are simply a means of checking to see

```
Frame of reference: *ref*  Scale: 1.0

running47 (operational-status conveyor7 in-service)
                ||-----------------|
widget14 (attachment lathe17 widget-bit)
                               ||----------------------------------------------->
gizmo12 (attachment lathe17 gizmo-bit)
                   ||----------|-----|
produce31 (produce order43 <(type widget) (number 4)>)
 |----------------------------------------|---------|
```

Figure 4.16: A slightly more complicated time map

```
Given a query of the form (tt pt1 pt2 P)
 1. If you're backtracking in a conjunctive fetch,  remove  any
   temporary constraints added previously in steps 4 and 6.
 2. Try to find a token with schema unifying with P that hasn't
   been tried yet. If none exist, fail.
 3. If the token can't begin before pt1, return to 2.
 4. If it can begin before pt1 but currently doesn't, add a
   temporary constraint to ensure that it will.
 5. If the token ends before pt2, remove any constraints
   imposed in 3 and return to 2.
 6. Add a temporary constraint to ensure that it ends after pt2.
```

Figure 4.17: Algorithm for handling tt queries

that conjunctions of tt formulae are satisfied properly. These constraints never become a permanent part of the time map.

The tricky part of the algorithm in Figure 4.17 is finding the tokens in step 1 and determining whether one point precedes another in steps 3 and 5. The TMM has two ways of indexing tokens. The first way we've already mentioned. Tokens can be referenced from their begin and end points in the network of constraints that is the time map. But it would be difficult to find a token matching a given pattern by searching through the time map checking all points. Instead, tokens are also stored in a discrimination network to facilitate finding all tokens matching a given pattern. In this respect, tokens are no different from other assertions in the data base. But, of course, it's not sufficient to find a token matching a given syntactic pattern; you generally want to know the interval over which its associated proposition is believed true. To determine this will require some amount of search.

In a temporal query of the form (tt pt1 pt2 P), the time map finds potentially applicable tokens in the following way. First, it marks all tokens unifying with P with a tag unique to that search[11]. Next it initiates a search similar to that described in the section on constraint propagation (Section 4.3.3). Paths are extended from both pt1 and the beginning points of all tokens just marked. The search is best-first with "best" being determined by the sum of the weights on the CONLINKs traversed in each path (the lower the sum the better the path). Objects of type TMPATH (described in the section on constraint propagation) are left on points to record the partial status of the search. When a path from a marked token collides with a path starting from pt1, the composite path is set aside as a measure of the offset of the beginning of the associated token from pt1. Usually the search is allocated a fixed amount of CPU time. If successful, the search returns with one or more paths indicating the best estimate found of the distance separating pt1 from the beginning of a token with schema matching P. The search can be extended by simply providing another allotment of CPU time. To clean up the markers and TMPATHs, the search has to be explicitly terminated. By allowing several searches to be in progress at the same time, the query routines can be more demand-driven (you try to do no more work than is absolutely necessary). This is especially useful if only one answer to a complex conjunctive query is needed. If you always needed all answers, then each individual search might as well be exhaustively carried out and the tokens and offset from pt1 doled out as needed.

Searches to determine the best known estimate on the distance separating two points

[11] Several searches can be in progress at the same time, so a token may have more than one mark.

are performed frequently in the time map, and it is worth the effort to (a) avoid them whenever possible and (b) optimize their operation if they are absolutely necessary. The performance of the actual search routines depend upon the same sort of factors that influence the performance of the constraint-propagation routines (*e.g.*, the number of points and constraints in the time map and degree of redundancy of the information encoded in the constraints). If all `CONLINKs` were given exactly the same weight, the search would be conducted in breadth-first manner. In applications involving sets of intricately interrelated events that can't easily be partitioned (*e.g.*, into distinct episodes), breadth-first search is probably the best strategy. The time of creation information on the `CONLINKs` could also assist in directing search in large time maps, but hasn't been shown to pay off in the applications tried so far. One technique for speeding up searches that has met with success involves the caching of derived estimates of the distance between selected pairs of points. These pairs of points are selected on the basis of some strategy for organizing the data base. I'll describe the details of caching in Section 4.4.3, along with an organizational strategy based on the task/subtask hierarchy used in planning. In the meantime, I want to show how the order of the conjuncts in a temporal query can influence the speed of retrieval by consolidating searches (and thereby avoiding unnecessary search).

4.4.1 Preprocessing queries to optimize fetches

Consider the following rather complex query:

```
(and (inst ?mach1 lathe)
     (capacity turning-radius ?mach1 12)
     (tt ?begin ?end (operational-status ?mach1 in-service))
     (inst ?mach2 cylindrical-grinder)
     (capacity turning-radius ?mach2 10)
     (tt ?begin ?end (operational-status ?mach2 in-service)))
```

It basically says to find an interval such that it's true thoughout the interval that there's an operational lathe of at least 12 inch turning radius and an operational cylindrical grinder of at least 10 inch turning radius. Now consider how the query will actually be processed.

The first two conjuncts `(inst ?mach1 lathe)` and `(capacity turning-radius ?mach1 12)` will serve to bind `mach1` to a particular lathe. Next, the form `(tt ?begin ?end (operational-status ?mach1 in-service))` will instigate a search for all tokens that match `(operational-status ?mach1 in-service)` given the bindings imposed by the previous two conjuncts and satisfying whatever constraints are imposed on points bound to the variables ?begin and ?end. It may find none for the lathe chosen, or it may find one for which there is no token satisfying the last three conjuncts as well. In either case, the deductive machinery will have to backtrack, find another substitution for ?mach1 that satisfies the first two conjuncts, and initiate another search for tokens which satisfy the resulting bindings. A good deal of work will be repeated in each search. The cost of setting up the search in the first place, plus the cost of the extending the search from whatever point ?begin is bound to, may have to be duplicated many times.

The TMM can avoid this repeated work by preprocessing conjunctive queries to extract constraints on conjuncts that require searching through the time map for tokens. The idea is that the constraints (conjuncts that constrain the binding of variables in conjuncts like (tt ?begin ?end P)) can be used to assist in marking tokens prior to initiating the search. The constraints provide an initial filter to limit search.

Sometimes the constraints will have to be used to generate additional bindings as well. So in the following query:

```
(and (instance-of ?mach parts-feeder)
     (attachment ?mach ?bin)
     (instance-of ?bin bar-stock-storage-bin)
     (tt ?begin ?end (operational-status ?mach in-service))
     (tt ?begin ?end (empty ?bin)))
```

the request is for an interval with an operational parts feeder with an empty bar stock storage bin. Some feeders may not have bar stock storage bins, so the second and third conjuncts provide a useful constraint to direct search. But, also, for a given parts feeder there may be several such storage bins, and hence the third conjunct also has to be applied as a generator for the fourth conjunct and a constraint on the fifth.

The TMM requires a bit of assistance on the part of the user to perform this sort of preprocessing efficiently. First, to avoid the overhead of preprocessing in handling conjunctions where it would have no advantage, the system requires that the user request preprocessing by replacing and with &. The connective & behaves just like and, except that it preprocesses the conjunction to speed up time map searches. Secondly, it is assumed that a formula that constrains a variable in a temporal conjunct (*e.g.*, one involving tt or t) will precede the conjunct it constrains. With this assistance, the system can speed up performance considerably for commonly occurring query types.

It should be noted that the simple techniques described here are designed to take advantage of properties of the algorithms used in a particular implementation of the TMM.

I expect that these techniques could be easily combined with the techniques for ordering conjunctive queries described in [Davis 85] to increase performance even more. As a simple example, it is relatively cheap to determine the number of tokens of a certain type present in the time map. In most cases, temporal conjuncts for which there are few tokens of the appropriate type should be tried first. The preprocessing necessary to make such decisions, is negligible in comparison with the time required for searching through the constraint network in order to determine the "when" of time tokens.

4.4.2 Setting up protections

Recall that in the section on deductive retrieval systems (Section 3.2), the use of deductive contexts were discussed along with an implementation in terms of objects of data type ANS. An ANS refers to a set of bindings and a set of ddnode/support type pairs useful for establishing the justification for assertions. Now I'd like to describe how temporal queries work in conjunction with assertions. There are two issues. First, how are the abductive premises generated during a query handled? and, second, what additional dependencies have to be added to the current answer in order to monitor the continued validity of assumptions made during the query?

In the following code fragment:

```
(for-first-answer (fetch '(tt pt1 pt2 P))
    (add some-predictions))
```

the current answer `ans*` is augmented in two ways during the fetch. First, the system adds to the answer a schema describing how to construct a protection corresponding to the warrant for believing `(tt pt1 pt2 P)` (see Section 4.3.2). Second, if necessary, the system adds to the answer an abductive premise corresponding to the constraint that some time token of type P precedes pt1 (see Section 3.6). These protections and abductive premises are said to be *latent* in that they are only directions for installing certain dependencies and adding certain constraints. A latent abductive premise is IN just in case it is contained the current answer. (Latent protections and abductive premises can be extracted from the current answer using the functions `extract-protections` and `extract-scheduling-constraints`, respectively.)

Since the protections and abductive premises are latent or unrealized, the user can keep track of several objects of data type ANS, each constituting a separate hypothetical situation.

Each ANS has the necessary information to reinstitute and make permanent its associated abductive premises, and set up the dependencies needed to monitor the continued validity of the temporal assertions in the original query. These separate ANSwers can be consulted to determine the exact nature of the commitments they require (additional constraints that must be added in order to justify the associated protections).

The normal way that a hypothetical situation is established (*i.e.*, its associated commitments added to the time map on a more permanent basis) is by making an assertion in the context of the answer associated with that hypothesis. That is, the user simply binds the desired answer to be the current answer (or uses one of the macros designed for that purpose), and then makes an assertion in the scope of that global answer. This will cause the necessary constraints to be added and the protections to be installed in the justification for the new assertion. The constraints in an answer can also be actualized by simply applying the function `realize-scheduling-constraints` to the answer. It makes no sense to actualize the protections unless they are used to justify something (protections that don't justify anything only waste time and storage).

4.4.3 Caching: exploiting the structure of time maps to expedite point-to-point fetches

One of the most frequently performed operations in the TMM involves determining the best bounds on the distance separating two points in the time map. This operation is carried out repeatedly in processing queries, updating type II protections (see Section 4.3.2), and in updating TCONDITs when their associated ddnodes become OUT (see Section 4.3.3). It is possible to speed up the computation of point-to-point distance estimates by keeping track of (caching) the best known estimates of distances separating selected pairs of points in the time map. If you cache distance estimates for all pairs of points, then, obviously, determination of the best estimate for a given pair is trivial. The entire computational burden rests upon the routines for updating the table used to store point-to-point distance estimates. An earlier version of the TMM actually did compute the best estimates for all pairs of points in a restricted portion of the time map called the *kernel* [Dean 84]. Updating this kernel required on the order of $n^3 \log n$ integer additions and array references where n is the number of tokens in the kernel. In typical applications, however, the kernel grew to several hundred tokens, and updating (resulting from adding one or more constraints) required several minutes even with all the important procedures optimized and coded in a low level language. It became apparent that much of the work expended in updating all of these point-to-point distances was wasted. The version of the TMM used in this dissertation performs selective caching; only certain pairs of points are chosen for maintaining an accurate estimate of their separation. This means that the burden of computing point-to-point distance estimates rest partially with the routines for updating the selected point-to-point distances, and partially with routines for determining a given estimate on demand.

The point-to-point distance estimates chosen to be cached are selected on the basis of a general strategy for exploiting the hierarchical structure of tokens in the time map. The heuristic search routines employed in determining distance estimates take advantage of the cached distance estimates in order to quickly converge upon a "good" estimate. The expectation is that for pairs of points for which one might reasonably need to know an accurate distance estimate, the system would perform as well as exhaustive search, and for pairs of points that are not normally related to one another, performance would degrade reasonably[12]. It is possible to "tune" the system (adjust the way constraints are weighted and the search routines compute the best path to follow) for a given application in order to get significantly faster response time with no decrement in performance compared with a system without caching. In this subsection we will explore the issues involved in determining point-to-point distances in some detail.

In the TMM, without caching and making no assumptions about the constraints in the time map, determining the best bounds on the distance separating two points will require on the order of n^3 arithmetic operations where n is the number of points in the time map. If we assume that the length of a path used in establishing the best bounds never exceeds some constant, then we can limit the search by simply limiting the length of paths explored. The assumption is that temporal connectivity is largely determined locally, and rarely does it require consideration of all the points in the time map. While this assumption is reasonable in most applications, we can still do a lot better. Tokens in a time map are constrained relative to one another in fairly regular ways: tasks are related to their sub and super tasks, persistences are constrained by some offset from the event that caused them, and

[12] I am relying upon the reader's intuitions here, but a simple criterion of reasonableness might be that the scale of the separation between a pair of points determines the accuracy of an estimate of their separation. For events widely spaced in the time map, no precise estimation of their separation is necessary even though given enough time an exhaustive search could supply such a precise estimation. For example if I worked hard enough I could determine to within a day how long it's been since my last visit to the dentist. For most purposes, however, "approximately three months" will work just fine.

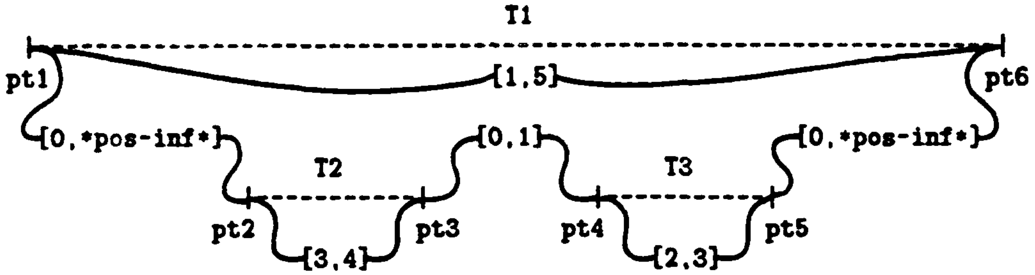


Figure 4.18: A simple network of points

prerequisite tasks are constrained to precede the tasks they serve. If we could rely upon certain point-to-point distances being known with precision (*i.e.*, the values cached in some way), then the search paths for most other point-to-point distances would be comprised primarily of these cached values. Good examples of candidate pairs of points for caching point-to-point distances are the beginning and end of a token and, for a token corresponding to a task, the beginning of the token and the beginning of the token corresponding to its immediate supertask in the task/subtask hierarchy. In order to motivate some of the lower-level details, I'll present a simple example illustrating some of the issues involved in caching and then proceed to demonstrate how selective caching pays off in certain circumstances.

Consider the diagram in Figure 4.18. The vertical lines labeled pt1 through pt6 indicate points. The bracketed numbers indicate low and high bounds on the estimated distance between the two points which they separate. T1, T2, and T3 are tokens corresponding to the intervals pt1 to pt6, pt2 to pt3, and pt4 to pt5 respectively.

Given the information supplied in the diagram, one should be able to determine that pt3 is coincident with pt4 (*i.e.*, (elt (distance p3 p4) 0 0)). Determining that the two points are coincident will require on the order of 6^3 operations using blind search. If the system selectively caches certain point-to-point distance estimates, then it can make such determinations with significantly less effort. Suppose that the diagram in Figure 4.18 represents a small part of a task network in a time map. Suppose that the three tokens T1, T2, and T3 correspond to tasks in this network, and that T2 and T3 are subtasks of T1. Now, suppose further that the system keeps track of the best estimates on point-to-point distances separating the following pairs of points:

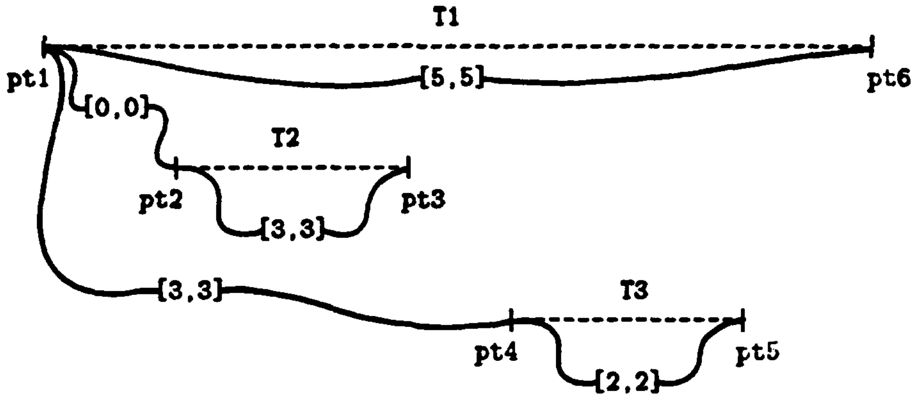


Figure 4.19: A simple task network with cached values

- the beginning and end of each time token
- the beginning of a task and the beginning of its supertask

Figure 4.19 shows the network of Figure 4.18 indicating the cached point-to-point distance estimates.

Notice that in this network, finding the best estimate between any two points will require at most a path of length 4. The search strategy can be stated as follows. If you are trying to extend a path from a point, choose a link corresponding to a cached value that takes you up (toward supertasks' in the task/subtask hierarchy. If the point has no such link, then take whatever link corresponding to a cached value you can get. If in the course of extending a path you find a second path struggling up the task hierarchy, see if you can combine the two paths to get the required distance estimate.

This strategy is not guaranteed to provide best estimates. If you need convincing, consider the network of Figure 4.18 with the constraint linking pt1 and pt6 labeled [1,6] instead of [1,5]. In this network, augmented using the caching scheme outlined above, our simple search strategy would return the value [-2,1] in response to a request for the best estimate of the distance between pt3 and pt4. The correct answer should be [0,1]. We can correct for this in most cases by combining a search strategy similar to the one outlined above with a bounded breadth-first search.

There are three main issues that have to be addressed with respect to our caching scheme:

1. improving cached values: determining when a new constraint can assist in providing a better estimate on the distance separating a pair of points selected for caching.
2. removing invalid estimates: determining when an old cached value is no longer valid due to the removal of constraints employed in its original derivation.
3. expediting heuristic search: using the cached values in a search strategy for speeding up the computation of point-to-point distance estimates.

The first two of these issues are handled by the same techniques used for keeping track of temporal conditions. Caching in the time map is implemented using objects of data type TCONDIT, described in the section on constraint propagation (Section 4.3.3). Recall that TCONDITs are used for keeping track of certain relationships between pairs of points. A given TCONDIT can be referenced from either of the two points it relates. During constraint propagation, if a path is found between the two points of a TCONDIT, an attempt is made to determine whether or not the path satisfies some criterion, and if so, the path is used to construct a justification for believing whatever relationship the TCONDIT refers to. In caching, TCONDITs are used to update special constraints, called *caching constraints*, that record the best known estimates of the distance separating pairs of points in the time map. Every CONLINK points to a ddnode that determines whether or not it is acceptable to traverse the corresponding directed edge in the time map. The ddnode for a CONLINK associated with a caching constraint is IN just in case the upper and lower bounds labeling the CONLINK are up to date. The propositions corresponding to to the current distance estimates of caching constraints are not directly identified with the propositional content of any particular ddnodes. This means that caching constraint ddnodes do not directly participate in the justification of other ddnodes. They can participate indirectly, however, by using the stored justification for the current cached value. This justification is guaranteed to be composed of ddnodes corresponding to user supplied constraints.

As with TCONDITs for keeping track of temporal conditions, the estimates for caching constraints are updated routinely as a part of constraint propagation. The satfun for a caching constraint just determines if the new-found path supports a better distance estimate than the current best estimate. If so, the assimfun installs the new estimate and replaces the old justification with one supporting the new path. Old estimates and their justifications are saved so that if the current justification ever becomes OUT the signal function associated with the constraint ddnode will install the next best estimate with a justification that will be IN given the current constraints. Since the justification for the current best estimate is just the ddnodes for constraints in the path, the ddnode will toggle OUT if the derivation for the cached value is ever undermined. In such a situation, the TCONDIT signal function will see to it that some valid estimate is installed as the current cached value. If the current estimate cannot be improved (*i.e.*, the upper bound is equal to the lower bound), the assimfun also removes the TCONDIT from the points that it relates (the TCONDIT signal function reinstalls if the estimate degrades for any reason). This means that in a time map in which many of the distances between points are completely determined, propagating constraints for new points will not be hampered by checking lots of caching TCONDITs unnecessarily.

Search strategies for taking advantage of cached values are implemented by simply weighting the CONLINKs associated with caching constraints. If you want search to be biased in favor of moving up the task/subtask hierarchy, then you associate little weight (or cost) with the CONLINK that goes from the beginning of a subtask to the beginning of its immediate supertask, but some more significant cost with moving the other way. In the implementation used in the examples in this dissertation, there was a single weight, *cache-bias*, associated with constraints for caching TCONDITs. Directional caching TCONDITs (like those that bias search up a task/subtask hierarchy) assign this weight to only one of the two CONLINKs associated with a caching constraint. The other CONLINK gets the default weight for regular constraints. All other caching constraints are bidirectional; that is both CONLINKs associated with the constraint are assigned the *cache-bias* weight. The only directional caching TCONDITs are task/subtask links and those that link a top level task to a reference point. The latter are used when a task is given a deadline with respect to some absolute time (*e.g.*, the customer needs this order before noon). Bidirectional caching constraints are used exclusively for caching the duration of time tokens.

The cached values make many searches extremely fast. A time map corresponding to a *pure hierarchy* is one in which each task is constrained only with respect to its immediately superior supertask in the task/subtask hierarchy or with respect to one its sibling tasks (both tasks share the same immediate supertask). Top level tasks are constrained only with respect to one another. If time maps were pure hierarchies, then all point-to-point distance estimates could be performed in time proportional to the depth of the hierarchy.

In experiments involving several top level tasks expanded into over a hundred subtasks through several levels of refinement, the results are fairly clear. The additional cost involved in handling caching during constraint propagation increased the time to actually construct such time maps by approximately ten percent over the time required without caching. The time required to determine point-to-point distances accurately in the time map with caching decreased significantly over the scheme using breadth-first search with a depth cutoff carefully selected to take into account the sort of connectivity expected in the time map. The scheme with caching broke even with the breadth-first scheme after determining approximately 20 point-to-point distance estimates. In robot problem solving tasks involving a time map containing on the order of 100 tokens, it is reasonable to expect hundreds of point-to-point distance estimates. In such situations, it is expected caching will result in significantly faster processing times, though no detailed comparisons have been made as yet.

One obvious obstacle to getting the caching scheme described above to perform well is the fact that in most applications time maps are simply not pure hierarchies. For example, networks of tokens corresponding to tasks typically become extremely tangled in the process of planning. This is due to the interleaving of tasks and the action of persistence clipping to resolve apparent contradictions. The average time to determine best possible distance estimates using the caching scheme is still much better than undirected (breadth first) search. I'm currently experimenting with various techniques for weighting CONLINKs that depend upon specific properties of particular domains and problem solving applications. Preliminary results appear promising, but we still need a lot more experience before we'll be able to tell to what extent savings made in speeding up searches are offset by the increased effort expended during the propagation of constraints.

4.4.4 Review

This section has explored some of the issues dealing with the processing of temporal queries. Methods for indexing time tokens and determining their relative offsets were discussed along with techniques for expediting queries. In particular, two techniques were explored. The first involved preprocessing conjunctive queries to exploit the manner in which searches are conducted, and the second dealt with the selective caching of point-to-point distance estimates.

4.5 Temporal forward chaining

There are two basic types of temporal forward chaining, and each serves a very different purpose. The first will be referred to as *overlap forward chaining* and is characterized by rules of the form (->t (and P_1 ... P_n) Q) where the intent is that Q is true whenever P_1 through P_n are simultaneously true. The second type is called *causal forward chaining*. This type has a variety of special forms, but all subscribe to the same general format. This consists of a set of facts, a triggering event, and a set of further events and facts that follow if the triggering event occurs, and the facts can be shown to span certain intervals. A rule of the form (pcause (and P_1 ... P_n) E Q) is interpreted as saying, if an event of type E occurs such that P_1 ... P_n are true throughout interval associated with the event, then Q is believed to be true immediately following the event and persisting indefinitely into the future. Events of type E are said to cause the persistence of facts of type Q in the context of P_1 through P_n being true. Causal forward chaining can be used to model some of the physics of the domain. It is useful in planning to assist in noticing interesting repercussions of the planner's knowledge. Not only do these two forms of forward chaining serve different purposes, but each one raises its own own idiosyncratic implementation issues. The following two sections will treat each in turn.

4.5.1 Overlap forward chaining

I'll begin by considering how one might go about implementing (static) forward chaining rules of the form (-> (and P_1 ... P_n) Q). I will assume familiarity with the sort of simple forward chaining rules found in most deductive retrieval systems. Simple forward chaining rules have two components: a trigger pattern and a consequent pattern. The procedural interpretation is that if an assertion unifying with the trigger pattern is ever added to the data base, then the system will substitute the unification bindings into the consequent pattern and assert the result to the data base. Simple forward chaining rules are notated (-> P Q).

It should be apparent that simple forward chaining rules will not do the right thing for rules of the form (-> (and P_1 ... P_n) Q); it's not likely that anyone will be adding anything precisely of the form (and P_1 ... P_n). So we'll create a new predicate, fwd, and define it in terms of -> so that it will do the right thing with conjunctive antecedents. The form (fwd !<P_1 ... P_n> Q) is just a notational variant of (-> (and P_1 ... P_n) Q).

The only difference is due to the way the deductive retrieval system treats `->`. For convenience, it is simple enough to make the assertion machinery smart enough to recognize when something of the form (-> (and P_1 ... P_n) Q) is added and convert it into the fwd format.

To implement forward chaining with conjunctive antecedents it is sufficient to add the following two simple forward chaining rules:[13]

```
(-> (fwd !<?first-conjunct !& ?remaining-conjuncts> ?consequent)
    (-> ?first-conjunct
        (fwd ?remaining-conjuncts ?consequent)))

(-> (fwd !<> ?consequent) ?consequent)
```

Notice that in the first rule above the consequent pattern is itself another forward chaining rule. A rule spawned in this way is called an *intermediate* rule. The proliferation of such rules can add considerably to the cost of implementation, and hence we should not employ them carelessly.

Forward chaining techniques, like any other deductive strategy, can be applied in inappropriate circumstances. The functionality of fwd is potentially dangerous if handled carelessly. A simple rule like (fwd !<(trans ?x ?y) (trans ?y ?z)> (trans ?x ?z)) used to generate the transitive closure of some relation trans can result in something on the order of n^2 assertions of the form (trans ?x ?y) where n is the number of objects that are explicitly referred to in such assertions. In this case, there would also be an equal number of intermediate rules spawned. In implementing deductions using forward chaining, the time/space tradeoffs should be carefully considered. Once committed to using forward chaining, the user should be alert to the possibilities for avoiding unnecessary work.

Notice that if we ordered the antecedent conjuncts in the order of least likelihood of appearing, a considerable amount of work could be saved in some cases. This is one way in which the user can help in eliminating unnecessary forward chaining. In the time map we'll see how the system can also help in this respect.

In static forward chaining rules, all the variables are implicitly univerally quantified. In temporal forward chaining there is also implicit quantification over time tokens. A rule of the form (->t (and P_1 ... P_n) Q) is interpreted as saying, where there exist time tokens

[13] Note that in any reasonable system supporting simple forward chaining it does not matter the order in which (-> P Q) and P are added. In either order Q should be in the data base.

asserting $P_1 \ldots P_n$ such that the intersection of these time tokens is not empty, then there exists a time token asserting Q whose duration is exactly that intersection. A first pass implementation would follow along the lines of the static case, except in this case we'll have to keep track of the time tokens. We start with a predicate **tfwd** that that serves as an internal form for ->t rules and provides an extra argument to keep track of the tokens participating in a particular application of a ->t rule. The system transforms assertions of the form (->t (and $P_1 \ldots P_n$) Q) into the internal form (tfwd !<$P_1 \ldots P_n$> !<> Q). To begin with we'll need the following rules:[14]

```
(-> (tfwd !<?first-conjunct !& ?remaining-conjuncts>
          ?list-of-tokens-involved-thus-far
          ?consequent)
    (-> (time-token ?first-conjunct ?new-token)
        (tfwd ?remaining-conjuncts
              !<?new-token !& ?list-of-tokens-involved-thus-far>
              ?consequent)))

(-> (tfwd !<> ?tokens ?consequent)
    (call (establish-consequent-token ?consequent ?tokens)))
```

Of course, the function **establish-consequent-token** is going to be doing all the interesting work here, but before we describe its operation we should note a rather glaring potential for inefficiency.

Consider a rule of the form:

```
(->t (and (operating ?machine1 ?location1)
          (instance-of ?machine1 paint-sprayer)
          (operating ?machine2 ?location1)
          (instance-of ?machine2 arc-welder))
     (danger-of-explosion ?location1))
```

This just says that, if a paint sprayer and an arc welder are operating in the same location, there is danger of an explosion. The problem is that the function **establish-consequent-token**, which we can presume is likely to be expensive, will be executed even for pairs of tokens that are not in the least likely to overlap. If the paint sprayer is only used for half an hour in the morning, and the arc welder only in the afternoon, we will still have to incur whatever costs are associated with executing **establish-consequent-token**. In the

[14] Recall from Section 3.2 that when the predicate **call** is encountered during forward chaining the embedded LISP form is evaluated. The arguments to the function are not evaluated except in the sense that the bindings for the variables are substituted for the variables themselves. It is assumed that the all the variables will be bound.

case of rules with a larger number of antecedent conjuncts, there may also be a considerable cost associated with the spawning of intermediate rules. For instance, suppose that a rule with five conjuncts has resulted in the generation of the intermediate rule (-> (time-token P_3 ?anon17) (tfwd !<P_4 P_5> !<?new-token tok2 tok1> Q)). Now suppose that tok1 and tok2 do not (currently) overlap. If there are lots of tokens matching P_3, then a great deal of unnecessary work will be done in spawning useless intermediate rules. Of course we can't just forget about tok1 and tok2, as the constraints may change and at some point their overlap be nonempty, but we should be able to suspend work until such a change occurs.

We can avoid a certain amount of unnecessary work by establishing some extra conditions that prevent forward chaining in the event that two of the tokens participating in a particular application of a rule can be shown not to overlap. The details are rather hairy, but the basic idea is quite simple.

The trick is to make the intermediate forward chaining rules of the form:

```
(-> (time-token ?first-conjunct ?new-token)
    (tfwd ?remaining-conjuncts
          !<?new-token !& ?list-of-tokens-involved-thus-far>
          ?consequent))
```

depend upon[15] justifications which are OUT just in case it can be determined that any pair of tokens in ?list-of-tokens-involved-thus-far cannot possibly overlap. These justifications are implemented using TCONDITs. The actual strategy for setting up these justifications can vary. You can set up a justification for each pair of tokens, or you can set up justifications for selected pairs. Strategies that set up justifications for selected pairs tend to generate a greater amount of unnecessary forward chaining than schemes that set up justifications for all pairs. As usual, there's a tradeoff. In the time map used in the dissertation examples, the only justifications set up are between consecutive tokens in ?list-of-tokens-involved-thus-far. This seemed to be effective for squelching unnecessary forward chaining in most cases.

Now let's return to the question of what the function establish-consequent-token does. We want to construct a token that is IN just in case the set of involved tokens (*i.e.*,

[15] Simple forward chaining rules have their own associated ddnodes. The rule does not have its intended effect unless its associated ddnode becomes IN. The actual consequent assertion part of forward chaining is accomplished using signal functions. The assertion will depend upon, among other things, the ddnode associated with the rule.

```
Frame of reference: (begin operating6)  Scale: 1.0
                        •
operating5 (operating paint-spraying-unit13 work-bay4)
               ||------------------|
operating6 (operating arc-welding-unit46 work-bay4)
                         ||--------------------------------------------->
danger35 (danger-of-explosion work-bay4)
                         |--------||
```

Figure 4.20: Token derived by temporal forward chaining

those tokens collected in the course of forward chaining) have a non-empty intersection. To do so, we construct a special sort of token called a *derived* token. Tokens constructed by the user have the property that their beginning point is always constrained to occur before their end point. Derived tokens have no such property. In fact, derived tokens are determined to be IN just in case it can't be shown that the "end" of the token comes before its "beginning". If we add the right set of constraints, this will coincide with the set of *involved tokens* having a non-empty intersection.

The procedure is simply to constrain the beginning of the derived token to coincide or follow the beginning of all tokens in the set of involved tokens, and constrain the end to coincide or precede the end of all such involved tokens. The ddnode corresponding to the derived token has a single justification that consists of those dependencies extracted from the current answer at the time `establish-consequent-token` is executed, a set of in-justifier ddnodes corresponding to the set of involved tokens, plus the out-justifier corresponding to the ddnode associated with the temporal condition `(pt=< (end` *the-derived-token*`) (begin` *the-derived-token*`))`. This will ensure that the derived token is IN just in case all of the involved tokens are IN and their intersection is non-empty.

This establishes that the token exists but it fails to pin down very accurately when the derived token begins. This raises a rather complicated issue for which I have no complete answer. Consider the time map in Figure 4.20. According to this map it seems reasonable to assume that the beginning of the token `operating6` and the beginning of the token `danger35` are coincident, but given the constraints we've described thus far for derived tokens there's no way of deducing this.

Let the set of candidate beginners be the set of all beginning points of tokens in the set of involved tokens. We can eliminate from this set any points that precede other points in the set. If there is only one candidate beginner then it can be made coincident with the beginning of the derived token. The dependency network for managing this is set up by establish-consequent-token. It incorporates a number of TCONDIT/constraint pairs similar to those used for clipping persistences to resolve apparent contradictions. This method of "pinning" down the beginning of the derived token does not require that all the tokens be totally order, but it is certainly not as accurate as it could be. Suppose that a data base for managing a real estate firm has the following rule:

```
(->t (and (gas-service-connected ?residence)
          (electric-service-connected ?residence))
     (available-for-occupancy ?residence))
```

Suppose further that it's known that the house at 315 West 42nd Street has had both gas and electric services connected sometime in the morning. But there is no way of determining the order in which the actual connections occurred. Whatever else is known, it's clear the house at 315 West 42nd Street is ready for occupancy after 12:00, but the scheme described above won't allow you to conclude that. In most cases this will not cause a problem. It is still possible to conclude the duration persistences accurately, but it can cause incompleteness in the case of interactions with causal forward chaining rules of the sort to be described next. It doesn't tell you false information; it just misses some rather obvious inferences. I have made no effort to correct this deficiency in the current time map routines. I suspect that a general solution will require a good deal of computation. Perhaps I will feel more motivated to correct the problem when this sort of forward chaining proves itself to be a critical part of temporal reasoning and one that is seriously plagued by problems arising from just this incompleteness. For now, the current scheme is serving adequately. In fact, it is causal forward chaining that promises to be the more useful of the two types of forward chaining, and, by itself, causal forward chaining is not hampered by the problem we've just been discussing.

4.5.2 Causal forward chaining

This subsection describes techniques for implementing auto-projection rules that capture the underlying physics of a domain. In Chapter 3 we saw examples of such rules that made it easy to reason about how parts move around a factory under the influence of robot arms and conveyors. In this section, I will sketch the implementation for performing a special class of auto-projection rules involving the predicate pcause.

The actual implementation of auto-projection rules is relatively simple in comparison with overlap chaining. I'll describe one way[16] that pcause might be implemented, but the more general auto-projection rule is not much more complex.

First rules of the form (pcause (and P_1 ... P_n) E Q) are transformed into (-> (time-token E ?tok) (->causal !<P_1 ... P_n> ?tok Q)). Next we have a set of simple forward chaining rules similar to those used for implementing overlap chaining rules:

```
(-> (->causal !<?first-conjunct !& ?remaining-conjuncts>
              ?trigger-token
              ?consequent)
    (-> (time-token ?first-conjunct ?new-token)
        (call (pcausal-complex ?trigger-token
                               ?new-token
                               ?remaining-conjuncts
                               ?consequent))))

(-> (->causal !<> ?trigger-token ?consequent)
    (and  (time-token ?consequent ?result-token)
          (elt (distance (end ?trigger-token) (begin ?result-token) 0 0))))
```

The function pcausal-complex simply asserts (->causal ?remaining-conjuncts ?trigger-token ?consequent) dependent upon ?new-token spanning ?trigger-token. This dependency tends to squelch unnecessary forward chaining.

In Chapter 3, I mentioned that temporal forward chaining rules could be "temporally gated". This meant that one could limit the scope of temporal forward chaining rules by simply expressing them as time tokens. The assertion:

```
(time-token (pcause (and (operational-status ?appliance in-service)
                         (at ?appliance ?location)
                         (instance-of ?appliance television))
                    (lightning-strikes ?location)
                    (operational-status ?appliance fried))
            summer-prediction31)
```

is meant to capture the rule that a working TV in the vicinity of lightning striking will probably be put out of commission. But you might only believe that rule during the later part of the summer when the lightning storms are really violent. In such a case you could

[16] The actual implementation takes advantage of the details of the deductive retrieval system. The implementation shown is primarily for tutorial purposes though it conforms to the spirit of the actual TMM code.

simply constrain the token summer-prediction31 to span only the appropriate interval. The token is said to gate the rule.

Implementing gated temporal forward chaining rules is quite simple using the methods we've been discussing in this section. All that is required is a simple preprocessor to watch for tokens with the proper schemas and some means of setting up the necessary dependencies and simple forward chaining rules.

4.6 Dealing with choices

This section describes the reason maintenance system developed for the TMM. This RMS combines features of McDermott's system for handling contexts [McDermott 83] and deKleer's assumption based RMS [deKleer 84]. The RMS used in the TMM supports the sort of reasoning about alternatives described in Section 3.8.

Section 3.8 described an *option* as a special sort of premise. An option is a proposition that has no justifications, but participates in the justifications of other propositions. Options are used for labeling other assertions in order to facilitate reasoning about alternatives. Simplifying somewhat, if an assertion is labeled with an option, then that assertion is believed only if that option is taken. A data base containing assertions labeled with options represents a set of overlapping belief configurations [Martins 83]. Each configuration corresponds to a set of options and the assertions that are believed given that those options are taken.

Options are a special instance of what are called *gating objects*. Every ddnode corresponding to an assertion in the data base has a label consisting of a boolean combination of gating objects. The system sees to it that these labels are maintained in accord with certain criteria for consistency and well foundedness. The label is kept in disjunctive normal form as a list of justifications composed solely of gating objects. Routines outside the RMS interpret these labels for various purposes. The interpretation of a label usually consists of considering one or more possible assignments to the set of gating objects (considered as boolean variables) where an assignment is just a mapping from gating objects to the set {IN,OUT} (IN being thought of as 1 and OUT as 0). A label is said to evaluate to IN or OUT under an assignment to the current set of gating objects.

The notion of consistency and well-foundedness for Doyle type data dependency networks can be extended to deal with gating objects in a straightforward manner. McDermott's paper [McDermott 83] describes such an extension for implementing *contexts*. A context can be thought of as simply a subset of the set of all assertions. Each context can be considered as a separate data base. A context is associated with an assignment to gating objects. An assertion is believed in a given context just in case its label evaluates to IN under that context's assignment. The status of an assertion with respect to a given context is just the valuation of that object's label under context's assignment. McDermott's techniques make it possible to perform reason maintenance in such a way that switching between contexts is handled efficiently. Switching contexts essentially requires no more than making sure that the indexing machinery retrieves only propositions corresponding to assertions with labels that evaluate to IN under the current context's assignment. The data base is said to be *gated* by the current context. Reason maintenance in McDermott's system effectively computes the status of all assertions with respect to all possible contexts (*i.e.*, all possible boolean combinations of the current set of gating objects).

In the time map, gating objects can be used to establish McDermott-type contexts. Their primary use, however, is for reasoning about several (possibly exclusive) alternatives simultaneously. This is more in line with deKleer's system for doing qualitative reasoning [deKleer 84].

DeKleer's system is an *assumption based* RMS. Instead of ddnodes we have *values* consisting of a proposition, a set of assumptions under which the proposition holds, and a set of justifications. Assumptions correspond roughly to what I have been calling gating objects. Unlike Doyle's system in which each proposition is identified with a unique ddnode, any number of values can refer to the same proposition. If the set of assumptions for a given value is found to be contradictory, then the value is removed from the data base. Values don't switch from being believed to not being believed. One advantage of deKleer's approach is that the notion of context is built in; every set of assumptions constitutes a context. One disadvantage is that we have lost our handle on belief. DeKleer's system won't tell us if there is no longer a warrant for believing in the proposition P.

Suppose that my current plan for going to New York this weekend depends upon a friend giving me a ride into Manhattan. If that ride falls through, then I will probably want to consider another means of transportation. It is precisely because the ride is no longer available that I am willing to consider alternatives. Noticing such lapses of belief and taking action to respond appropriately requires some method for backtracking, which deKleer's system seeks to avoid. DeKleer assumes that the set of choices being explored always lead to a solution; there is never any need to go back to a previous decision and propose additional alternatives. In most planning applications, the computational realities demand that we make choices that may later be retracted. The conditions under which a given choice is made can change. We cannot at all times pursue all possible alternatives, and hence some form of backtracking is unavoidable. On the other hand, this chapter is arguing that there are occasions in which it is useful to explore some number of alternatives simultaneously.

We need the functionality of the Doyle-type RMS in order to detect when our beliefs are undermined in the course of gathering new information. We need the functionality of the deKleer-type RMS in order to reason about a number of alternatives simultaneously. It turns out that we can modify McDermott's RMS to get exactly what we need. To show how, I want to begin by demonstrating what McDermott's system lacks.

Recall the example from Figure 3.21 on page 126. In the figure, the ddnode for the proposition `(linger-over-lunch lunch-spot)` depends upon our choosing `(= lunch-spot cafeteria)` and not choosing `(= lunch-spot diner)`. Of course *we* know that `(= lunch-spot cafeteria)` and `(= lunch-spot diner)` are mutually exclusive choices, but what's to stop the program from making inappropriate combinations? For instance, what's to stop us from formulating a plan that makes use of the facts that if you eat early, the food is fresher and there's more variety, and if you eat later, it's easier to find a table? Obviously, you can't take advantage of both, but how does the RMS assist in avoiding such impossible combinations?

In the deductive system that employs McDermott's RMS, the problem just doesn't come up. The system always has a context or *current data pool*. A data pool is just an assignment to the boolean variables or gating objects that comprise ddnode labels. At any one time the user sees only assertions in the data base with labels that evaluate to IN under the current assignment (data pool). It is assumed that the user will not create and reason in a data pool which supports two exclusive alternatives.

In the system employed by the time map, there is no notion of a current data pool. In some sense we are searching for a data pool: the right combination of choices that will make everything come out for the best. It would be inefficient to search through all the possible data pools sequentially because we don't know what we're looking for, and we have no idea when it will suddenly become apparent that we've found it. The planner would have to be continually cycling through all the possible data pools.

In the time map, the query routines are designed to return not only a set of bindings and a restriction on the current partial order but also a set of gating objects that must be chosen (IN) or not (OUT) in order for the query to succeed. This last constitutes a partial assignment to gating objects, and warrants one or more partial world descriptions (PWDs). A PWD consists of all assertions whose labels evaluate to IN under an assignment satisfying certain additional constraints specified by the user. Such an assignment is called a choice assignment. The additional constraints determine what constitutes a "consistent" description of the world.

An important part of assisting the user in reasoning about alternatives involves eliminating from consideration impossible or undesirable combinations of alternatives. Certainly we can't combine exclusive alternatives. It's also possible that the user might wish to specify that certain combinations of alternatives lead to unpleasant circumstances and should no longer be considered. Both of these are accomplished by allowing the user to specify conjunctions of gating objects that are not to be considered. Such a conjunction is traditionally referred to as a *nogood* [Stallman 79]. If the user specfies that the conjunction of gating objects n1 and n2 is nogood, then all assignments associated with PWDs must assign one of n1 or n2 OUT. A set of mutually exclusive alternatives generates one nogood for each pair of alternatives. In certain situations, it is also useful to be able to state that every choice assignment must assign IN to at least one alternative from every set of exclusive alternatives. This means that it's quite possible that the time map contains no "consistent" description of the world. Things become significantly more difficult once we allow arbitrary constraints on what constitutes a "consistent" description of the world. How do you recognize when all of your choices have gone bad? Answering this raises a number of complications that will b: explored in 4.6.3. Before we get to that, I want to describe the RMS update algorithm for handling gating objects.

4.6.1 Updating data dependency networks containing gating objects

This section presents an algorithm for incrementally updating data dependency networks containing gating objects. The algorithm was chosen for clarity, not optimal performance, and a number of improvements are incorporated in the actual implementation. Since most of these are obvious and uninteresting, they won't be discussed here.

In the Doyle system, a ddnode label was either IN or OUT. Now we're considering a system in which labels are boolean combinations of gating objects. It is worth considering some properties of the range of labels. First, we assume that at all times there are a finite number of gating objects (not too hard in any real system). A *satisfiable* label is just one for which there exists an assignment under which the label evaluates to IN. The set of all possible satisfiable labels (ignoring equivalent forms) together with the operations of conjunction and disjunction form a finite lattice. The elements of the lattice can be partially ordered by the (transitive and reflexive) relation of *subsumption*. We say that "L1 subsumes L2" where L1 and L2 are labels, if L2 evaluates to IN under all assignments that L1 evaluates to IN (there could be assignments under which L2 evaluates to IN but L1 evaluates to OUT). Given two justifications, J1 and J2, consisting only of gating objects, *J1 is said to subsume J2 iff the set of in-justifiers of J2 is a subset of the set of in-justifiers* of J1 and likewise for the sets of out-justifiers. So, for example, ({n1,n2}{n3}) subsumes ({n1}{}) whereas ({n2}{n3}) does not. A justification J1 subsumes a label or disjunction of justifications, L1, iff it's not the case that there exists an assignment under which all of J1's in-justifiers are IN and all its out-justifiers are OUT and L1 evaluates to OUT. One can determine if such an assignment exists by making the minimal partial assignment that forces J1 to be IN and then proceeding by the *reductio* method [Hughes 68] (also called semantic tableaux method) to try to make L1 OUT. Notice that it's not sufficient to simply check that there exists a justification, J2, in L1 such that J1 subsumes J2. If J1 is ({A}{}) and L1 is (({A C}{}) ({A}{C})), then J1 subsumes neither of the justifications in L1, but it's clear that J1 evaluates to IN whenever L1 evaluates to IN. Now we can extend this notion to a relation on labels (disjunctions of justifications) by saying that a label L1 subsumes a second label L2 iff each justification (disjunct), J1, in L1 subsumes L2. We can consider the label IN as the justification ({}{}) with the empty set for both in-justifiers and out-justifiers. The label OUT is simply (). Any label subsumes IN, and OUT subsumes any label. IN and OUT are the extremal values in the lattice. We say that a label becomes "more IN", if it is given a new label such that its old label subsumes the new but the new label does not subsume the old (for "more OUT" reverse old and new). It should be clear that a label can only become more IN (OUT) a finite number of times without becoming more OUT (IN)[17]. If there are no gating objects, then the lattice reduces to {IN,OUT}. The system is designed not to inflict a cost on those who have no need of gating objects.

[17] It's also possible that a label can change (the label now evaluates to IN under a different set of assignments than it did before) without becoming either more IN or more OUT. The elements of the lattice are only partially ordered. However, you can transform any label into any other by a finite number of changes each consisting of making a label either more IN or more OUT.

The objective of the update algorithm is to add or remove a justification `j` from a node `n`, and recompute the labels of all possibly affected nodes. Every node `n` has a set of *justificands*. A node `m` is a justificand of a node `n` if `m` has at least one justification in which `n` appears. A label is said to be a *legal label* if it has the form of a list of justifications such that all ddnodes appearing in those justifications are gating objects. As was mentioned earlier, this form can be interpreted as a boolean formula in disjunctive normal form (DNF). Every ddnode has two fields, both of which are legal labels. These are called simply the *label* and *old-label* of the node. The old-label is used by the system to determine the extent of the change due to the latest modification to the net. This information is used in conjunction with signal functions to respond to specific changes in a nodes status. The set of nodes possibly affected by a change in the justifications of `n` consists of `n` and (recursively) all the nodes possibly affected by a change to the justificands of `n`. The update algorithm also tries to simplify labels using a variety of techniques[18].

The update algorithm is divided into four basic stages:

1. find all nodes possibly affected by the current change and mark them
2. recompute the labels of all the nodes found in the previous step
3. filter all the newly recomputed labels to eliminate disjuncts (justifications) which are disallowed by the current set of nogoods
4. execute the signal function of each node (found in 1) whose label has changed

Having made changes to node `n` (added or removed justifications), step one is carried out by applying to `n` the function `mark-and-initialize` defined in Figure 4.21. This function also computes the set of nodes possibly affected by the change (*i.e.*, all those visited).

The second step is carried out by applying the function `compute-new-label` (again in Figure 4.21) to each node visited in the previous step. The easiest way to see that this terminates is to note that the algorithm continues only as long as the labels continue to become more IN (the newly created label subsumes the current label). Since this can only happen a finite number of times and no label ever becomes more OUT during this stage,

[18] If the label contains a disjunct that is just IN (*i.e.*, `({}{})`) or the label is a tautology (e.g. `(({A}{})({}{A})))` then replace it with `(({}{}))`. More generally, if the label contains two disjuncts, D1 and D2, such that D1 subsumes D2, eliminate D1. If the label contains a contradiction (*e.g.*, `(({A}{A})))` then replace it with `()`.

```
mark-and-initialize(n)
 1. set the old-label of n to be the label of n
 2. set the label of n to be OUT (the empty disjunction)
 3. place n on the list of visited nodes
 4. for each justificand m of n that has not been visited
      mark-and-initialize(m)

compute-new-label(n)
 1. create a new label as follows
   a. substitute for each non gating object m in the justifications of
     n the label of m
   b. convert the new label to DNF and simplify
 2. if the current label subsumes this new label but not vice versa
   then for each justificand m of n compute-new-label(m)
```

Figure 4.21: Functions for updating dependency networks

the algorithm must terminate.

The result is that every node now has a legal label. It should be clear that this label is consistent. The computation guarantees that the label of a node n follows from the label of the nodes in the justifications of n. To see that each label is well-founded, it is sufficient to see that, (a) following mark-and-initialize all labels are well-founded (they're all OUT) and, (b) that each substitution made by compute-new-label must therefore result in a well-founded label.

To understand how the third step of the overall update algorithm is carried out, it is necessary to understand how nogoods are implemented. As was mentioned earlier, all nogoods are conjunctions of gating objects. This allows us to represent a nogood as a sorted vector of gating objects. Each gating object has a pointer to all those nogoods that it is the first element of. Filtering is handled by examining each justification (disjunct) in a label and eliminating those whose set of in-justifiers is a superset of some known nogood. Only the nogoods referenced from the gating objects in the set of in-justifiers need be tried.

Finally, we have to determine what signal functions to call. One criteria might be simply that the labels are not equivalent (*i.e.*, they do not mutually subsume one another). However, more sophisticated criteria are often necessary. A common thing in the time map is to notice when a ddnode, say corresponding to a constraint or time token, becomes more IN or more OUT. Another change worth noticing involves a ddnode whose label is OUT under all assignments to gating objects that satisfy some subassignment. So, for instance, you might want to notice if a token is OUT, given that some other token is OUT. This is useful when you want to detect that an action is no longer warranted for preventing some event. Suppose that you justify paying a parking ticket on the basis that had you not paid, the city would impound your car or do something equally nasty. You plan to pay the ticket, but you base this decision on the assumption that, if you don't (*i.e.*, the gating object corresponding to the decision to pay is OUT), then the city will do something nasty. If the token corresponding to the city's potential for malevolence is ever OUT, given that the token corresponding to your paying is OUT (*i.e.*, the gating object corresponding to your decision to pay is assigned OUT), then you're wasting your money. Setting up a signal function that fires under just these conditions is relatively easy to do.

4.6.2 Modification to the TMM to handle gating objects

There are a number of modifications that we have to make to the TMM algorithms discussed in the preceding sections before they will work with gating objects. Every part of the algorithm that has to take into account the labels of ddnodes is affected. Constraint propagation, determining point-to-point distances, and caching routines all have to interpret the labels of ddnodes corresponding to constraints. Almost everywhere that the algorithms discussed in previous sections stated "becomes IN", or "becomes OUT", we have to replace with "becomes more IN", or "becomes more OUT". The top level invariant maintained by the temporal reason maintenance described in Section 4.3.5 system can be rephrased as:

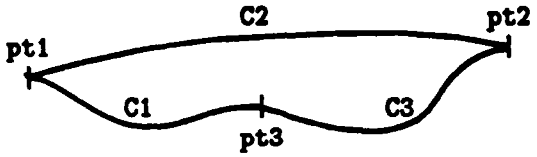
> Each token, T1, can be shown to clip each contradictory token, T2, under just those assignments (to gating objects seen as boolean variables) such that the labels of T1 and T2 evaluate to IN, and there exists a path through the time map such that the beginning of T2 can be shown to precede the beginning of T1, and the labels of each `CONLINK` in the path establishing the ordering relation also evaluate to IN.

All these additional complications are not worth a great deal of effort to explain. The proof of correctness is essentially the same except slightly more wordy. Caching requires some additional bookkeeping, but the main ideas still hold. There are, however, a couple of moderately high level ideas that are worth some discussion. In particular, I would like to communicate to the reader something of the flavor of interpreting time maps containing gating objects. In order to do so, it will be necessary to understand some of the rudiments of how constraint propagation and search works with gating objects.

To begin with, the notion of a path through the time map has become more complicated. Now every path has a label computed from the labels of the ddnodes in the set of supporting ddnodes for the path. That label determines which PWDs the upper and lower bounds of the path are believed in. Whether or not you can extend a given path (during constraint propagating or finding point-to-point distance estimates) using a particular directed edge (CONLINK) depends upon the current set of nogoods and the labels of the path and the ddnode corresponding to the directed edge. I'm not going to provide details for any of this. Most of it is just bookkeeping and graph search. I do want to illustrate some of the repercussions, however. In particular, the ddnodes corresponding to point-to-point relations will be IN under certain assignments to gating objects, and OUT under other assignments. This means, for example, that in one PWD a persistence might endure indefinitely (the upper bound of the persistence is *pos-inf*), while in another PWD it is clipped by a contradictory ddnode.

The invariant associated with TCONDITs and constraint propagation discussed in Section 4.3.3 was defined purely in terms of IN and OUT. Taking gating objects into account, the invariant could be restated as: each ddnode corresponding to a relationship between a pair of points has a label that evaluates to IN under just those assignments to gating objects where the labels of constraints in the time map warrant it. An example should illustrate the important points. Figure 4.22 shows a time map with three points, three constraints, and one point-to-point relation, (pt< pt3 pt2) (A and B are gating objects). The path from pt3 to pt2 using constraints C1 and C2 has a lower bound of 1 and an upper bound of 4. Since the lower bound is greater than or equal to zero, this means that (pt< pt3 pt2) is satisfied. The justification ({n1,n2}{}) is added to n4, and the node's label is updated using the RMS to reflect the new conditions for belief. There is another path from pt3 to pt2 using just C3 which also satisfies (pt< pt3 pt2). In this case, the justification is ({n3}{}), and as before the RMS performs the labeling. The label of n4 can be interpreted as saying that pt3 is believed to precede pt2 just in case either A or B is chosen. This sort of interpretation, suitably extended to deal with conjunctive queries, is employed by the time map query routines.

Now, let's suppose that pt3 is the beginning of token1, and pt2 is the beginning of token2 where token1 and token2 are contradictory (see Figure 4.23). Suppose that n4



```
ddnode:                                       label:
 n2 --> C1 = (elt (distance pt1 pt3) 1 3)      (({A}{}))
 n1 --> C2 = (elt (distance pt1 pt2) 4 5)      (({}{}))
 n3 --> C3 = (elt (distance pt3 pt2) 2 3)      (({B}{}))
 n4 --> (pt< pt3  pt2)                         (({A}{})({B}{}))
```

Figure 4.22: Monitoring point-to-point relations

```
token1  P
                 ||----------------?????????????????????????
                 pt3
token2  (not P)
                                   ||---------------------->
                                   pt2
```

Figure 4.23: Clipping constraint with gating objects

justifies the clipping constraint (elt (distance (end token1) (begin token2)) *pos-tiny* *pos-inf*) (*i.e.*, n4 determines whether or not token1 and token2 are apparently contradictory). Now given the additional constraints from Figure 4.22, the time map in Figure 4.23 can be interpreted as saying, if either A or B are chosen, then token1 precedes token2 and the latter clips the former. If neither A nor B are chosen, then token1 persists indefinitely.

There are any number of other details that have to be considered in designing reason maintenance systems of the sort described here. Eventually, I hope that someone will write a book detailing the functional and computational tradeoffs implicit in the various alternatives RMS designs available. This is not the place for such a discussion. As for the details of the TMM algorithms, any more detailed exposition would probably require publishing the code (suitably cleaned up). Unfortunately, this is not a viable option, given that this code runs over 6,000 lines, and requires intimate knowledge of the workings of the underlying deductive retrieval system. Most of the issues I have overlooked in the dissertation have straightforward solutions that should be familiar to most computer scientists. There is one further issue, however, that I wish to bring up. This concerns the problem of determining when you've run out of choices. The following section concerns certain inadequacies of the Doyle and deKleer type systems with regard to the task of reasoning about choices. I'll demonstrate why the propositional deduction system of McAllester [McAllester 80] is better suited for performing the requisite reasoning task.

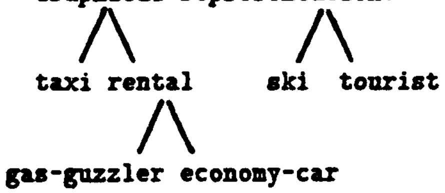
4.6.3 Running out of choices

One problem that arises in reasoning about alternatives concerns determining when an alternative is still viable. In the absence of gating objects a propositional object was either IN or OUT. If a ddnode corresponding to the assertion that a plan was working became OUT, then the planner had to correct the plan to ensure that the associated task was achieved. Now we have the situation where a plan may work under certain sets of alternatives, but not under others.

Consider two alternatives with corresponding gating objects A and B. Suppose that A justifies the expansion of a plan PL$_1$, and that there is a ddnode n_1, corresponding to (acceptable-plan-for PL$_1$ task31), which depends upon A and the fact Q is true throughout task31. Now suppose that B justifies the expansion of another plan PL$_2$, which results in (not P) and thereby undermines the protection of Q throughout task31. At the time when the effect (not P) is added, n_1 will become more OUT. Its label might include a justification of the form ({A}{B}). This can be interpreted as saying that PL$_1$ is an acceptable plan for task31 just in case the alternative A is chosen and B is not. The plan PL$_1$ may be acceptable in other circumstances, so this is not sufficient grounds for eliminating either of the options A or B. Of course, there are situations in which an option can be totally ruled out on the basis of alternatives available at the moment. Suppose that A$_1$ and A$_2$ are the alternatives for task1 and B$_1$ and B$_2$ are the alternatives for task2. If I determine that A$_1$ and B$_1$ result in an unsatisfactory state of affairs, and similarly for A$_1$ and B$_2$, then I have effectively ruled out A$_1$. Determining this in general is not easy, and responding effectively is even more difficult.

First of all, let's consider what has to be represented. As an example to make the discussion concrete, suppose that you are planning your itinerary for an upcoming conference. To begin with, a planner has to be able to describe its alternatives. For instance the choice,

Graphical representation:



```
==>  Clause representation:
      (or taxi rental)
      (or (not taxi) (not rental))
      (or ski tourist)
      (or (not ski) (not tourist))
      (or (not rental) gas-guzzler economy-car)
      (or (not gas-guzzler) (not economy-car))

Nogoods:
 (or (not ski) (not gas-guzzler))
 (or (not ski) (not economy-car))
 (or (not taxi) (not ski))

Deduced unit clause:
 (or (not ski))
 (or tourist)
```

Figure 4.24: Simple decision tree

"you can either take a taxi or rent a car to get from the airport to the convention center" might be notated as the disjunctive clause (or taxi rental) in which taxi, and rental are boolean variables. Alternatives are also often dependent upon one another. So, "if you rent a car, then you can rent either a big roomy gas guzzler or something more economical" could be handled with (or (not rental) guzzler economy-car). To state that the alternatives are mutually exclusive we add the clauses (or (not taxi) (not rental)) and (or (not guzzler) (not economy-car)). Finally, it will be necessary to state that certain sets of alternatives are unacceptable (*i.e.*, nogood). Let's make our example a little more interesting. Suppose that you are also either going to do some sight-seeing or ski on your trip (*i.e.*, (or ski tourist)). During planning you realize that you'll need transportation to travel to the ski areas so you can't both take a taxi and ski. This would be represented as (not (and ski taxi)), or in disjunctive form as (or (not ski) (not taxi)). Also, you can't afford to both rent the luxury car and ski, and, being the perfect yuppie, you're ashamed to be seen at a fancy ski resort in an economy car (*i.e.*, (or (not ski) (not gas-guzzler)) and (or (not ski) (not economy-car)) respectively). The complete situation is presented in Figure 4.24.

Notice that in this case you should be able to deduce that, at least according to the alternatives explored so far, you won't be skiing on this trip. If you take the clauses in Figure

4.24 at face value perhaps, there's no cause for alarm. After all, you had two alternative plans for enjoying yourself on this trip and only one of them appears to have failed: you can still go sight-seeing. Nevertheless, it seems that you would probably want to think a bit more about going skiing to see if you could save the plan (perhaps you could go in with some other young professional and rent a more suitable vehicle). The problem is a familiar one in planning; what to notice and how to respond. The ability to reason about alternatives does not help in the least in this respect. In fact, there's more to reason about and it's harder to determine when things are not working out. Even if the planner's rule base is guaranteed to contain the components for a solution to any problem selected, you can't always guarantee that the alternatives you choose to explore will provide the necessary components. You can't expect to reason about all possible alternatives simultaneously; there are just too many. The upshot of this is that you still have to be able to detect when things are failing.

Handling the sort of deduction required to determine that the skiing alternative in Figure 4.24 was in trouble is not easy in the Doyle or deKleer type reason maintenance systems without considerable augmentation. We're really trying to compute an assignment to boolean variables satisfying an arbitrary set of boolean constraints. The general task is intractable, but since we're assuming a small set of variables (gating objects), the potential exponentiality shouldn't concern us. In order to get the necessary functionality, the TMM employs some techniques used by David McAllester in his deductive system [McAllester 80]. In McAllester's system, propositions (in our case gating objects corresponding to alternatives) are either `true`, `false`, or `unknown`. The system performs propositional deduction using an internal clause form and a reductio method to resolve contradictions. McAllester's system uses invariants to assist the user in coordinating its computations with those of the deductive machinery. The TMM employs a high level invariant that sees to it that unknowns (alternatives which are not necessarily chosen) are forced to be false in order to ensure completeness of the propositional deduction algorithm. If assigning an unknown to be false results in a contradiction, the system retracts the assignment and tries another. By carefully adding additional clauses in the course of resolving conflicts, the system guarantees that if a satisfying assignment exists, the algorithm will find it, and if the constraints are inconsistent, the algorithm will detect this also. If the constraints are consistent, then the system will satisfy all clauses and detect when an alternative is forced one way or another (it does this by deducing unit clauses (*e.g.*, `(or (not ski))`). Signal functions can be attached to nodes corresponding to an alternative in order to notify the user if that alternative must necessarily be chosen or rejected. As I mentioned, the system computes whether or not a boolean formula in DNF is satisfiable. I am assuming that the number of alternatives is relatively small. The variant of McAllester's techniques used in the TMM seems to perform quite well on the sort of input expected from a planner, even on examples involving upwards of 100 boolean variables. It's not likely that the TMM could handle a time map of any complexity involving 100 gating objects. So far, the TMM appears to work reasonably well on time maps involving a dozen or so gating objects at a time. The techniques described here were not meant, and are not expected, to perform well with a large number of outstanding alternatives. Splitting the world (introducing addiadozenating objects) should be done only in situations where the planner has some reason to believe that it will pay off. Admittedly, determining the potential for such a payoff will be difficult. All I can say is that humans appear to be quite good at it, and eventually our programs will have to be also.

4.7 Loose ends

There are any number of issues that have not been discussed in this chapter, but that probably deserve some mention. In this section, I will talk about two issues that have concerned me during the design of the TMM, but for one reason or another were not addressed in the final implementation. The first deals with a sort of nonmonotonic inference that we have ignored so far, but that has been found to be useful in certain situations. How should the system handle queries of the form `(tt pt1 pt2 P)` in which it is not likely that there will be mention of tokens of type `P`, but there may be any number of tokens of type `(not P)`? In some circumstances, it might be useful for a query to succeed just in case there is no reason to believe it will fail. That is to say, the query should succeed just in case there is no subinterval `pt3` to `pt4` during the interval `pt1` to `pt2` such that `(tt pt3 pt4 (not P))`. This turns out to be difficult primarily because you are forced to quantify over all possible contradictory tokens. The second issue concerns the fact that there can be two time tokens, both asserting the same fact type, say `P`, whose corresponding intervals overlap, but the TMM cannot deduce the truth of `P` throughout the union of the corresponding intervals. In the following I will explain why these issues are worth worrying about and suggest some possible approaches to dealing with them.

4.7.1 Anti-protections

Suppose that we're in a domain where an object can have any number of other objects sitting on it (unlike simpler versions of the blocks world). Now consider how one would handle the query `(tt pt1 pt2 (clear table38))`. It would be asking too much for the data base to have explicit tokens indicating just those intervals in which `(clear table38)` is true. This might be reasonable in a totally ordered time map[19], but in a partially ordered time map things are considerably more difficult. Instead, we'll engineer it so that `(tt pt1 pt2 (clear table38))` succeeds just in case there are no tokens asserting something of the form `(on ?some-object table38)` such that the intersection of that token's corresponding interval and the interval pt1 and pt2 is nonempty. This is just a specific instance of a more general form of nonmonotonic inference described by the following rule:

```
(<- (M (tt ?begin ?end ?q))
    (forall (tok)
        (if (time-token (not ?q) tok)
            (or (pt< (end tok) ?begin)
                (pt< ?end (begin tok))))))
```

The query `(tt pt1 pt2 (clear table38))` is equivalent to `(M (tt pt1 pt2 (not (exists (x) (on x table38)))))`. Of course, the above rule is not a standard backward chaining rule. I think, however, that it is quite clear what is expected with regard to the quantification in the antecedent of the above rule. Handling queries involving the above rule is somewhat complicated, but it introduces no insurmountable problems. For the query `(tt pt1 pt2 (clear table38))`, it is necessary to ensure that no token contradicting `(clear table38)` intersects the interval pt1 to pt2. Setting up the correct protections is a bit more complex. In a standard protection of the sort discussed in Section 4.3, all we have to find is *some* time token asserting the protection schema such that its corresponding interval satisfies the necessary temporal constraints. In ensuring `(clear table38)` throughout an interval, we have to make sure that *all* tokens contradicting `(clear table38)` satisfy the necessary temporal constraints. The universal quantification requires that we set up dependency structures for every contradictory time token, not only those that appear in the time map at the time the query is processed, but any additional contradictory tokens that are subsequently added to the time map. The resulting data dependency structure is referred to as an *anti-protection*. Figure 4.25 shows how an anti-protection might be set up during

[19] For each point in the total order you could just keep track of the difference between the number of actions (preceding the point) which place objects on `table38` and the number of actions (preceding the point) that remove objects from `table38`. At each point where the difference is 0 you can conclude `(clear table38)`.

backward chaining.

Notice that in the time map of Figure 4.25 the anti-protection is OUT even though the query will succeed. The ddnodes corresponding to anti-protections generally appear as out-justifiers in justifications. Note also that in order to maintain this anti-protection, every time a token contradicting `(clear table38)` is added to the time map a new justification has to be added to the anti-protection ddnode. The current version of the TMM implements anti-protec'ions, but the routines are still very much experimental.

It is an interesting exercise to consider how this functionality might be incorporated into overlap chaining. The form `(->t (and` P_1 ... P_i `(M (not` P_{i+1}`))` ... `(M (not` P_j`))) Q)` can be interpreted as saying that `Q` is believed wherever P_1 through P_i are believed and P_{i+1} through P_j are not. Notice that this bears considerable resemblance to a ddnode justification with in-justifiers and out-justifiers. To strengthen this connection, I'll call P_1 through P_i ***in-types*** and P_{i+1} through P_j ***out-types***. An overlap chaining rule with out-types will be referred to as an ***augmented chaining rule*** or ACR.

All temporal forward chaining rules potentially result in the generation of new time tokens. In ACRs this proliferation can be extreme. Let's consider the set of tokens resulting from an ACR. Some additional terminology should help. A ***generating set*** for an ACR is a set of tokens: exactly one of each type in the in-types of the ACR[20]. Each generating set designates an interval of time corresponding to the intersection of the intervals of its component tokens. The set of all tokens of a type in the out-types of the ACR is called the ***sieve set*** for the ACR. The motivation for the terminology is that the generating sets provide a set of candidate intervals that the sieve set restricts or filters, thereby defining a set of actual tokens. The sieve set can be thought of as punching holes in the intervals provided by the generating sets. The resultant tokens are constructed from the intersection of the union of all intervals specified by generating sets with the complement of all intervals associated with tokens in the sieve set.

This is much easier to visualize. The example in Figure 4.26 illustrates the tokens spawned by a specific set of initial tokens and a single ACR. By convention, we associate a fixed set of identifiable tokens with each generating set. These consist of one token whose endpoint is identified with the least upper bound of the points ending tokens in the

[20] This treatment is admittedly a bit glib. There are a number of complications that arise in considering universally quantified variables in ACRs. These complications are easily dealt with and are ignored to simplify the discussion.

```
Frame of reference: (begin task1) Scale: 1.0

token1 (on salt-shaker table38)
 ||------------------|
token2 (on pepper-mill table38)
                                        ||------------------------>
task1 (achieve (clear table38))
                        |-------------|
```

Initial time map

```
(for-first-answer
    (fetch '(tt (begin task1) (end task1) (clear table38)))
    (add '(plan-for task1 noop)))
```

Controlled forward inference

```
Ddnode:  Data type:  Corresponding datum:
 n1        TOKEN       (time-token (on salt-shaker table38) token1)
 n2        TOKEN       (time-token (on pepper-mill table38) token2)
 n3        TOKEN       (time-token (achieve (clear table38)) task1)
 n4        TCONDIT     (pt< (begin token1) pt2)
 n5        TCONDIT     (pt< (end token1) pt1)
 n6        TCONDIT     (pt< (begin token2) pt2)
 n7        TCONDIT     (pt< (end token2) pt1)
 n8        PROP        (anti-protection pt1 pt2 (clear table38))
 n9        PROP        (plan-for task1 noop)

    n8 has the justifications: (({n1, n4}{n5}) ({n2, n6}{n7}))
    n9 has the justifications: (({}{n8}))
```

Resulting data dependencies

Figure 4.25: Dependency relations for anti-protections

```
|----------------------------| P1
                                   |-----------|  P2
     |-------------------------------------------| Q1
                         |--------------| R1
           |------|  R2
     |-----|   S1
                  |------| S2
                             † S3
           † S4
                  † S5
                                        |------| S6
```

```
Note:  Tokens generated but not currently believed ( i.e.,  IN)
       are indicated by a † marking their endpoint.
```

Figure 4.26: Tokens resulting from `(->t (and P Q) (M (not R)) S)`

generating set (relative to the ordering relation on points), and one each identified with the beginning of tokens in the sieve set. Consider what would follow from the addition of the ACR `(->t (and P Q (M (not R))) S)` to a time map consisting of the tokens P1, P2, Q1, R1, and R2. The result, shown in Figure 4.26, would include the set of tokens {Si : 1 ≤ i ≤ 6} of which only S1, S2, and S6 are currently IN. The current sieve set is {R1,R2}. There are two generating sets {P1,Q1} and {P2,Q1}. with resultant token sets {S1,S2,S3} and {S4,S5,S6} respectively.

ACRs are not difficult to understand functionally. Their implementation is straightforward using the techniques described in this dissertation. I believe that with a little effort such an implementation could be made reasonably efficient. The sort of nonmonotonic inference supported by ACRs and anti-protections will probably find considerable use in reasoning about time.

4.7.2 Overlapping time tokens

Let's consider a robot messenger in a large factory. Suppose that this robot employs a vision system for navigation and obstacle avoidance, and hence it requires illumination in order to cross an open area without getting into trouble. Now let's suppose the rcbot is currently contemplating crossing `area35`. There are two lights, `light1` and `light2`, in `area35`, either

```
Frame of reference: (begin task1) Scale: 1.0

token1 (on light1 area35)
   ||------------------------------|
token2 (illuminated area35)
   ||------------------------------|
token3 (on light2 area35)
                       ||------------------------------|
token4 (illuminated area35)
                       ||------------------------------|
task1 (traverse area35)
          |--------------------------------------|
```

Figure 4.27: Handling overlapping intervals

one of which is capable of illuminating area35 sufficiently to enable the robot's successful traversal. To be able to deduce that an area is illuminated whenever a light is on in that area we have the following overlap chaining rule:

```
  (->t (and (on ?appliance ?location)
            (instance-of ?appliance light))
       (illuminated ?location))
```

The problem is that neither light is on sufficiently long to allow the robot to complete the crossing. Figure 4.27 displays a time map describing the situation.

It would seem that the query (tt (begin task1) (end task1) (illuminated area35)) should succeed in the time map of Figure 4.27. Of course, it won't because there is no (single) time token that satisfies the protection criteria. The TMM will fail in this situation to provide the right sort of assistance.

There are a number of ways that this deficiency might be corrected. One approach is to define the ending point of a token dynamically. I'll speak of a *terminator* of a token to distinguish a dynamically chosen end point from the (unique) end point associated with the actual time token data structure (*i.e.*, the end slot of an object of data type TOKEN). A token T_1 is said to be *subordinate* to a second token T_2 asserting the same fact type if T_2 begins before T_1 and the end point of T_2 cannot be shown to precede the beginning of T_1. A point pt_0 is said to be a terminator for a token T_1 if either pt_0 is the end point of T_1 or there exists a second token T_2 such that T_2 is subordinate to T_1 and pt_0 is a terminator for T_2. In Figure 4.27, one terminator of token token2 would be defined to be the same as the end of token4. We could add some restrictions to eliminate certain other terminators so that, for example, the end of token2 would not be a terminator for token2. However, in a partially ordered time map, time tokens will not in general have unique terminators. Processing queries would proceed as usual, except for the fact that several terminators will have to be considered instead of just one end point. Setting up the necessary data dependencies is a little more complicated than before, but presents no special problems. As with the anti-protections, the issue here is the added computational cost of building this permanently into the search routines. It's not apparent that it will be used frequently enough to override the cost of maintaining the necessary data structures (in the case where the set of terminators for each token is maintained) or extending the search for every query (in the situation where the candidate terminators have to be recomputed for each query). Again, this will require further investigation to determine if this capability represents a critical functionality.

4.8 Summary

This chapter represents a compendium of techniques and issues involved in managing temporal data bases. The basic functionality behind reasoning about partial orders and metric constraints was addressed in terms of a set of routines for processing temporal queries. These routines depend upon techniques for temporally indexing time tokens and determining the best estimates of point-to-point distances. Two specific methods for optimizing these operations were discussed. The first involved caching certain point-to-point distance estimates. The cached values were used to speed heuristic search for the best path(s) between pairs of points in the time map. A best path was defined to be a set of constraint links leading from one point to another such that the sum of the lower (upper) bounds was maximal (minimal) over all other such paths. The second method of optimizing temporal queries involved preprocessing queries. This allows the system to exploit certain capabilities of the search routines used to derive point-to-point distance estimates. In particular, these search routines can (generally) find the best estimates between a destination point and each member of a set of target points far quicker if all the target points are specified at once, than if they're given individually.

The problems surrounding the treatment of defeasible predictions was addressed in terms of a temporal reason maintenance system and methods for detecting and responding to changes in temporalized beliefs. A proof of correctness for a temporal data dependency update algorithm was provided along with details of the implementation. The algorithms for temporal reason maintenance are one of the main computational contributions of this work, and a fair amount of space was devoted to its explication. The interplay between resolving apparent contradictions and keeping track of the status of protections is crucial here, and it was necessary to introduce some conventions for using the TMM which guaranteed that the algorithms terminate. The main restriction involved care in separating temporal connectivity from certain sorts of inferential connectivity. It was argued that this separation does not constitute a major inconvenience, and that in many instances it makes it clear what's actually going on in the time map.

This chapter also dealt with the details of managing certain types of temporal forward chaining. Both logical or overlap forward chaining and a method for performing restricted envisionment were discussed. The latter, called auto-projection, was shown to be quite similar to controlled forward chaining, except that such rules were not able to introduce new constraints to the existing partial order on their own initiative.

Finally, a number of deficiencies in the current TMM were discussed along with suggestions about how they might be corrected, and warnings that they might involve considerable computation.

Chapter 5

Applications in planning

5.1 Introduction

Planning is a synthetic process. You don't begin with a plan and then prove that it is correct. Rather, you begin with a bunch of goals that you wish to achieve, and then you proceed to make guesses about how you might go about achieving them. The sort of inferences involved in planning can be divided into two classes: deductive and abductive. Deduction is used to verify that some proposed plan is likely to succeed. Abduction is used to make guesses about how to proceed in constructing a plan. In general, you begin with an intention: something that you are committed to making true. Once you are clear about what you want to achieve, you set about trying to formulate a set of steps to bring about the desired state of affairs. In the course of formulating a course of action to achieve some desired state, a planner is likely to discover that the actions he is considering conflict either with one another, or with the planner's expectations concerning other events. For instance, I had thought there would be enough time to finish a proposal by 5:00 PM, but this afternoon was the only time I could schedule an appointment with the dentist, so the proposal will have to be delayed. The process of exploring the repercussions of various courses of action by proposing actions and then responding to noticed conflicts is what I will be referring to as planning in this chapter.

A *task* is a description of an intention. A task stipulates a process to be carried out by the planner. In the time map, a task is represented as a time token whose corresponding interval determines the period of time during which the process is performed. The type of the token indicates what sort of process is involved (*e.g.*, `(achieve (on block34 table17))`). A task has associated with it a number of constraints: constraints that specify the current estimate of the duration of the task, and constraints on when the task begins and ends (deadlines).

Some tasks are used to indicate actions that require no further elaboration given the level of detail expected in a plan. These actions and the tasks corresponding to carrying out such actions are referred to as *primitive*. Other tasks, referred to as *problematic*, require further elaboration (or refinement) before they can be said to be adequately specified. The distinction between primitive and problematic tasks is purely pragmatic. If you have a routine for carrying out a task that is almost guaranteed to bring about some desired state of affairs, and the steps in the routine are invariably carried out in the same way , then that routine is a good candidate for associating with a primitive task. For example, a robot might have a foolproof method for placing one block on another given that both blocks are clear. In that case, `(puton block34 table17)` might be considered as primitive. The task `(achieve (on block34 table17))`, on the other hand, might be considered problematic given that there are a number of methods for bringing about `(on block34 table17)` each of which is appropriate in different circumstances. You might argue that the only sorts of tasks that should be considered primitive for a robot correspond to applying power to motors. You can't extend your arm if you're tied up, but if you can do anything at all, then you can probably flex the muscle in your arm. This criterion is too strict for some reasoning tasks, and not strict enough for others. After all, pulsing a stepper motor can be just as complex and unpredictable at the level of circuit interactions as planning out the day's activities. In many cases, however, it would be exorbitantly expensive and likely a waste of time to plan down to the level of pulsing stepper motors. As was mentioned in the first chapter, the sort of problem solving I am interested in involves anticipating certain types of foreseeable problems. The problems that one is likely to encounter in moving an appendage are better handled at the time they occur, not by carefully planning things out in advance. The distinction between primitive and problematic tasks should be based on whether or not the detailed explication of a task is likely to assist in reliably predicting potential interations between tasks. If the only time at which sufficient information is available for detecting such interactions is at run time, then it is better to suppress the detail until execution time and consider the task as primitive.

The sort of planning we will be looking at in this chapter I refer to as *reductionist* planning. A reductionist planner is one that breaks down (or reduces) a problematic task into a set of steps, or subtasks, that can be considered in relative isolation. That such reductions can result in practical plans depends upon the assumption that our knowledge of planning can be decomposed into small self-contained units, not unlike program subroutines. Since these units have a relatively wide scope of application, they often have to be tailored for a specific situation. This might be done by ordering certain steps so that they don't interfere with one another, or by carefully choosing a method for accomplishing a step that does not introduce side effects that conflict with other tasks being considered. These interferences and conflicts are commonly called *interactions*, and a good deal of effort has gone into explicating strategies for eliminating or *resolving* them [Wilensky 83] [Pednault 85] [Chapman 85]. It is generally assumed that interactions between steps can be resolved locally. By this, we simply mean that the decision process involved does not have to take into account all of the current tasks. Later, we'll see occasions in which this assumption breaks down in attempting to handle deadlines and resources.

A plan is simply a partially ordered set of tasks. A *complete*, or fully specified, plan is one in which all unreduced steps are primitive. There is no generally agreed-upon notion of what it means to have a "good" or even a "correct" plan. But it is reasonable to expect that an "acceptable" plan is complete, satisfies the initial specifications (*e.g.*, deadlines or resource usage limitations), and has no unresolved interactions.

Many interesting planning problems can be formulated in terms of coordinating processes. The processes may be under the control of a single agent, or they may have external origin, including natural phenomena. Planning proceeds by hypothesizing a set of steps that might bring about the desired state of affairs. This is in some respects similar to the sort of inference commonly carried out in diagnosis. A diagnostician wants to formulate a set of processes that fit with the observed facts and at the same time serve to explain how a given phenomenon came about. Analogous to planning, explanations for two different phenomena may conflict with one another. The hypothesis that a diode is shorted may explain the high current in a circuit, but contradicts the fact the diode appears to be clipping an AC signal. In an effort to explain the high current, the diagnostician had to actually entertain the possibility that the diode was shorted, making assertions, projecting consequences, and detecting and resolving possible conflicts. in order to test out the hypothesis.

Planning consists of much the same sort of process. A planner wants to formulate a set of actions, coordinated with observed and predicted events, such that the planner's intentions are realized. Suppose that we have a task in the blocks world domain to clear a given block block47. This would be represented in the time map using a token of type (achieve (clear block47)). This token corresponds to saying (perhaps a bit optimistically) that the planner will perform whatever actions are necessary in order to achieve the effect of (on block47 table). Not surprisingly, the planner wants to know just what actions it will be performing. One of the rules in the data base may have the following form:

```
(if (and (tt (begin ?tsk) (end  ?tsk) (on ?y ?x))
         (time-token (achieve (on ?y table)) ?subtsk)
         (pt= (end ?subtsk) (end ?tsk)))
    (time-token (achieve (clear ?x)) ?tsk))
```

This rule essentially says that, if there is something ?y on the block ?x throughout the interval in which (clear ?x) is to be achieved, then one way in which it might actually have occurred is if another task achieving (on ?y table) occurred whose end point is coincident with the task to achieve (clear ?x). The antecedent of the rule actually constitutes a plan for achieving (clear ?x). We started with the consequent, and we wanted to determine how it might come to pass. The inference fits the form of Pople's abduction schema [Pople 73]. The antecedent of a rule used for making an abductive inference can be divided into things that must be true and things that we are willing to grant for the sake of entertaining a particular hypothesis. If everything must be true, then the rule can be used only deductively. In the rule for clearing a block it must be true that the block already has another block on it, but we're willing to postulate an additional task to place the offending block on the table. If it is ever the case that the necessary conditions are no longer satisfied, then the inference is no longer warranted. It is also possible, for one reason or another, that the antecedents that were granted for the sake of entertaining the hypothesis are untenable. Either they lead to unpleasant consequences, or they fail to fit in with other predictions and observations. In either case, they can be withdrawn since there was no deductive warrant for believing them in the first place. In the rest of this chapter we will be looking at a language for representing plans and determining their applicability. In addition we are interested in a procedure for making inferences about plans that allows us to notice when the necessary conditions for an inference are endangered by the consequences of some other hypothesis being considered. It is with regard to this last that the role of temporal reason maintenance in planning becomes clear.

5.2 Planning

The purpose of this chapter is to discuss the role of temporal reasoning in planning. I want to make it clear that I will be simplifying many of the issues that are critical to the construction of realistic planners. In this section, I am primarily interested in showing how the TMM and the temporal reason maintenance mechanism, in particular, assist in detecting and avoiding unpleasant interactions. I will begin by discussing a simple plan language (essentially the plan language of [Charniak 85]).

A plan for achieving a given task is represented as a function of four arguments (*i.e.*, `(plan ?steps ?constraints ?protections ?cost-function)`). The arguments are:

1. a set of individual steps that comprise the plan
2. a set of constraints on the order in which those steps are to be achieved
3. a set of protections which must be maintained if the individual steps in the plan are to achieve their intended purpose. Each protection is a four-tuple, indicating the fact type that is being protected, the step responsible for making it true, and the beginning and end points of the interval over which the fact is being protected.
4. a cost function which provides some measure of how desirable a particular plan is, relative to the other plans appropriate for a given task type

The function is declared as:

```
(define-function (plan PROP PROP PROP FUN))
```

We also define a predicate `to-do` (repeated here from Section 3.6), which is used to index plans according to the type of tasks they are likely to be useful in achieving.

```
(define-predicate (to-do TOKEN PROP PROP))
```

The token corresponding to the task that is being worked on is mentioned in order to make reference to the temporal context in which the task is to be achieved. Quite often the criterion for determining whether or not a plan is applicable depends upon the time during which it is to be carried out. Internally, the steps, constraints, and protections

```
(to-do ?tsk (achieve (on ?x table))
       (plan !<(achieve (clear ?x))
               (puton ?x table)>
             !<(pt=< (end 1) (begin 2))
               (pt= (end 2) (end ?tsk))
               (elt (dist (begin 2) (end 2)) 2 2)>
             !<!<(clear ?x) 1 (end 1) (end 2)>>
             (lambda () *pos-tiny*)))
```

a. Internal format for representing plans

```
(to-do ?tsk (achieve (on ?x table))
       (plan   steps: t1  (achieve (clear ?x))
                      t2  (puton ?x table)
         constraints: (pt=< (end t1) (begin t2))
                      (pt= (end t2) (end ?tsk))
                      (elt (dist (begin t2) (end t2)) 2 2)
         protections: (protect (clear ?x) t1 (end t1) (end t2))))
```

b. A "sugared" version to increase readibility

Figure 5.1: Different notations for describing a plan to place a block on the table

are represented using the list notation of DUCK (see Section 3.2.1). In order to refer to individual tokens corresponding to the steps in a plan, we use numbers indicating the *n*th token corresponding to the *n*th step. This makes it easy for the programs manipulating the plans, but it makes writing and reading such plans difficult. To remedy this, the TMM employs a macro facility that converts a "sugared" version of the plan into a suitable internal form. To illustrate I will present some simple blocks world plans.

The plan to place one block on the table would be represented internally using the notation shown in Figure 4.1.a. In the "sugared" version (shown in Figure 4.1.b), we allow the user to introduce symbols to refer to the steps in the plan. One can also eliminate mentioning any arguments that coincide with the default (*i.e.*, `(plan !<> !<> !<> (lambda () *pos-tiny*))`).

Figures 4.2 and 4.3 show some additional plans necessary for achieving tasks in the blocks world domain. It is also necessary to determine the effects of actions, and to provide

```
(<- (to-do ?tsk (achieve (on ?x ?y))
           (plan   steps: t1  (achieve (clear ?x))
                          t2  (achieve (clear ?y))
                          t3  (puton ?x ?y)
             constraints: (pt=< (end t1) (begin t3))
                          (pt=< (end t2) (begin t3))
                          (pt= (end t3) (end ?tsk))
                          (elt (dist (begin t3) (end t3)) 3 3)
             protections: (protect (clear ?x) t1 (end t1) (end t3))
                          (protect (clear ?y) t2 (end t2) (end t3))))
    (instance-of ?y block))
```

Figure 5.2: A plan for placing one block on another

criteria for what it means for one fact type to contradict another. Some rules necessary for solving simple blocks world problems are described in Figures 4.4 and 4.5.

In order to specify a planner we would have to provide routines for:

1. selecting the next task to work on
2. determining which plans are appropriate for a given task and then choosing from among them
3. actually performing the plan expansion and setting up the necessary protections
4. methods for resolving conflicts (ordering tasks to avoid protection violations)

```
     (<- (to-do ?tsk (achieve (clear ?x))
                (plan  steps: t1 (achieve (on ?y table))
                 constraints: (pt= (end t1) (end ?tsk))))
         (and (instance-of ?x block)
              (tt  (begin ?tsk) (end ?tsk) (on ?y ?x))))

;; One way of clearing a block that has a second block on it is
;; to put the second block on the table (this assumes that each
;; block can have no more than one block sitting directly on it).

     (<- (to-do ?tsk (achieve ?p)
                (plan constraints: (pt= (begin ?tsk) (end ?tsk))
                          utility: (lambda () 0)))
         (tt (begin ?tsk) (end ?tsk) ?p))

;; If something is already true, don't bother trying to achieve it.

     (to-do ?tsk (achieve (and ?p ?q))
            (plan steps: t1 (achieve ?p)
                         t2 (achieve ?q)
            constraints: (pt=< (end t1) (begin ?tsk))
                         (pt=< (end t2) (begin ?tsk))
            protections: (protect ?p t1 (end t1) (end ?tsk))
                         (protect ?q t2 (end t2) (end ?tsk))))

;; To achieve the conjunction of ?p and ?q achieve each
;; separately, and then make sure that they continue to be
;; true at least until the end of the conjunctive task.
```

Figure 5.3: Plans for clearing a block, ignoring a task whose purpose is already achieved, and achieving a conjunction of facts

```
(<- (contradicts (on ?x ?y) (on ?z ?y))
    (thnot (:= ?x ?z)))

(contradicts (on ?x ?y) (clear ?y))

(<- (contradicts (and ?p ?q) ?r)
    (or (contradicts ?p ?r)
        (contradicts ?q ?r)))
```

Figure 5.4: Blocks world contradiction criteria

```
(pcause t (achieve ?p) ?p)

(pcause (and (thnot (:= ?y ?z))
             (on ?x ?y))
        (puton ?x ?z)
        (not (on ?x ?y)))

(pcause (and (instance-of ?y block)
             (thnot (:= ?y ?z))
             (on ?x ?y))
        (puton ?x ?z)
        (clear ?y))

(pcause t (puton ?x ?y) (on ?x ?y))
```

Figure 5.5: Capturing the effects of actions in the blocks world

time the form is evaluated. In such a case, the support for the current answer was said to be augmented. In the form (answer-support (= *support-predications*) *code*), the = indicates that assertions occurring in *code* are to depend *only* upon the *support-predications*. In this case, the support in the current answer is overridden within the scope of the answer-support. In the code fragment in Figure 4.6, I want to keep the variable bindings from the current answer but substitute a new set of support predications. The reason is that I want to be precise in annotating protection failures. If I allowed the tokens in the plan to depend upon the current answer established during plan choice, then if the plan choice criterion is for any reason undermined, all of the tasks in the expansion will become OUT, and all of their expansions in turn, resulting in a virtual barrage of protection failures. Instead, I will show how to set up the dependencies in such a way that makes it easy to to pinpoint and respond to undesirable interactions in a given set of circumstances. Also, recall from Section 3.6 that the macro abductive-support provides the justifications for abductive premises asserted in the course of actualizing an abductive answer. Note that the tokens corresponding to the actualized steps of the plan are gathered in the variable step-tokens. The routines in both *code-to-add-constraints* and *code-to-add-prerequisite-predications* must refer to these tokens. In order to do so, these routines perform a bit of straightforward parsing and substitution not shown in Figure 4.6.

We want to maintain a handle on plans that have been expanded in the time map, in

```
(let (ANS (best-plan-answer ()) FLONUM (best-plan-cost *pos-inf*))
 (for-first-answer (fetch '(and (instance-of ?tsk task)
                                (time-token ?task-type ?tsk)
                                (thnot (or (plan-chosen-for ?tsk)
                                           (primitive ?tsk)))))
   (for-each-answer (fetch '(to-do ?tsk ?task-type
                                   (plan  ?steps
                                          ?constraints
                                          ?protections
                                          ?cost-function)))
        (cond ((< (evaluate ?cost-function) best-plan-cost)
               (:= best-plan-cost  (evaluate ?cost-function))
               (:= best-plan-answer ans*))))
   (and best-plan-answer
        (add '(plan-chosen-for ?tsk))
        (let ((lst TOKEN) (step-tokens ()))
         (abductive-support (+ '(plan-chosen-for ?tsk))
          (bind (ANS (ans* best-plan-answer))
                (add '(applicable-reduction ?tsk))
                (answer-support (= '(plan-chosen-for ?tsk))
                  (for-each-answer (fetch (member ?step ?steps))
                    (add '(and (time-token ?step ?step-token)
                               (instance-of ?step-token task)))
                    (:= step-tokens (cons ?step-token step-tokens)))
                  (for-each-answer (fetch (member ?constraint ?constraints))
                        code-to-add-constraints)
                  (for-each-answer
                    (fetch '(and (member !<?fact-type ?step ?begin ?end>
                                         ?protections)
                                 (tt  ?begin  ?end  ?fact-type)))
                        code-to-add-prerequisite-predications)))))))
```

Figure 5.6: Code fragment for task expansion

case the plan has to be withdrawn. The predicate **plan-chosen-for** is used to erase a plan should it at some point prove to be inappropriate. All tokens and constraints are made to depend upon **plan-chosen-for** predications.

```
(define-predicate (plan-chosen-for TOKEN))
```

5.3 Monitoring planning assumptions

There are three types of protections that are set up in the course of planning. I refer to them as *projection-assumptions*, *prerequisite-assumptions*, and *reduction-assumptions*. The first of these is carried out automatically by the time map in applying **pcause** rules. Projection-assumptions determine whether or not a given effect of an action continues to be believed as the time map changes over the course of planning. Since the planner is not responsible for managing such assumptions, we won't concern ourselves further with them. Prerequisite-assumptions are mentioned explicitly in the individual plans. The prerequisite task is set up solely for the purpose of establishing certain effects that must persist over a specified interval of time. If the effect is for some reason prevented from persisting long enough, then the prerequisite is no longer doing its job and the plan which gave rise to it is not likely to succeed. We generally want to avoid protection failures involving prerequisite-assumptions if at all possible. The predicate **prerequisite** is used to keep track of such assumptions:

```
(define-predicate (prerequisite TOKEN PROP))
```

In Figure 4.6, the *code-to-add-prerequisite-predications* sets up predications of the form **(prerequisite** *tok1* **(protect** *fact-type tok2* **(begin** *tok2*) **(end** *tok3*))), where *tok1* corresponds to the task being expanded, *tok2* to the task responsible for achieving the prerequisite fact, and *tok3* to the task requiring the prerequsite fact being true. When the token corresponding to the task reponsible for the prerequisite fact being true is added to the time map, auto-projection rules see to it that the appropriate effects are added as well. The prerequisite predications will depend upon the prerequisite fact persisting throughout the required interval due to the **tt** conjunct in the fetch for the **for-each-answer** surrounding *code-to-add-prerequisite-predications*.

As much as we would like to avoid undermining prerequisite-assumptions, it is often useful to be able to notice when, by committing to various orderings between tasks, we have placed ourselves in a position where a prerequisite task will fail to achieve its intended purpose. For instance, I might decide to wear a clean shirt this morning so that I will be suitably dressed for an important meeting in the evening. The clean shirt is believed to be a prerequisite for arriving at the meeting properly attired. Suppose that I remember that I have promised to play volleyball during lunch, an activity that is likely to soil the shirt. I may want to modify my plan for preparing for the evening meeting (*e.g.*, bring along a change of clothes), but I am probably not likely to dismiss playing volleyball out of hand.

Protections involving reduction-assumptions arise in the process of selecting an appropriate plan for carrying out a given task. For instance, the following plan for clearing a block is appropriate only if the block is not already clear.

```
(<- (to-do ?tsk (achieve (clear ?x))
           (plan  steps: t1 (achieve (on ?y table))
            constraints: (pt= (end t1) (end ?tsk))))
    (and (instance-of ?x block)
         (tt  (begin ?tsk) (end ?tsk) (on ?y ?x))))
```

The reduction-assumption is simply that it continues to be the case that some block is resting on the block to be cleared. The protection is set up by the TMM in the course of backward chaining. If we assert something in the context of the resulting answer, then the assertion will be dependent upon the protection continuing to be satisfied. To monitor reduction-assumptions we introduce the predicate **applicable-reduction**:

```
(define-predicate (applicable-reduction TOKEN))
```

Using the predicates **prerequisite** and **applicable-reduction**, and the TMM change-driven-interrupt facility, the planner can detect when a plan becomes threatened and use the information about protection failures to direct the debugging process. The simplest way of going about this might involve defining a pair of functions **recover-from-prerequisite-failure** and **recover-from-plan-applicability-failure** and a pair of **if-erased** demons:

```
(if-erased (prerequisite ?tsk ?purpose)
     (recover-from-prerequisite-failure ?tsk ?purpose))

(if-erased (applicable-reduction ?tsk)
     (recover-from-plan-applicability-failure ?tsk))
```

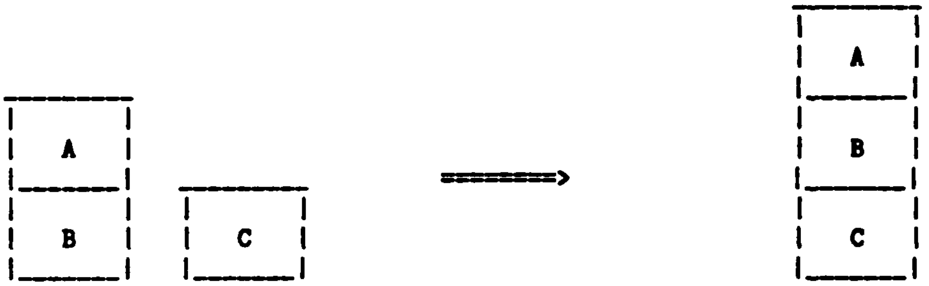
```
Time map depicting potential violation:
    tok1  P
          |----------------------------------->
    tok2  some-task-whose-reduction-depends-upon-P
             |--------------|
    tok3 (not P)
    |>---------------------------------------->

Scheduling constraints:
    (or (pt< (begin tok3) (begin tok1))
        (pt< (end tok2) (begin tok3)))
```

Figure 5.7: A potential protection violation and the scheduling constraints generated by the TMM to avoid it.

Obviously, the definition of these functions will be quite complicated. I won't be addressing the general problem of recovering from planning errors, but the interested reader is referred to [Pednault 85], [Chapman 85], and [Simmons 83].

It is also important for the TMM to provide information about potential problems. Such potential problems arise when a reduction or prerequisite-assumption depends upon P being true throughout an interval, and an action is entered into the time map with effect (not P) such that it is possible that the effect can violate the underlying protection. Potential problems also arise in the event that the planner sets up a protection in a time map that already contains effects that would serve to violate the protection. The time map assists in eliminating such potential violations by generating what are called *scheduling-constraints*. Generally, the TMM generates disjunctions of scheduling constraints such as those shown in Figure 4.7. It is the planner's responsibility to decide whether the action should precede the token currently used in satisfying the protection or whether the action should succeed the protection interval. The TMM just calls a user-defined function by the name of **process-scheduling-constraints** which takes a single argument consisting of a conjunct of disjunctions of scheduling constraints of the form shown in Figure 4.7.

It's not always possible to satisfy a set of scheduling constraints. If there is no way to satisfy the scheduling constraints, then it must be the case that there is some protection violation. The protection violation will involve either a prerequisite-assumption or a reduction-assumption. To simplify things, let's assume that the planner is successful in



```
initial conditions: (on A B) (on B table) (on C table) (clear A) (clear C)
goal: (achieve (and (on A B) (on B C)))
```

Figure 5.8: The "creative-destruction" problem

avoiding protection violations involving prerequisite-assumptions. When a failure involving a reduction-assumption occurs, the planner has basically three options. The simplest solution is often just trying another plan. Of course, this is an option only if there are alternative reductions (plans) that it can try. The second option involves trying another way of ordering the tasks, and the third option requires retracting some other plan in an attempt to eliminate the offending action that gave rise to the failure in the first place.

To illustrate, let's consider a popular blocks world problem. The initial situation and goal are depicted in Figure 4.8. The objective is to devise a sequence of primitive puton actions that will serve to transform the initial situation into one in which the goal `(and (on A B) (on B C))` is satisfied. This problem is a variation on what has been called the "creative-destruction" problem [Charniak 85]. Its solution requires undoing something (in this case `(on A B)`) that was true in the initial situation, and must be true in the final situation, in order to satisfy some intermediate goal (`(on B C)`)[1]. To set up the goal and initial situation we assert:

```
(time-token (clear A) clear1)
```

[1] The creative-destruction is not to be confused with another problem called Sussman's anomaly [Sussman 75]. Sussman's anomaly is superficially similar to the creative-destruction problem; the goal is the same, but the initial situation consists of `(on C A)`, `(on A table)`, `(on B table)`, `(clear C)`, and `(clear B)`. If the planner chooses to put B on C first, it will be forced to reverse this action in order to get A on B. Avoiding this sort of unnecessary work in general requires a certain amount of sophistication (see [Chapman 85] for an interesting analysis).

```
(time-token (clear C) clear2)
(time-token (on A B)  on1)
(time-token (on B table) on2)
(time-token (on C table) on3)
(pt< (begin clear1) *now*)
(pt< (begin clear2) *now*)
(pt< (begin on1) *now*)
(pt< (begin on2) *now*)
(pt< (begin on3) *now*)
(time-token (achieve (and (on A B) (on B C))) stack24)
(instance-of stack24 task)
(pt< *now* (begin stack24))
```

Executing the code fragment in Figure 4.6 will result in expanding the task corresponding to stack24, since stack24 is the only task in the data base. The expansion of stack24 results in two additional tokens (corresponding to the tasks (achieve (on A B)) and (achieve (on B C)), and the addition of two prerequisite-assumptions. One prerequisite-assumption is valid just in case one effect of the (achieve (on A B)) task, that A is on B, is true from the end of the task at least until the end of stack24. The other prerequisite-assumption is similar, and is associated with the task of achieving B on C. Now let's suppose we execute the code fragment in Figure 4.6, and this time it expands the task corresponding to (achieve (on A B)). The planner will use the following plan repeated here from Figure 4.3:

```
(<- (to-do ?tsk (achieve ?p)
           (plan constraints: (elt (distance (begin ?tsk) (end ?tsk)) 0 0)
                      utility: (lambda () 0)))
    (tt (begin ?tsk) (end ?tsk) ?p))
```

We are assuming that the robot controlled by our planner can only lift one block at a time, and so it may seem obvious to the reader that that this is the wrong plan to choose. The reason, however, that it is obvious to us, stems from the fact that we have thought ahead far enough to realize that the robot will have to remove A from B in order to carry out the (achieve (on B C)) task. The only way our simple planner can think ahead is by actually expanding out its tasks and detecting possible problems arising from prior commitments.

To this end, we expand the only remaining task, that corresponding to (achieve (on B C)). In this case, there is only one plan available: the plan shown in Figure 4.2. Notice that one prerequisite of this plan is (clear B). Notice also, that we have two protections that depend upon (on A B), and that (clear B) and (on A B) contradict one another. One protection resulted from the prerequisite for the (achieve (and (on A B) (on B C))) task, and the other protection is associated with the reduction-assumption generated for the plan for achieving (on A B) (*i.e.*, the plan of doing nothing and taking advantage of the fact that (on A B) is already true in the world. There is no alternative plan for achieving the conjunction of (on A B) and (on B C). There is, however, an alternative plan for achieving (on A B). The expansion of the (achieve (on B C)) task does not directly result in a protection failure. What it does result in is a set of scheduling constraints that must be satisfied if the planner is to avoid protection failures. In the current situation, there is no way to satisfy the scheduling constraints generated by the TMM. To see why, let's look at the situation in a little more detail.

A time map depicting our situation is shown in Figure 4.9. To understand what's going on, we have to be clear about how the TMM sets up protections (or passumptions (see Section 4.3.2)), and what it does when it detects a possible protection failure. First, let's consider what the relevant protections are. The prerequisite-condition:

```
(prerequisite TOKEN2 (protect (on A B) TOKEN3 (end TOKEN3) (end stack24)))
```

depends upon the passume predication:

```
(passume (on A B) (begin TOKEN5) (end stack24))
```

The reduction-assumption:

```
(applicable-reduction TOKEN3)
```

depends upon the passume predication:

```
(passume (on A B) (begin on1) (end TOKEN3))
```

When the token TOKEN10 is generated as a result of expanding TOKEN6 (this is accomplished by the rules in Figure 4.5), the TMM generates the following conjunction in an attempt to help the calling program avoid a protection failure:

```
(and (or (pt< (begin TOKEN10) (begin TOKEN5))
         (pt< (end stack24) (begin TOKEN10)))
     (or (pt< (begin TOKEN10) (begin on1))
         (pt< (end TOKEN3) (begin TOKEN10))))
```

```
Frame of reference: *now* Scale 0.3

clear1 (clear A)
<------||--------------------------------------------------------->
clear2 (clear C)
<------||--------------------------------------------------------->
on1 (on A B)
<------||--------------------------------------------------------->
stack24 (achieve (and (on A B) (on B C)))
           |----------------|
TOKEN2 (and (on A B) (on B C))
                            ||------------------------------------>
TOKEN3 (achieve (on A B))
                      |-----|
TOKEN5 (on A B)
                            ||------------------------------------>
TOKEN4 (achieve (on B C))
                      |-----|
TOKEN6 (on B C)
                            ||------------------------------------>
TOKEN7 (achieve (clear C))
                       |-|
TOKEN11 (clear C)
                         ||--------------------------------------->
TOKEN8 (achieve (clear B))
                       |-|
TOKEN10 (clear B)
                         ||--------------------------------------->
TOKEN9 (puton B C)
                           |-|
TOKEN12 (not (on B table))
                             ||----------------------------------->
```

Note: This time map was not directly generated by the rules presented in this chapter. To make the display clearer I have added some additional constraints that don't change the problem, but make it easier to tell what's going on.

Figure 5.9: Time map after expanding TOKEN2, TOKEN3, TOKEN4, and stack24, but before adding any scheduling constraints.

Unfortunately, since we also know that:

```
(and (pt= (begin TOKEN5) (end TOKEN3))
     (pt< (begin on1) (begin TOKEN10))
     (pt=< (begin TOKEN10) (end stack2)))
```

there is no way to satisfy the scheduling constraints. What we do instead is try to satisfy those constraints that ensure that there are no protection failures involving prerequisite-assumptions. In this case, we have to satisfy:

```
(or (pt< (begin TOKEN10) (begin TOKEN5))
    (pt< (end stack24) (begin TOKEN10)))
```

which can be done in only one way by adding (pt< (begin TOKEN10) (begin TOKEN5)). Adding this constraint will generate a protection failure that will result in (applicable-reduction TOKEN3) becoming OUT. This, in turn, will result in firing one of the if-erased demons defined on page 228, thereby calling the function recover-from-plan-applicability-failure with the single argument TOKEN3. In general, this function might need to perform an arbitrary amount of computation in order to decide how to resolve the difficulty, but let's just assume that the planner decides to replan the (achieve (on A B)) task. We first have to get rid of the old plan by erasing (plan-chosen-for TOKEN3). In this case the only assertion associated with the plan (the plan to do nothing) is the constraint that the task takes no time (its duration is 0). Now, when we expand TOKEN3 the only applicable plan is that shown in Figure 5.2.

After this, the rest of the expansions will occur without a problem. The final plan will consist of the sequence of primitive actions {(puton A table) (puton B C) (puton A B)}.

It is fairly simple to construct a planner using the TMM that performs with the power of Tate's NONLIN [Tate 77] or Vere's DEVISER [Vere 83]. To do so it is necessary to keep track of all reduction and ordering decisions in order to guide backtracking. The temporal reason maintenance system of the TMM also makes it quite simple to implement dependency-directed methods for debugging plans [Daniel 83] [Vere 85] [Wilkins 85]. The auto-projection facility of the TMM has not as yet been fully exploited, but methods for performing conditional projection should simplify reasoning about the physics of a domain. Such methods also provide a means for recognizing when certain effects combine to give rise to new tasks. For instance, suppose we have a rule that says, whenever the doors to one of the welding bays have to remain open during an interval in which welding is being performed, then you have a task to turn on a powerful exhaust fan to prevent the fumes from escaping into other parts of the factory. It is convenient to represent this piece of knowledge as a separate rule, rather than attaching the information to plans which perform welding, or might have reason to open the doors to the welding bay. In Chapter 3, I gave some other examples in which this sort of reasoning might come in handy.

The TMM also provides a number of utilities which have proved useful in implementing the FORBIN planner [Miller 85a] and I'll take a little time to describe them here.

At some point during expansion, tasks and events are treated as primitive. As I mentioned in the introduction, the criterion for determining what is primitive and what is not is purely pragmatic. The criterion may depend upon what other tasks are actively being pursued or on how much time the planner actually has to think about what to do. It is necessary in reasoning about tasks with deadlines to assign an estimate of how long primitive tasks are likely to take. If, however, the duration of primitive tasks is the only way we have of determining how long nonprimitive tasks will take, then we are likely to run into some serious problems. The reason is simply that the planner is not likely to recognize a deadline failure until all of the tasks are expanded down to primitives. The way to get around this is quite simple. Each task type provides an estimate of how long it is likely to take. This estimate is made, assuming that all of the steps in the plan that will ultimately be used to carry out the task are executed without interruption. Of course, in the completed plan the steps are quite likely to be spliced together with tasks for carrying out other plans. The time actually spent in service to a given task will be spread out over time. The estimated duration of a task is added to the time map at the time it is first entered into the time map. Later, when it is expanded, this estimate is removed. The duration of the task can then be determined as the sum of the duration of the tasks in its expansion. This technique is a special instance of what is referred to as *hierarchical* planning.

Hierarchical planning is a somewhat refined version of what we have been calling reductionist planning. A reductionist planner is one that breaks down (or reduces) a complex task into a set of steps, or subtasks, that can be considered in relative isolation. Hierarchical planning [Sacerdoti 74] deals with certain strategies for performing the reductions in such a way as to avoid unnecessary search. The space of possible plans for achieving even the simplest sort of task is quite large. The search itself consists of making and rescinding decisions concerning task ordering, the method of reducing an individual task, and the choice of instruments or resources used in carrying out a particular task. If the reduction process is not skillfully directed, there is a significant chance that the program will not terminate in a reasonable amount of time.

Hierarchical planning attempts to direct the reduction process by decomposing the problem in such a way that decisions made early in planning do not preclude options for achieving tasks yet to be considered. In effect, this means either the early decisions don't really matter, or the planner has the necessary information to anticipate and sidestep possible problems. Providing the sort of direction needed to avoid reversing decisions has turned out to be quite difficult in practice. Special purpose strategies have been proposed, however, to significantly reduce the search. One such strategy is employed by the FORBIN planner to avoid reduction and ordering decisions that might lead to deadline failures. The technique involves using compiled information to anticipate scheduling problems (this constitutes a limited form of lookahead). The library of methods for reducing tasks contains precompiled estimates of the resources (in particular the amount of time) each task in a reduction is expected to use. In performing reductions or considering which of several ordering constraints to add in resolving an interaction, the planner uses these estimates to make decisions that are not likely to require reversal. The estimates of the resource usage for a given task are discarded after that task is reduced. A more detailed estimate is then available from the estimates of the resource use of all the subtasks. This technique has been especially useful in avoiding deadline failures and it promises to generalize to other sorts of resource-allocation problems.

The TMM also provides a simple method for managing a resource such as a set of machine tools. This method essentially linearizes all tasks which propose to employ a given machine. The predicate for invoking this utility behaves similarly to the true-throughout `tt` predicate at query time. The predicate definition is:

```
(define-predicate (reserved POINT POINT PROP))
```

At query time, a conjunct of the form `(reserved ?begin ?end ?machine)` is treated by the backward chaining machinery as `(tt ?begin ?end (production-status ?machine free))`. If an assertion occurs in the context of an answer in which `(reserved ?begin ?end ?machine)` succeeded, then the TMM will constrain `?begin` to follow whatever token asserting `(production-status ?machine free)` the time map used in making the query succeed. This is the same as it would do for a `tt` conjunct. In addition, the TMM will assert

```
Frame of reference: *ref*  Scale: 2.0

running35 (operational-status lathe35 running)
||-----------------------------------------------------------------------> 
free1 (production-status lathe35 free)
||-----------------------|
make1 (manufacture widget)
                         |-------------|--|
inuse1 (production-status lathe35 inuse)
                         |-------------|--|
free2 (production-status lathe35 free)
                                       |--||----------------------------->
make2 (manufacture gizmo)
 |-----------------------------------------|||
```

Figure 5.10: Initial situation for demonstrating the TMM resource handling utility

(production-status ?machine inuse) throughout the interval **?begin** to **?end**, and also constrain **?end** to precede any interval also requiring the use of **?machine** that is currently unordered with respect to **?end**. A simple example *will* make *this clear*.

Consider the time map depicted in Figure 4.10. There are two tasks shown: one **make1** to manufacture a widget, and a second **make2** to manufacture a gizmo. The **(manufacture widget)** task has presumably already been expanded. The time map indicates that **lathe35** is reserved throughout the interval associated with **make1**. There are two intervals in which **lathe35** is free for use by another task: just before **make1** and just after. Now, let's suppose that we want to expand the **(manufacture gizmo)** task, **make2**. This task has to be completed before the current reference point ***ref***. In Figure 4.10, the duration of **make2** is constrained only by the deadline ***ref***. Obviously, if we want to reserve the lathe for some period of time, we have to be able to specify that amount of time in the query. Figure 4.12 shows a rule for establishing the applicability of a simple manufacturing plan. Consider the applicability criteria itself. Notice that the rule determines a different pair of bounds, **?low-estimate** and **high-estimate**, depending upon whether the product-type is widget or gizmo, and whether the **machine** chosen for the job is a lathe or a milling machine. The conjunct **(A (elt (distance (begin ?tok) (end ?tok)) ?low-estimate *pos-inf*))** enforces the appropriate lower bound during the query. This means that the conjunct **(reserved (begin ?tok) (end ?tok) ?machine)** will succeed only if it can find

```
Frame of reference: *ref*  Scale: 2.0

running35 (operational-status lathe35 running)
||---------------------------------------------------------------------->
free1 (production-status lathe35 free)
||
make1 (manufacture widget)
                        |-------------|--|
inuse1 (production-status lathe35 inuse)
                        |-------------|--|
free2 (production-status lathe35 free)
                                      |--||---------------------------->
make2 (manufacture gizmo)
 |---------|------------|--|
inuse2 (production-status lathe35 inuse)
 |-----------||------------|
free3 (production-status lathe35 free)
            |--------------||-----|
setup1 (setup lathe35 gizmo)
 |--------------------|||
load1 (load lathe35 (blank gizmo))
 |--------------------|||
run1 (run lathe35)
 |--------------------|||
```

Figure 5.11: Time map after plan expansion involving resource query

```
(<- (to-do ?tok (manufacture ?product-type)
           (plan
             steps: t1 (setup ?machine ?product-type)
                    t2 (load ?machine (blank ?product-type))
                    t3 (run ?machine)
       constraints: (pt= (end ?tok) (end t3))
                    (pt=< (begin ?tok) (begin t3))
                    (pt< (end t1) (begin t3))
                    (pt< (end t2) (begin t3))
estimated-duration: (elt (distance (begin ?tok) (end ?tok))
                         ?low-estimate ?high-estimate)
       protections: (protect (installed ?machine (fixture ?product-type))
                             t1 (end t3))))
    (& (or (and (= ?product-type widget)
                (or (and (inst ?machine lathe)
                         (>= (turning-radius ?machine) 16)
                         (:= ?low-estimate 5) (:= ?high-estimate 6))
                    (and (inst ?machine milling-machine)
                         (attachment ?machine rotary-table)
                         (:= ?low-estimate 22) (:= ?high-estimate 25))))
           (and (= ?product-type gizmo)
                (inst ?machine lathe)
                (>= (turning-radius ?machine) 12)
                (or (and (>= (turning-radius ?machine) 16)
                         (:= ?low-estimate 6) (:= ?high-estimate 7))
                    (:= ?low-estimate 7) (:= ?high-estimate 8))))
       (A (elt (distance (begin ?tok) (end ?tok)) ?low-estimate *pos-inf*))
       (reserved (begin ?tok) (end ?tok) ?machine)
       (tt (begin ?tok) (end ?tok)
           (operational-status ?machine in-service))))
```

Figure 5.12: Plan for expanding tasks for manufacturing widgets and gizmos

an interval at least `?low-estimate` in duration such that it is true throughout the interval that `(production-status ?machine free)` and `make2` can be constrained to fall within that interval. In the time map in Figure 4.10, there is only one machine suitable for the `(manufacture gizmo)` task, namely `lathe35`. Given the length of time required for manufacturing a gizmo with a lathe, and the deadlines imposed on the `make2` task, the only time it is possible to use the plan shown in Figure 4.12 is before the `make1` task. The result of expanding the plan is shown in Figure 4.11. Notice that the `make2` token is constrained to endure between 6 and 7 units. This is due `estimated-duration` constraint found in the plan schema. The time map in Figure 4.11 guarantees that `lathe35` will be available for use by `make2`.

5.4 Recognizing opportunities to improve plans

In this section, I want to describe how the methods for dealing with alternatives presented in Section 3.8 fit in with the techniques of this chapter. The example to be used in this section is one that you should be quite familiar with by now: manufacturing a bunch of widgets and gizmos. The only difference from the problems we looked at towards the end of Section 4.3 is that the planner will be considering more than one method for achieving certain manufacturing tasks. To give the reader a better idea of what's going on, I will be depicting the time maps generated during planning in a manner that captures the information content, but belies the conciseness of the underlying representations. Each time map display corresponds to a particular admissible choice assignment. In addition, I will use simplified versions of system generated time maps to isolate just those aspects of the situation that are important for understanding the example.

The problem presented to the planner is as follows. There are four tasks, two to make widgets and two to make gizmos. Widgets are rather large and unwieldy objects. You can manufacture a widget on a lathe with a turning radius of at least 16 inches or on a milling machine with a rotary table. Unfortunately, it takes forever to make a widget on a milling machine. A special fixture for use on a lathe avoids many of the tedious operations that have to be carried out manually on the milling machine. Still, it is convenient to have the milling machine available as an alternative method for producing widgets. Gizmos are somewhat more delicate, but they require the thread cutting capability of a lathe. You can manufacture a gizmo on any lathe with a turning radius of at least 12 inches. This is a

```
(<- (to-do ?tok (manufacture ?product-type)
           (plan
             steps: t1 (if ?installation-step-flag
                           (setup ?machine ?product-type))
                    t2 (load ?machine (blank ?product-type))
                    t3 (run ?machine)
       constraints: (pt= (end ?tok) (end t3))
                    (pt=< (begin ?tok) (begin t3))
                    (pt< (end t1) (begin t3))
                    (pt< (end t2) (begin t3))
estimated-duration: (elt (distance (begin ?tok) (end ?tok))
                         ?low-estimate ?high-estimate)
       protections: (protect (installed ?machine (fixture ?product-type))
                             t1 (end t3))))
    (& (or (and (= ?product-type widget)
                (or (and (inst ?machine lathe)
                         (>= (turning-radius ?machine) 16)
                         (:= ?low-estimate 5) (:= ?high-estimate 6))
                    (and (inst ?machine milling-machine)
                         (attachment ?machine rotary-table)
                         (:= ?low-estimate 22) (:= ?high-estimate 25))))
           (and (= ?product-type gizmo)
                (inst ?machine lathe)
                (>= (turning-radius ?machine) 12)
                (or (and (>= (turning-radius ?machine) 16)
                         (:= ?low-estimate 6) (:= ?high-estimate 7))
                    (:= ?low-estimate 7) (:= ?high-estimate 8))))
       (A (elt (distance (begin ?tok) (end ?tok)) ?low-estimate *pos-inf*))
       (reserved (begin ?tok) (end ?tok) ?machine)
       (or (tt (begin ?tok) (end ?tok)
               (installed ?machine (fixture ?product-type)))
           (:= ?installation-step-flag true))
       (tt (begin ?tok) (end ?tok)
           (operational-status ?machine in-service))))
```

Figure 5.13: Plan for expanding tasks for manufacturing widgets and gizmos

	lathe1	lathe2	mill1
widgets	NA	[5,6]	[22.25]
gizmos	[7,8]	[6,7]	NA

Figure 5.14: Approximate times (in minutes) to complete widget and gizmo manufacturing tasks on the three available machines

```
(<- (to-do ?tsk (setup ?machine ?product-type)
        (plan constraints: (elt (distance (begin ?tok) (end ?tok)) 0 0)))
    (tt (end ?tok) (end ?tok)
        (installed ?machine (fixture ?product-type))))

(<- (to-do ?tsk (setup ?machine ?product-type)
          (plan steps: t1 (remove ?machine (fixture ?type))
                       t2 (store (fixture ?type) tool-room)
                       t3 (install ?machine (fixture ?product-type))
          constraints: (pt= (end ?tok) (end t3))
                       (elt (distance (begin ?tok) (end ?tok)) 4 5)))
    (or (& (tt (begin ?tok) (end ?tok) (installed ?machine (fixture ?type)))
          (thnot (:= ?product-type ?type)))
        (tt (begin ?tok) (end ?tok) (installed ?machine nothing))))
```

Figure 5.15: Plans for expanding the task to install the necessary fixtures for manufacturing widgets and gizmos

variation on the plan of Figure 4.12, but I'm going to complicate things somewhat by adding a criterion for an optional part of the plan schema. Essentially, what we want to do is to use a little foresight when selecting a time and method for carrying out plans to achieve goals. I will assume that one of the most expensive operations in carrying out widget and gizmo manufacturing tasks on a lathe involves installing widget and gizmo fixtures. This means that at the time we are considering when to perform the task, we might as well also take into account what sort of things might serve to expedite the task. If we can find an interval such that there's a lathe available with the right fixture already installed, then it would be wise to use that interval in most situations. Figure 4.13 shows a plan schema with associated applicability criteria that results in a different expansion depending on whether or noi the appropriate fixture is installed in the machine chosen for the task. In the factory we will be considering, there are two lathes and one milling machine mill1. The first lathe lathe1 has a 12 inch turning radius and the second lathe2 has a 16 inch turning radius. Figure 4.14 shows the approximate times required to complete widget and gizmo manufacturing tasks using these machines according to the constraints set up in the plan of Figure 4.13. I will assume that the utility function (not shown) will take into account the machine chosen for the job, the type of product being made, and whether or not the correct fixture is already installed.

The only other plans I will be referring to involve the task for setting up a machine with the proper fixture. These plans are shown in Figure 4.15. The first of these plans is a noop. The reduction-assumption for this plan is that the appropriate fixture is already installed in the machine. The second plan shown in Figure 4.15 has the reduction-assumption that some fixture other than the one needed is installed, thereby requiring its removal and the installation of the correct fixture. This reduction-assumption ensures that that this plan will be applicable only in a situation where it is necessary. It may be the case that at the time the setup task is expanded, there is no indication that the correct fixture will be installed over the required interval. Subsequent expansions, however, may result in the correct fixture being installed. The reduction-assumption enables the planner to recognize such fortuitous situations and re-expand the setup task to take advantage.

In the initial situation, neither of two lathes has any fixture installed. The first task gizmo1 is expanded and the planner decides to use lathe1. Since there is no fixture installed in lathe1, the expansion includes a step to install the widget fixture.

Next, the planner decides to work on the first of the two widget manufacturing tasks,

```
(let ((list-of-options ()))
     (for-each-answer
        (fetch '(and (to-do ?task ?type ?plan)
                     (viable-alternative ?task ?plan)))
        (:= list-of-options (cons (option '(use ?task ?plan))
                                  list-of-options))
        (answer-support (+ '(use ?task ?plan))
              code-for-expanding-plans-in-the-time-map))
     (mutually-exclusive list-of-options))
```

Figure 5.16: Code fragment for expanding alternative plans for a task

	PWD1	PWD2	PWD3	PWD4
gizmo1	lathe1	lathe1	lathe1	lathe1
widget1	lathe2	mill1	lathe2	mill1
gizmo2	lathe2	lathe2	lathe1	lathe1

Figure 5.17: Machines chosen for the tasks gizmo1, widget1, and gizmo2 in four separate partial world decriptions

```
Frame of reference: *ref*  Scale: 1.0

installed1 (installed lathe1 nothing)
||¯¯¯¯¯|
installed2 (installed lathe2 nothing)
||¯¯¯¯|
gizmo1 (manufacture gizmo) <<using lathe1>>
   |-------|¯|
setup1 (setup lathe1 gizmo)
     |-|
installed3 (installed lathe1 (fixture gizmo))
       ||¯-------------------------------------------------------------->
widget1 (manufacture widget) <<using lathe2>>
   |-----|¯|
setup2 (setup lathe2 widget)
    |-|
installed4 (installed lathe2 (fixture widget))
      ||¯¯¯¯¯¯¯|
gizmo2 (manufacture gizmo) <<using lathe2>>
           |------|¯|
setup3 (setup lathe2 gizmo)
             |-|
installed5 (installed lathe2 (fixture gizmo))
               ||¯----------------------------------------------------->
```

Figure 5.18: Time map depicting the situation in PWD1

```
Frame of reference: *ref*  Scale: 1.0

installed1 (installed lathe1 nothing)
||‾‾‾‾‾‾|
installed2 (installed lathe2 nothing)
||‾‾‾‾‾|
gizmo1 (manufacture gizmo) <<using lathe1>>
   |-------|‾|
setup1 (setup lathe1 gizmo)
      |-|
installed3 (installed lathe1 (fixture gizmo))
        ||-----------------------------------------------------------------> 
widget1 (manufacture widget) <<using mill1>>
   |-----------------------|‾‾‾|
gizmo2 (manufacture gizmo) <<using lathe2>>
   |------|‾|
setup3 (setup lathe2 gizmo)
     |-|
installed5 (installed lathe1 (fixture gizmo))
       ||------------------------------------------------------------------>
```

Figure 5.19: Time map depicting the situation in PWD2

```
Frame of reference: *ref*  Scale: 1.0
                      .
installed1 (installed lathe1 nothing)
||~~~~~~|
installed2 (installed lathe2 nothing)
||~~~~~|
gizmo1 (manufacture gizmo) <<using lathe1>>
   |------|~|
setup1 (setup lathe1 gizmo)
      |-|
installed3 (installed lathe1 (fixture gizmo))
        ||~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
widget1 (manufacture widget) <<using lathe2>>
   |-----|~|
setup2 (setup lathe2 widget)
     |-|
installed4 (installed lathe2 (fixture widget))
       ||~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
gizmo2 (manufacture gizmo) <<using lathe1>>
            |------|~|
```

Figure 5.20: Time map depicting the situation in PWD3

```
Frame of reference: *ref*  Scale: 1.0
                      .
installed1 (installed lathe1 nothing)
||~~~~~~|
installed2 (installed lathe2 nothing)
||~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
gizmo1 (manufacture gizmo) <<using lathe1>>
   |--------|~|
setup1 (setup lathe1 gizmo)
      |-|
installed3 (installed lathe1 (fixture gizmo))
        ||~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>
widget1 (manufacture widget) <<using mill1>>
   |---------------------|~~~|
gizmo2 (manufacture gizmo) <<using lathe1>>
              |-------|~|
```

Figure 5.21: Time map depicting the situation in PWD4

widget1, and this time the planner considers two alternatives. The first plan uses the larger of the two lathes **lathe2**, and the second plan uses the milling machine **mill1**. Obviously these two plans represent exclusive options, so we'll want to use the machinery developed in Section 3.8 to distinguish the two plans. It's fairly simple to modify the task expansion code shown earlier in Figure 4.6 to handle the appropriate tagging of alternative plans. The required modifications include code for (1) collecting several answers corresponding to alternative plans for achieving the same task, (2) creating an option corresponding to each alternative plan, (3) adding the appropriate option to the support of each answer as the associated plan is expanded into the time map, and (4) making sure that the alternatives are declared mutually exclusive.

The actual process of selecting the set of viable alternatives could be carried out entirely in DUCK. For this selection process, we might employ a predicate **viable-alternative**. (The hardest part of handling multiple alternatives will be encoding the knowledge for choosing a set of alternatives in a set of rules involving such a predicate.) The code for handling the expansion of multiple alternatives might look something like the LISP fragment shown in Figure 4.16. To simplify things I assume that the process of task selection has already been carried out and **?task** and **?type** are bound in the current answer **ans***: **?task** is bound to a time token corresponding to task in need of reduction and **?type** is bound to the type of that token. As each viable alternative is selected, the associated plan is expanded into the time map supported by a option corresponding to that alternative. In a full implementation, there would have to be tests to make sure that an option is not added in cases where there is only one alternative being considered. There would also have to be additional code to ensure that reduction assumptions get handled properly. I'll ignore these complications and return to our example.

At this point, the time map contains two possibilities for achieving the **widget1** task, and the planner is considering methods for carrying out **gizmo2**. Using **lathe1** looks like a good idea, since the task **gizmo1** has already installed the gizmo fixture in this machine. In the process of expanding this alternative, the token **gizmo2** is automatically constrained to follow **gizmo1**, dependent upon the option corresponding to using **lathe1** for **gizmo2**. This constraint is due to an abductive premise in the current answer generated in the course of applying the rule in Figure 4.13.

The other lathe **lathe2** will also work, but in this case the planner will have to make the necessary fixture installation. Let's suppose that the planner wants to leave its options open, and decides to add alternatives corresponding to both lathe1 and lathe2 to the time map. There is one additional complication. As the planner expands the installation task for the plan to carry out gizmo2 using lathe2, it will notice a potential protection violation. Recall that the planner is already considering a plan for widget1 using lathe2. One of the reduction assumptions for this plan involves protecting the fact that the widget fixture is installed in lathe2 throughout the interval widget1. Since the installation task for gizmo2 involves installing the gizmo fixture in lathe2, there is a potential protection violation. At the point when the expansion occurs, the TMM will inform the planner that gizmo2 must follow widget1 just in case lathe2 is used for both tasks. I'll assume that the planner responds to this suggestion by adding the appropriate constraint.

Now there is one alternative for gizmo1 and two alternatives apiece for each of widget1 and gizmo2. There are four admissible choice assignments corresponding to four different partial world descriptions. I'll refer to these partial world descriptions as PWD1, PWD2, PWD3, and PWD1. Figure 4.17 shows the machines chosen for the tasks gizmo1, widget1, and gizmo2 in these four distinct partial world decriptions. In addition, I have provided four time maps in Figures 4.18, 4.19, 4.20, and 4.21 depicting the situations in PWD1, PWD2, PWD3, and PWD1 respectively.

Now let's consider the possibilites for expanding widget2, and to simplifiy things, suppose that this task is constrained in such a way that it's not possible to use mill1. The planner's only alternative is to use lathe2. There are two opportunities for avoiding the cost of installing the widget fixture. These opportunities correspond to PWD1 and PWD2. In the case of PWD2, all three tasks widget1, gizmo2, and widget2 would have to be performed in sequence. I could easily provide additional constraints corresponding to deadlines for these tasks so that such a sequence would not be possible. In this case, PWD1 would be the only choice. If we assume that using mill1 is to be avoided (this is often the case as milling machines are versatile machines that shouldn't be tied up unnecessarily), then perhaps it should be obvious to go for the set of alternatives underlying PWD1. This would also be true if you were looking for the shortest overall plan. I don't want to pretend that this decision is a simple one. The only point I'm making here is that the time map provides the right sort of information to assist in making such decisions.

The process of selecting a reasonable set of alternative plans, noticing opportunities to improve plans, and ruling out alternatives that are shown to be less attractive than the competition is extremely hard. It is impossible unless the planner has some means for exploring the possibilities. The TMM method for maintaining several partial world descriptions simultaneously provides an alternative to traditional backtracking methods. The main advantage of the approach proposed here is that, by representing several possibilites simultaneously, the planner has access to the right sort of information for making an informed decision. Admittedly, there is a cost attached to maintaining several partial world descriptions. However, since a great deal of the cost associated with storing and maintaining multiple PWDs is shared, the techniques described in this dissertation should perform better on the average than sequential (backtracking) methods. Of course, such a claim assumes that the solution is to be found in that part of the search space defined by the sets of alternatives selected. If the planner makes poor selections, then at least it should be no worse off than it would have been backtracking. The advantage of the multiple PWD method is simply that the selection process need only include the correct choices.

5.5 Summary

This section has provided some examples illustrating how the techniques developed for temporal imagery are applicable to planning. In particular, I have shown how one might go about representing plans, actions, and the effects of actions using the TMM. I have also shown how one can reason about the applicability of plans for achieving tasks involving partial orders and metric time. The hard problems of determining how to recover from an interaction detected in the process of expanding a plan and how to choose an appropriate plan in order to avoid such interactions in the first place, are not addressed by issues raised in this chapter. I have, however, presented techniques for efficiently detecting such interactions, and setting up the necessary machinery to tell the planner what constraints he might possibly impose to avoid interactions. The technique of representing the effects of actions using auto-projection rules provides a powerful and modular approach to representing plans. Also described were techniques used in the FORBIN planner for providing temporary estimates of the duration of tasks and handling a special class of resources. The basic ideas for planning using a single partial world description were then extended to reasoning about alternatives. and the methods for maintaining several partial world descriptions simultaneously were proposed as an alternative to traditional backtracking methods used in planning.

Chapter 6

Conclusions

6.1 Contributions

This dissertation deals with extending classical predicate-calculus data base systems to naturally and efficiently handle a useful class of temporal reasoning chores (see Section 1.4). Common sense reasoning invariably requires some consideration of time. Temporal reasoning of the sort described in this work is so common, in fact, that I have proposed building the necessary supportive machinery into the deductive engine underlying predicate-calculus data base systems.

Chapters two through four of this dissertation describe a particular approach to extending classical data base systems to deal with time. There are three issues to address in assessing the contribution of this extension:

1. **naturalness and expressive power of the notation:** How clear is the language for expressing temporally dependent facts, and what are the limitations on what I can say in this language?

2. **supported functionality:** What sort of operations on temporal data are handled by augmented deductive engine?

3. **efficiency:** What sort of performance can I expect from the system in handling routine temporal reasoning chores of the sort that typically arise in artificial intelligence applications?

In the following three subsections I will speak of each of these in turn.

6.1.1 A notation for expressing temporal information

The ontological commitments presented in Chapter 2 (Section 2.2) are so basic and so innocuous that I refuse to spend any time defending them (though I am sure there will be those that take exception to this stance). I have chosen to take points as primitive and define intervals in terms of points, but that is essentially a detail. It happens to make sense from a computational point of view, and I had no intuitions concerning whether points or intervals as primitive provide a better foundation for a psychological theory (but see [Allen 85] for an interesting discussion).

The language primitives introduced in Section 3.3 are again rather basic, though slightly more controversial. Some may question the emphasis on metric constraints. I think that the ability of the TMM to express and reason about duration and metric information is one of its strong points. However, there are those that would claim that interval relations like `precedes`, `overlaps`, and `meet` [Allen 83] deserve greater prominence in a theory of time. I concur and hope that the fact that predicates like `precedes`, `overlaps`, and `meet` can be easily defined in terms of `elt` and the function `distance` with minimal computational overhead will appease these potential detractors. I think, however, that reasoning about duration involving metric constraints is a critical prerequisite to a computational approach to reasoning about time. Purely qualitative information is easily handled in the time map using `elt`, `distance`, and the simple quantity space [Forbus 84] {`*neg-inf*`, `*neg-tiny*`, `0`, `*pos-tiny*`, `*pos-inf*`}. Incomplete information, in the form of partially ordered time tokens and "fuzzy" durations, is easily expressible in the notation for referring to upper and lower bounds on the time separating pairs of points. The TMM provides a suitable foundation for reasoning about all sorts of temporal information, qualitative as well as quantitative. Suitable predicates can easily be defined to the meet the individual programmers representational requirements.

The class of useful queries possible using the predicates `elt`, `time-token`, and `tt` in conjunction with the operators `M` and `A` is quite extensive. I hope that the examples in Chapter 3 have demonstrated some of the range of the TMM in this regard. There are also severe restrictions on what can be expressed using these language constructs. The most glaring deficiency concerns the expression of information about continuously changing quantities. Actually the notion of "continuous change" is a red herring. The time map cannot even deal effectively with discretely changing quantities like the number of lathes available for use on a production line. I have considered extensions to the time map for performing the requisite reasoning tasks, but the requirement that the extension deal with partial orders and provide a corresponding extension to the temporal reason maintenance algorithms has proved difficult to meet. The computational problems for the general case appear to be intractable (the issues are quite similar to those involving the problem of reasoning about resources (*i.e.*, transactions on "pools" of similar objects) discussed in Section 1.5). I believe that the techniques described in this dissertation can be extended to deal with certain aspects of reasoning about continuous change. The hard part will be isolating a useful functionality that can be efficiently supported (see [Chapman 85] for a proposal along these lines involving what Chapman calls *cognitive cliches*). The difficult part is not expressing such information, it is *reasoning* about such information. And reasoning about temporally dependent facts is what this dissertation is most concerned with.

6.1.2 Functional requirements for temporal data base management

As I have repeated many times in the course of this dissertation, the temporal information possessed by an agent in realistic situations is both incomplete and potentially defeasible. In order to contend with these factors, I introduced the notion of persistence (Section 1.3.1) and a strategy for interpreting the information stored in time maps (Section 3.5). This provided the basis for processing queries in partially ordered time maps involving default assumptions about the persistence of fact tokens. The notion of protection (Section 1.3.2) was extended to deal with defeasible antecedent conditions involving facts persisting over certain intervals, where these conditions are used as the basis for making various consequent predictions. These techniques for managing temporal deductions fit in quite nicely with methods for performing forward inference in static data base systems (*e.g.*, [deKleer et al 77]).

Since the information contained in a time map is typically incomplete, there has to be some method of exploring some of the possible completions. The query routines should make the applications program aware of various possiblities that might provide a basis for supporting certain sought-after conclusions or consequences. The TMM facilitates this sort of reasoning by providing an abductive interpretation of the "true throughout" predicate (Section 3.6). Using the abductive operator A, the system can propose additional constraints (abductive premises) that restrict the current partially ordered time map in order to allow a deduction to precede that would fail otherwise. In addition, the use of abductive answers provides an application program with an inexpensive method for keeping around several hypothetical situations to facilitate the process of selecting the best such hypothesis.

In addition to techniques supporting what was termed controlled forward inference (Section 3.2), the TMM also provides the necessary machinery for performing a temporal version of pattern directed inference. Section 3.7 described notation for expressing rules that support a simple form of temporal implication (referred to as overlap chaining rules), as well as a method for capturing certain causal effects of actions represented in the time map (referred to as auto-projection rules). These rules don't suffer from the timing complications inherent in controlled forward inference. They are especially useful in planning applications (Section 5.2), as they free the planner to concentrate upon only those aspects of a planning situation that are of critical importance in achieving the desired coordination of a set of conjunctive tasks. Side effects of actions need only come to the attention of the planner when they are noticed to interfere with or somehow facilitate a plan currently under consideration.

Finally, there are times when reasoning about a single course of events or partial world description (Section 2.5) is inadequate for certain tasks. In Section 3.8, I describe an approach to dealing with uncertainty and indecision that involves maintaining several partial world descriptions simultaneously. This approach is functionally quite different from methods involving the use of traditional context mechanisms [Wilkins 84]. The important difference is that the deductive system takes on the responsibility of noticing sets of alternatives or choices (constituting contexts in a more traditional approach) in which a given query might succeed or the consequent predictions of a particular auto-projection rule might be realized. This provides a mechanism for recognizing opportunities for improving plans (Section 5.4) and avoiding some amount of backtracking in certain situations [deKleer 84].

6.1.3 Efficiently managing temporal data bases

In Section 1.4, I claimed that the techniques developed in this dissertation constituted a solution to the temporal data base update problem (sometimes referred to as the frame problem). There are two aspects to this problem. The first concerns determining whether or not a proposition is true at a point or throughout an interval. The second aspect concerns the reorganization of a temporal data base in response to various changes in its contents and the detection of important consequences that follow from those changes. The TMM addresses both of these aspects.

The organization of the time map makes it quite simple to determine whether or not a token of a given type spans a given point or interval. This determination can be made without taking into consideration all of the events and their associated effects that fall between the beginning of that token and the end of the period in question. The main techniques used in performing such operations involve the use of temporal conditions (used to monitor the validity of relationships between pairs of points) (Section 4.3.3), clipping constraints (used in resolving apparent contradictions in order to restrict the duration of persistences) (Section 4.3.1), and caching constraints for expediting the estimation of point-to-point distances (used in determining the ordering relationship of two points) (Section 4.4.3). Simplifying somewhat, determining whether or not a fact P is true throughout an interval pt1 to pt2 requires finding the token of type P that most recently precedes pt1 and then determining whether or not the end of that token can possibly follow pt2. The most expensive part of this involves determining good estimates of the distance separating pairs of points in the time map, and this operation is optimized using the techniques of Section 4.4.3.

Incremental reorganization of the time map is an important consideration in efficiently maintaining the data base to support the above sort of queries. The techniques of temporal reason maintenance (Section 4.3) see to it that only those parts of the data base affected by the most recent modifications have to be updated. The temporal reason maintenance algorithm effectively performs a sweep forward in time selectively updating only those aspects of the time map as are indicated by the dependencies recorded in time map protections. The methods described in Chapter 4 for propagating constraints (Section 4.3.3), monitoring protections (Section 4.3.2), and resolving apparent contradictions (Section 4.3.1) make this incremental reorganization possible. Temporal reason maintenance also plays an important role in noticing certain critical consequences that follow from modifications to the data base. Using special programs called change driven interrupts (Sections 3.2.5 and 3.5.2), an application program can specify the type of consequences it is interested in being alerted to and exactly what response is to be made assuming that those consequences manifest themselves. The basic technique is quite common in AI languages [Hewitt 71] [McDermott 73]. Its application to reasoning in time maps provides a natural generalization of the notion of a critic [Sacerdoti 77] that responds to the detection of interactions (or in the case of the TMM, protection failures).

It is ultimately up to the reader to determine whether or not the TMM represents an efficient and natural extension of classical predicate-calculus data base techniques. Without access to the program it will be difficult to assess its efficiency. I would hope, however, that the explication of implementation details and the discussion of the assumptions made in guiding (Section 4.4.3) and cutting off (Section 4.3.3) search will at least convince the reader that such implementations are feasible. Versions of the TMM have already been employed in planning applications [Miller 85a], and I hope to have a complete distribution version within the coming year. As for the naturalness of the notation and the range of functionality supported, the examples of Chapter 3 should provide a basis for the reader to make his own judgements. Admittedly there is a great deal of research that has yet to be done. The hope is simply that the techniques presented here provide a reasonable start. The next section suggests certain areas for further research.

6.2 Problems and possible extensions

There are a lot of problems and areas for further research that were mentioned in the course of this dissertation. In this section, I will simply review these problems and refer the reader back to the text in which the associated issues came up.

Time maps were originally developed for coping with large amounts of temporal information. The time maps in the applications explored so far have yet to exceed 300 time tokens. One ideal application of time maps would be to manage a large data base of facts dealing with the correspondence of a journal editor or publishing house. The time map would monitor publication deadlines, automatically generate nasty letters to laggard reviewers and writers, and notice when some reviewer was inadvertently swamped with more work than he or she could possibly handle. Such a data base for a large journal would easily involve thousands of time tokens and hundreds of auto-projection rules and change-driven interrupts. Without some additional strategies for partitioning large time maps and guiding search, I expect that the current implementation would be swamped by such applications. The techniques described in Section 4.4.3 just begin to exploit the available structure for organizing time maps. Hierarchical organizations based on the calendar (days, weeks, months, years) suggest simple but highly effective techniques that could be used to partition time maps. Another problem involved in dealing with large time maps involves setting up data dependencies to detect apparent contradictions (Section 4.3.1). Setting up such predictions for all possible pairs of contradictory tokens is prohibitive. Luckily, in most cases it's hardly necessary. Most information in a temporal data base is unlikely to ever change. If the galley proofs for a certain manuscript are sent to the copy center, then that fact is history. Every subsequent action that results in a change in the location of those proofs needn't worry about whether the action occurs before or after the proofs were sent to the copy center. It's highly unlikely that the data entry operator was mistaken about the time that the proofs were sent to the copy center. It would seem that the problem of selectively setting up data dependencies to detect and resolve apparent contradictions can be handled efficiently in many applications.

In Section 4.7.2, I discussed the problem of reasoning about overlapping tokens of the same type. If I know that there's a light in the kitchen from 8:00 AM until midnight and a light in the living room from midnight until noon, then I know there is a light on in the house 24 hours a day. Currently the time map is not capable of making the general observation that if P is true throughout the interval from pt1 to pt2, P is true throughout the interval from pt3 to pt4, and pt3 is between pt1 and pt2, then P is true throughout the interval from pt1 to pt4. Getting the time map to handle this sort of inference appears to be one the simpler extensions of the TMM. Section 4.7.2 described a solution to this problem, but it has yet to be implemented.

Section 4.7.1 described a method for dealing with nonmonotonic inferences of the form, if you have no reason to believe `(not P)` is true anywhere in the interval from pt1 to pt2, then you are licensed to believe P throughout this interval. Writing code to handle queries of the form `(M (tt pt1 pt2 P))`, and monitor the continued validity of the underlying default assumptions is not particularly difficult. The TMM supports such inferences using what are called anti-protections. However, there are a number of efficiency considerations that have to be dealt with before this sort of inference can be relied upon not to bog down processing. In particular anti-protections must keep track of *all* tokens of a particular type. There must be some method for concentrating on a restricted subset of the set of all tokens of a type. In this regard, the issues involved in handling anti-protections are much the same as those involved in setting up dependencies to detect and resolve apparent contradictions (see above). I am convinced that some method for efficiently performing nonmonotonic inferences of the form `(M (tt pt1 pt2 P))` can added to the TMM.

The current techniques for reasoning about alternatives are the newest addition to the time map routines, and as such the least is known about their general performance. Constraint propagation in time maps involving gating objects (see Section 4.6.2) can lead to an exponential (in the number of gating objects) proliferation of search paths in the worst case.

In a sense this is unavoidable. The expectation (borne out by all of my experiece to date) is that in most applications this exponential blowup will not occur. Even the general methods for handling contexts [McDermott 83] have exponential worst case behavior, but, again, this just doesn't appear to be a problem in practice. Nonetheless, this sort of issue requires more attention if reasoning about multiple PWDs simultaneously is to be demonstrated as a viable alternative to sequential methods.

The section containing the proof of correctness for the temporal reason maintenance system (Section 4.3.5) provided a fairly restrictive criterion for guaranteeing termination. There are two methods whereby the user might be freed from adhering to such a criterion. First, it is possible that less restrictive criteria that still guarantee termination can be found. In fact, such criteria are already available. However, these less restrictive criteria are more complicated and hence more difficult for the programmer to adhere to. The second method for relaxing termination criterion would be to make it the system's responsibility to detect and recover from from circular dependencies that would normally lead to the current algorithm failing to terminate. The latter might involve considerable processing overhead (I suspect that the general problem of detecting time map dependency circularities leading to nontermination is intractable). I hope that it will be easier to formulate less restrictive criterion. Perhaps something akin to the *no-odd-loops* criterion for static data dependency systems [Charniak 80] can be formulated for time maps. In any case, there is a lot more to be done to settle these issues.

There are occasions in which it would be convenient to handle apparent contradictions involving tokens whose schemata (types) contain variables or terms whose properties change over time (Section 4.3.1). For instance, `(color block37 c1)` contradicts `(color block37 c2)` even though all we currently know about `c1` and `c2` is that `(member c1 !<red green)` and `(= c2 blue)`. Noticing such contradictions might be handled using various nonmonotonic inference techniques (*e.g.*, *abductive unification* [Charniak 86]), but keeping track of exactly which pairs of tokens are contradictory under the addition and deletion of facts is likely to be expensive. In planning, this sort of inference may play an important role in the process of managing the properties of partially instantiated parameters (or *script variables*) introduced in plan expansion [Stefik 81]. For instance, suppose that the robot has decided to service either `lathe7` or `lathe34`. One consequence of this is that the robot will be in either `room41` or `room17`. From this the planner should be able to conclude that the robot will no longer be in `room5`. Furthermore, if the success of a particular plan depends upon

the robot being in `room5`, then the planner should be made aware of the fact that this plan is endangered. I have very little in the way of suggestions at this point. I find the underlying functionality suspiciously open-ended.

Chapter 7

References

[Allen 83] Allen, James, *Maintaining knowledge about temporal intervals*, Communications of the ACM, 26/11 (1983), pp. 832–843.

[Allen 85] Allen, James F. and Hayes, Patrick J., A Common-Sense Theory of Time, *Proc. IJCAI 9, Los Angeles, Ca.*, IJCAI, 1985.

[Bolour 82] Bolour, A., Anderson, T.L., Dekeyser, L.J., Wong, H.K.T., *The Role of Time in Information Processing: A Survey*, SIGART Newsletter, /January (1982), pp. 28–48.

[Bowen 81] Bowen, D. L., Byrd, L., Pereira, L. M., Pereira, F. C. N., and Warren, D. H. D., *PROLOG on the DEC System-10 User's Manual*, Technical Report, University of Edinburgh, Department of Artificial Intelligence, 1981.

[Bruce 72] Bruce, B, *A model for temporal reference and its application in a question answering program*, Artificial Intelligence, 3 (1972), pp. 1–25.

[Chapman 85] Chapman, David, *Planning for Conjunctive Goals*, Technical Report AI-TR-802, MIT AI Laboratory, 1985.

[Charniak 80] Charniak, Eugene, Riesbeck, Christopher K. and McDermott, Drew V., *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, 1980.

[Charniak 85] Charniak, Eugene and McDermott, Drew V., *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Co., 1985.

[Charniak 86] Charniak, Gene, *Motivation Analyisis, Abductive Unification, and Non-Monotonic Equality*, Artificial Intelligence, (1986).

[Cheeseman 84] Cheeseman, Peter, A Representation of Time for Automatic Planning, *Proc. IEEE Int. Conf. on Robotics*, IEEE, 1984.

[Clocksin 84] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, Springer-Verlag, 1984.

[Daniel 83] Daniel, Lesley, Planning and operations research, *Artificial Intelligence: Tools, Techniques, and Applications*, Harper and Row, New York, 1983.

[Davis 82] Davis, Randall, Teiresias: Applications of Meta-Level Knowledge, Davis, Randall and Lenat, Douglas B. eds., *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill International Book Company, 1982, pages 227–490.

[Davis 85] Smith, David E. and Genesereth, Michael J., *Ordering Conjunctive Queries*, Artificial Intelligence, 26/2 (1985), pp. 171–216.

[Dean 83] Dean, Thomas, *Time Map Maintenance*, Technical Report 289, Yale University Computer Science Department, 1983.

[Dean 84] Dean, Thomas, Planning and Temporal Reasoning under Uncertainty, *Proc. IEEE Workshop on Principles of Knowledge-Based Systems*, IEEE, 1984.

[Dean 85] Dean, Thomas, Temporal Reasoning Involving Counterfactuals and Disjunctions, *Proc. IJCAI 9, Los Angeles, Ca.*, IJCAI, 1985.

[deKleer et al 77] de Kleer, Johan, Doyle, Jon, Steele, Guy L., and Sussman, Gerald J., *Explicit control of reasoning*, SIGPLAN Notices, 12/8 (1977).

[deKleer 78] de Kleer, Johan., Doyle, Jon., Rich Charles., Steele, Guy L., & Sussman, Gerald J., *AMORD A Deductive Procedure System*, Technical Report AI TR-435, MIT AI Laboratory, 1978.

[deKleer 82] deKleer, Johan and Brown, J.S., Foundations of Envisioning, *Proc. AAAI-82, Pittsburgh, Pa.*, AAAI, 1982.

[deKleer 84] deKleer, Johan, Choices Without Backtracking, *Proc. AAAI-84, Austin, Texas.*, AAAI, 1984.

[Dennett 84] Dennett, Daniel C., *Elbow Room: The Varieties of Free Will Worth Wanting*, MIT Press, 1984.

[Doyle 79] Doyle, Jon, *A truth maintenance system*, Artificial Intelligence, 12/3 (1979), pp. 231–272.

[Doyle 80] Doyle,Jon, *A Model for Deliberation, Action, and Introspection*, Technical Report AI TR-581, MIT AI Laboratory, 1980.

[Fahlman 79] Fahlman, Scott, *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press, 1979.

[Fikes 71] Fikes, Richard and Nilsson, Nils J., *STRIPS: A new approach to the application of theorem proving to problem solving*, Artificial Intelligence, 2 (1971), pp. 189–208.

[Findler 71] Findler, N. and Chen, D., On the problems of time retrieval, temporal relations, causality, and coexistence, *Proc. IJCAI 2, London, England*, IJCAI, 1971.

[Firby 85] Firby, R. James, Dean, Thomas L., Miller, David P., Efficient Robot Planning with Deadlines and Travel Time, *Proceedings of the 6th International Symposium on Robotics and Automation, Santa Barbara, Ca.*, IASTED, 1985.

[Forbus 84] Forbus, Kenneth, *Qualitative Process Theory*, Artificial Intelligence, 24 (1984), pp. 85–168.

[Forgy 81] Forgy, Charles L., *OPS5 User's Manual*, Technical Report CMU-CS-81-135, Carnegie-Mellon University Department of Computer Science, 1981.

[Fox 82] Fox, M.S., Allen, B., and Strohm, G., Job-Shop Scheduling: An Investigation in Constraint-Directed Reasoning, *Proc. AAAI-82, Pittsburgh. Pa.*, AAAI, 1982, pp. 155–158.

[Garey 79] Garey, Michael R. and Johnson, David S., *Computing and Intractibility: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[Goldstein 75] Goldstein, Ira P., NUDGE: A Knowledge-Based Scheduling Program, *Proc. IJCAI 4, Tbilisi, Georgia, USSR*, IJCAI, 1975.

[Hayes 79] Hayes, Patrick, The Naive Physics Manifesto, Michie, Donald ed., *Expert Systems in the Microelectronic Age*, Edinburgh University Press, 1979.

[Hendrix 73] Hendrix, Gary, *Modeling simultaneous actions and continuous processes*, Artificial Intelligence, 4 (1973), pp. 145–180.

[Hewitt 71] Hewitt, Carl, PLANNER: A Language for Proving Theorems in Robots, *Proc. IJCAI 2, London, England*, IJCAI, 1971.

[Hughes 68] Hughes, G.E. and Cresswell, M.J., *An Introduction to Modal Logic*, Methuen and Co Ltd., London, 1968.

[Kahn 77] Kahn, Kenneth and Gorry, G. Anthony, *Mechanizing Temporal Knowledge*, Artificial Intelligence, 9 (1977), pp. 87–108.

[Lumelski 85] Lumelski, V. J., Path planning with Uncertainty, *Proc. The Fourth Yale Workshop on Applications of Adaptive Systems Theory*, Yale Unversity, 1985.

[Malik 83] Malik, Jitendra, and Binford, Thomas O., Reasoning in Time and Space, *Proc. IJCAI 8*, IJCAI, 1983, pp. 343–345.

[Martins 83] Martins, J.P. and Shapiro, S.C., Reasoning in Multiple Belief Spaces, *Proc. IJCAI 8, Karlsruhe, West Germany*, IJCAI, 1983.

[McAllester 80] McAllester, David A., *The Use of Equality in Deduction and Knowledge Representation*, Technical Report 550, MIT AI Laboratory, 1980.

[McAllester 82] McAllester, David A., *Reasoning Utility Package User's Manual*, Technical Report 667, MIT AI Laboratory, 1982.

[McCarthy 69] McCarthy, John and Hayes, Patrick J., Some Philosophical Problems from the Standpoint of Artificial Intelligence, Meltzer, B. and Michie, D. eds., *Machine Intelligence 4*, Edinburgh University Press, 1969.

[McDermott 73] McDermott, Drew V. and Sussman, Gerald J., ***The Conniver Reference Manual,*** Technical Report 259, MIT AI Laboratory, 1973.

[McDermott 80] McDermott, Drew V. and Doyle, Jon, ***Non-monotonic logic I,*** Artificial Intelligence, 13/1,2 (1980), pp. 41–72.

[McDermott 82] McDermott, Drew V., ***A temporal logic for reasoning about processes and plans,*** Cognitive Science, 6 (1982), pp. 101–155.

[McDermott 83] McDermott, Drew V., ***Contexts and data dependencies: a synthesis,*** IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-5/3 (1983), pp. 237–246.

[McDermott 84] McDermott, Drew V. and Davis, Ernest, ***Planning routes through uncertain territory,*** Artificial Intelligence, 22 (1984), pp. 107–156.

[McDermott 85] McDermott, Drew V., ***The DUCK Manual,*** Technical Report 399, Yale University Computer Science Department, 1985.

[Miller 83] Miller, David P., ***Scheduling Heuristics for Problem Solvers,*** Technical Report 264, Yale University Computer Science Department, 1983.

[Miller 85a] Miller, David P., Firby, R. James, Dean, Thomas L., Deadlines, Travel Time, and Robot Problem Solving, ***Proc. IJCAI 9, Los Angeles, Ca.,*** IJCAI, 1985.

[Miller 85b] Miller, David P., ***Planning by Search Through Simulations,*** Technical Report XXX, Yale University Computer Science Department, 1985.

[Minsky 81] Minsky, Marvin, A Framework for Representing Knowledge, Haugeland, John ed., ***Mind Design,*** MIT Press, 1981.

[Moore 75] Moore, Robert C., ***Reasoning from Incomplete Knowledge in a Procedural Deduction System,*** Technical Report AI-TR-347, MIT AI Laboratory, 1975.

[Nilsson 80] Nilsson, Nils J., ***Principles of Artificial Intelligence,*** Tioga Publishing Company, 1980.

[Pednault 85] Pednault, Edwin P. D., *Preliminary Report on a Theory of Plan Synthesis*, Technical Report 358, Artificial Intelligence Center SRI International, 1985.

[Pople 73] Pople, H., On the Mechanization of Abductive Logic, *Proc. IJCAI 3, Menlo Park, Ca.*, IJCAI, 1973, pp. 147–152.

[Sacerdoti 74] Sacerdoti, Earl, *Planning in a Hierarchy of Abstraction Spaces*, Artificial Intelligence, 7/5 (1974), pp. 231–272.

[Sacerdoti 77] Sacerdoti, Earl, *A Structure for Plans and Behavior*, American Elsevier Publishing Company, Inc., 1977.

[Shoham 85a] Shoham, Yoav and Dean, Thomas, Temporal Notation and Causal Terminology, *Proc. Seventh Annual Conference of the Cognitive Science Society*, Cognitive Science Society, 1985.

[Shoham 85b] Shoham, Yoav, *Time and Causation from the Standpoint of Artificial Intelligence*, Technical Report XXX, Yale University Computer Science Department, 1986.

[Simmons 83] Simmons, Reid G., *Representing and Reasoning About Change in Geologic Interpretation*, Technical Report AI-TR-749, MIT AI Laboratory, 1983.

[Simon 81] Simon, Herbert A., *The Sciences of the Artificial*, MIT Press, 1981.

[Smith 83] Smith, Stephen F., *Exploiting Temporal Knowledge to Organize Constraints*, Technical Report CMU-RI-TR-83-12, Carnegie-Mellon University Intelligent Systems Laboratory, 1983.

[Stallman 79] Stallman, Richard M. and Sussman, Gerald J., Problem Solving About Electrical Circuits, Winston, Patrick H. and Brown, Richard H. eds., Volume 1: *Artificial Intelligence: an MIT Perspective*, MIT Press, 1979, pages 33–92.

[Stefik 81] Stefik, Mark J., *Planning with Constraints*, Artificial Intelligence, 16/2 (1981), pp. 111–140.

[Sussman 71] Sussman, Gerald J., Winograd, Terry, and Charniak, Gene, *Micro-Planner Reference Manual*, Technical Report 203, MIT AI Laboratory, 1971.

[Sussman 75] Sussman, Gerald J., *A Computer Model of Skill Acquisition*, American Elsevier Publishing Company, Inc., 1975.

[Swartout 83] Swartout, William, The GIST Behavior Explainer, *Proc. AAAI-83, Washington, D.C.*, AAAI, 1983, pp. 402–407.

[Tate 77] Tate, Austin, Generating Project Networks, *Proc. IJCAI 5, Cambridge, Ma.*, IJCAI, 1977.

[Vere 83] Vere, Steven, *Planning in Time: Windows and Durations for Activities and Goals*, IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-5/3 (1983), pp. 246–267.

[Vere 85] Vere, Steven, Splicing Plans to Achieve Misordered Goals, *Proc. IJCAI 9, Los Angeles, Ca.*, IJCAI, 1985.

[Vilain 82] Vilain, Marc, A System for Reasoning About Time, *Proc. AAAI-82, Pittsburgh, Pa.*, AAAI, 1982.

[Wilensky 83] Wilenskey, Robert, *Planning and Understanding*, Vanity Press, 1983.

[Wilkins 84] Wilkins, David, *Domain Independent Planning: Representation and Plan Generation*, Artificial Intelligence, 22/3 (1984), pp. 269–302.

[Wilkins 85] Wilkins, David, *Recovering from Execution Errors in Sipe*, Computational Intelligence, 1/1 (1985), pp. 33–45.

END

FILMED

12-85

DTIC