1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

**LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE**

Metaplanning Using Time-Relation Constraints and Assumption-Based Reasoning

By
Mark Waldvogel, B.C.S.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
April 1, 1986

© copyright
1986, Mark Waldvogel

2

• Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-29880-4

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis

**"Metaplanning Using Time-Relation Constraints and
Assumption-Based Reasoning"**

submitted by Mark Waldvogel, B. C. S.,
in partial fulfilment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Chairman, School of Computer Science
Carleton University

Date

Abstract

Planners (and AI systems in general) typically have ad hoc or unrealistic temporal representations. We describe a more sophisticated world model, built upon the expressive time-interval based temporal representation of Allen[1983a,1984,1985a] and the assumption-based reasoning methods of DeKleer[1984a,1986]. The time-relation constraints of Allen's representation are used to describe when facts, actions and events occur. The addition of assumption-based reasoning allows reasoning about uncertain future facts, actions or events. We base the design of a planner to use this representation on the *metaplanning* model of planning proposed by Wilensky[1983]. The most important *metagoal* is the *resolve-conflict* goal, which is generated to solve planning conflicts. The combination of our temporal-representation and assumption-based reasoning allows effective reasoning about planning conflicts. This includes the ability to plan to prevent bad facts, even if these facts are not the result of actions by the planner.

Acknowledgements

I would like to thank my adviser Wilf Lalonde for his support and advice. I would also like to thank Franz Oppacher, for valuable discussions; George Wang, for his contribution towards the preparation of the thesis; and Gerhard Roth, for commenting on earlier drafts of the thesis and providing moral support. My family deserves thanks for their love and support that helped keep me going. My wife Terry deserves special thanks for her cheering me up when I was discouraged and making sacrifices so that I could pursue this work.

This work was partially supported by NSERC and DREA grants.

Dedication

This thesis is dedicated to Terry.

Table of Contents

Acceptance Sheet	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
Table of Contents	vi
 Chapter 1 Introduction	
1.1 Background	1
1.1.1 Planning	1
1.1.2 Metaplanning	2
1.1.3 Temporal Reasoning	3
1.1.4 Assumption-based Reasoning	4
1.2 Goals	5
1.3 Knowledge Representation	5
1.4 System Description	7
1.5 Examples	11
1.6 Limitations and World Assumptions	12
1.7 Thesis Overview	13
 Chapter 2 Related Work	
2.1 The STRIPS Planner and its Descendants	15
2.2 The NOAH Planner and its Descendants	17
2.3 Metaplanning	22
2.4 Temporal Representations	29
2.5 Causality and the Frame Problem	32
2.6 Nonmonotonic Reasoning	35
2.7 Modelling Future Uncertainty	39
2.8 Prevention	40
2.9 Execution and Replanning	42
 Chapter 3 Knowledge Representation	
3.1 Introduction	45
3.2 Objects, Constants and Variables	47
3.3 Time Relations	48
3.4 Defining Object Types	51
3.5 Defining Value Types	53
3.6 Defining Physical Object Types	53
3.7 Defining Action and Fact Types	54
3.8 Defining Fact Types	55

3.9 Defining Mutually Exclusive Facts	56
3.10 Defining Command Types	56
3.11 Goal Types	57
3.12 Defining Event Types	58
3.13 Defining Plan Types	65
3.14 Define-Planfor	69
3.15 Defining Bad Facts	71
 Chapter 4 Assumption-based Reasoning	
4.1 The Fundamentals	73
4.2 Kinds of Derivations	76
4.2.1 Constant and Variable Derivations	76
4.2.2 Constraint Derivations	77
4.2.3 Other Relation Derivations	78
4.3 Making Assumptions	78
4.3.1 Variable Assumptions	79
4.3.2 Constraint Assumptions	79
4.3.3 Other Assumptions	80
4.4 Top-level Assumptions	81
4.5 Detecting All the Nogood Assumption Sets	81
4.6 Removing Assumptions from Assumption Sets	84
4.7 Degree of Belief	84
 Chapter 5 The Inference Engine	
5.1 The Inference Loop	87
5.2 Pseudo Rules	88
5.3 Unification Assumptions	89
 Chapter 6 Projection	
6.1 Constraint Propagation	92
6.1.1 Time-Relation Inferences	92
6.1.2 Inequality and Instance Inferences	97
6.2 Imposing Time-Relation Constraints Between Objects	99
6.3 Event Generation	100
6.4 Handling Non-Planning Conflicts	101
6.5 Negation Inferences	103
6.6 Predicting Bad Facts	104
 Chapter 7 Plan Generation and Selection	
7.1 Generating Plans	105
7.2 Plan Instantiation	107
7.3 Solving Bind Goals and Achieve Goals	109
7.4 Generating Prevention Plans	111
7.5 Planning Conflicts	112
7.6 Detecting Planning Conflicts	114

7.7 Solving Resolve-Conflict Goals	118
7.8 Solving Make-Disjoint Goals	121
7.9 Plan Selection	121
7.10 Handling Blocked Goals	123
7.11 Task Network Consistency	124
7.12 Copying Choice Variables	126
7.13 Replanning	126
Chapter 8 Command Execution and Event Detection	
8.1 The Command Executor	128
8.2 The Event Detector	129
8.3 Matching Hypothetical Facts Against Observed Facts	130
8.4 Inferring That Event Variables Describe Actual Events	131
Chapter 9 Examples	
9.1 The Door Example	132
9.1.1 The Problem	132
9.1.2 The Domain Definition	133
9.1.3 Generating the Task Network	135
9.1.4 Normal Execution	139
9.1.5 Replanning	144
9.2 The Nell and Train Example	147
9.2.1 The Problem	147
9.2.2 The Domain Definition	149
9.2.3 Generating the Task Network	153
9.2.4 Execution	160
9.3 The Raincoat Example	162
9.3.1 The Problem	162
9.3.2 The Domain Definition	163
9.3.3 Generating the Task Network	168
9.3.4 Execution	179
9.3.5 Replanning	180
Chapter 10 Future Work and Conclusions	
10.1 Summary	182
10.2 Implementation	183
10.3 Future Work	185
10.4 Conclusions	186
References	188
Appendix A: Index of Goal and Plan Types	195

Chapter 1

INTRODUCTION

1.1 Background

1.1.1 Planning

Planning is typically defined as the problem of finding a sequence of actions that could be performed in order to transform some initial state into a final goal state. Occasionally the related problem of how to execute plans is also considered part of planning. Plan execution is only interesting when the actions do not have the expected effects, or unexpected events occur. We will consider both plan generation and plan execution.

Most conventional planners are patterned after the NOAH planning system (Sacerdoti[1977]). They are given an initial world state and a possibly conjunctive goal to achieve from this initial state. They produce a partially-ordered network of actions called a task network. The task network initially consists of a single plan node to solve the problem, and is successively refined to lower levels of detail. At each level the planner checks for interactions between unordered parts of the network, and may add ordering constraints to take advantage of helpful interactions or to avoid conflicts. New goals may be produced at each level which then need to have plans proposed to solve them. Goals can also be considered to be solved if they hold in the initial world model and no preceding action in the network deletes this initial world fact. The task network is fully elaborated, solving the problem, when there are no more goals requiring plans. Such planners are examples of

hierarchical non-linear planners. They are based on the philosophy of least commitment with respect to ordering actions.

1.1.2 Metaplanning

Wilensky[1983] makes criticisms of NOAH-type planners including :

- They do not do not reason about complex planning interactions adequately.
- They have no notion of bad situations to be avoided.
- They cannot generate their own goals.
- They are designed to solve a single complex goal, whereas a planner should be able to handle multiple competing goals. This is particularly important when the planner can generate its own goals.

He defines *metaplanning* as the process of solving planning problems, such as planning conflicts, using metaplans and metagoals. The same basic planning machinery is used to find metaplans for metagoals as is used to find regular plans for regular goals. This allows for much more flexible reasoning about the planning process, and in particular about interactions between plans. Wilensky proposes a model in which top-level goals are generated due to actual facts which occur, to hypothetical facts which are projected or to planning problems which are detected. Plans are found to solve any outstanding goals. When no goals require plans, the planner can start executing its actions. This model allows for integrated planning and execution.

The ideas proposed by Wilensky[1983] are useful ones, but there is a fairly wide gap between the issues he discusses and what his program (PANDORA, Faletti[1982], Wilensky[1983]) can do. PANDORA uses an adhoc

representation of time which puts strong limits on its applicability. It is not clear whether PANDORA can solve the standard complex subgoal interaction problems which NOAH-type planners can. Wilensky[1983] avoids working on such *blocks world* problems as a matter of principle. His examples are typically expressed in English, making it hard to follow how he really intends them to be solved because the same English description could have many alternative encodings in a knowledge representation.

1.1.3 Temporal Reasoning

McDermott[1982, 1985] and Allen[1983a, 1983b, 1984, 1985b] criticize the temporal representations used in most AI systems (including NOAH-based planners) as insufficient to handle a more realistic world model. For example, the NOAH-type planners assume that all changes in the world are due to the actions of the planner. McDermott and Allen propose first-order temporal logics which are more expressive than conventional temporal representations and which can be used to reason with a more realistic world model. They describe more sophisticated plans and goals which can be expressed using their representation. In contrast with NOAH-type planners where a goal state only needs to hold for an instant, planners based on the temporal logics of McDermott or Allen can describe a fact that should hold over a desired interval.

Allen proposes a temporal representation in which the primitive temporal unit is the time-interval. Facts, events and actions are associated with time-intervals. The temporal relations between facts, events and actions are expressed in terms of the relations between their time-intervals. Allen identifies seven primitive mutually exclusive interval relations and their inverses. The

time-relation between two intervals is described in terms of the set of possible primitive relations which might hold between them. This provides an effective and powerful way of describing disjunctive temporal information. A mechanism for computing transitive relations between each set of three intervals is described in Allen[1983a].

Allen's interval-based temporal representation is an alternative to the *state* based model used by most AI systems. States represent instantaneous snapshots of the universe.

1.1.4 Assumption-Based Reasoning

To reason about an uncertain future (or past or present), it is necessary to be able to make tentative inferences which can be later withdrawn given more information. This kind of reasoning is called *nonmonotonic reasoning*, since the number of assertions believed does not grow monotonically, as in classical logic. Programs to support nonmonotonic reasoning are typically called Truth Maintenance Systems. We base our design on the assumption-based reasoning approach described by DeKleer[1984a,1986]. It is an alternative to the conventional TMS of Doyle[1979], which could be called justification-based reasoning. Doyle's TMS is over-zealous in contradiction avoidance and has problems associated with switching contexts. DeKleer's approach does not require a single current consistent context, but instead allows derivations of assertions based on alternate sets of assumptions (contexts) to be made in parallel. When a contradiction is detected, the assumption set underlying the contradiction is marked *nogood*. It is not necessary to backtrack to revoke an assumption underlying the contradiction. Derivations that are based on

assumption sets that are supersets of *nogood* assumption sets are deleted from the knowledge base.

In approaches which require global consistency to be maintained, the reasoning system has to make arbitrary choices to decide which assumption underlying a contradiction should be *revoked*. This is particularly undesirable when the truth of the assumptions that the reasoner has to choose between are not under the planner's control; if they are *guesses* as opposed to *choices*. With DeKleer's approach there is no need to make such arbitrary decisions.

The planner does not want to have to consider all possible plans for all its goals in parallel. Some modifications to the description of DeKleer[1984a] are required so that we can keep only one plan under consideration per goal and so that we can maintain consistency in the task network.

1.2 Goals

The goals of this thesis are to:

1. Propose a method for reasoning about an uncertain future using Allen[1983a]'s time-relation reasoning and DeKleer[1984a, 1986]'s assumption-based reasoning. It needs to make distinctions between hypothesized and actual facts and events since we consider the problem of observing changes in the world.
2. Describe how a planner based on Wilensky[1983]'s model could metaplan using this more sophisticated world model.

1.3 Knowledge Representation

It is possible to create programs which manipulate representations of

events, actions and facts in an adhoc way and solve impressive examples. The problem with such programs is that it is difficult to predict which examples the programs cannot solve, since they are essentially coded with a small number of examples in mind. Our approach is the opposite. We have designed a representational language in which many important distinctions in meaning can be made. We then show how a planner can take advantage of this representation to do more sophisticated planning than is possible with simpler more adhoc representations.

Our knowledge representation is built upon the foundation of the time-relation reasoning techniques of Allen[1983a,1983b,1984,1985b] and the assumption-based reasoning techniques of DeKleer[1984a,1986]. In Allen's temporal representation, time-intervals are the primitive temporal units. This does not make the kinds of restrictive world assumptions of state-space based representations. By combining the techniques of Allen and DeKleer in a novel way, we are able to reason effectively about an uncertain future. This technique allows the system to hypothesize future events, and even try to prevent them. It does not support reasoning about *possible worlds* which are known not to correspond to the actual world.

Our terminology is to call events, facts, physical objects and actions *objects*.

We derive the assumptions under which object variables refer to actual objects. For example, a fact variable describes a hypothesized fact. It is asserted to refer to an actual fact under some assumptions. If the system discovers that the variable refers to an actual fact, the fact variable will become derived under the empty set of assumptions. If it is determined that the assumptions under which the variable is derived are contradictory, the fact

variable will be known not to represent an actual fact. No further inferences will be made using such an impossible fact, and all temporal constraints which mention its time-interval will be automatically deleted by the assumption-based reasoner. The sorts of distinctions we make between hypothesized occurrences and actual ones are typically not made by AI systems, most of which are concerned either with only actual or only hypothetical occurrences.

Unlike most approaches to representing an uncertain future, ours is not based on a branching future model. This suggests that our methods could be used to reason about an uncertain ~~past~~ and present as well. Branching future methods have difficulty reasoning about possible past events.

Our knowledge representation is particularly useful for detecting and reasoning about planning conflicts. Planning conflicts are contradictions whose assumptions describe a conflict between one or more plans, perhaps the persistences of background facts, and perhaps other types of assertions as well. The assumptions underlying contradictions can be efficiently examined to determine whether the contradiction is a planning conflict, and if so determine the minimum set of assertions which are the cause of the planning conflict.

Frequently the meaning of plans in AI programs is not well-defined, due to inadequate distinctions between what a plan asserts will occur, and when the plan should be used. We define exactly under what conditions a plan occurs, and distinguish between the assertion that it does occur, and that it is wanted to occur.

1.4 System Description

The high-level functional description of our proposed system is essentially.

the same as that of PANDORA (Wilensky [1983], Faletti[1982]). It has the following components :

1. The *Plan Generator*. Domain-dependent rules can be defined to generate new plans. Such plans could be generated to solve existing goals, or as top-level plans (which are to solve a *dummy* top-level goal). Domain-independent top-level resolve-conflict plans are generated to deal with planning conflicts.
2. The *Projector*. The Projector has several tasks. It propagates time-relation, inequality and type constraints on objects. It also predicts the occurrence of events and bad facts (which the planner should try to prevent), and projects the consequences of plans. Events may be predicted which are not the result of the planner's actions.
3. The *Plan Selector*. The Plan Selector selects plans to solve unplanned goals.
4. The *Command Executor*. The system executes the commands of its task network only once the network is fully elaborated. The Executor makes a command actually start executing. If there is more than one executable command it decides between them.
5. The *Event Detector*. The Event Detector notices actual changes in the world and updates the knowledge base accordingly.

We will classify these functional components into three classes: *Inference Components*, *Choice Components* and *Observation Components*.

1. The Plan Proposer and the Projector are *Inference Components*. These components can be described via inference rules. Chapter 3 describes the form of these rules.

2. The Plan Selector and the Command Executer are *Choice Components*.

The current choice options are updated due to changes in the Knowledge Base.

3. The Event Noticer is an *Observation Component*. It is concerned with updating the World Model to reflect what actually occurs. It detects when commands start and stop executing, as well as when fact changes occur. Unexpected fact changes may occur and event predictions may fail.

We can distinguish between the Plan Selector and the Executer on the grounds that the Plan Selector updates the Knowledge Base when it makes a choice, whereas the Executer makes a change in the physical world and notifies the Event Detector. The following diagram describes the system:

Figure 1.1 System Diagram

In the implementation of this system that we envisage, a user at a console would play the role of environment/sensors/effectors. The user could also perhaps make the choices for the Plan Selector and/or the Command Executor. This could allow the investigation of different control strategies. We do not address in any detail the issue of how the Plan Selector and Command

Executor should choose between alternative choices. The user can type in new facts to indicate changes in the real world. These facts may be arbitrary. They do not have to be expected by the planner, and may even contradict expectations. The user also indicates when commands (primitive actions) stop executing. In this way, it is easy for the user to describe changes which occur while a command is still executing.

The top-level loop of the system is essentially the same as that of Wilensky[1983]:

```
LOOP
  IF there is an event detected by the event detector THEN
    infer real world changes
  ELSE
    IF there is a goal requiring a plan to solve it THEN
      select a plan for it
    ELSE
      IF there is at least one executable command THEN
        select an executable command and execute it
      ENDIF
    make all forward inferences
  ENDLOOP
```

1.5 Examples

We do not have a working system, but we do show in some detail how our proposed system would tackle some planning problems. The first example we describe is a very simple one which is only given to give an idea of how our system replans when unexpected events occur.

We show how our techniques could be used to solve the *Nell and Train* example of McDermott[1982]. In this example, Nell is on the train tracks and a train is expected to arrive. The planner predicts that Nell will get killed if she is still on the tracks when the train arrives. This is a bad fact which the planner

then wants to prevent. A planning conflict is produced between the prevention plan and Nell getting killed. The planning conflict is described as containing the prevention plan and persistence assumptions which state that Nell is on the tracks when the train comes. A resolve-conflict goal is generated to resolve this conflict. It generates a plan to get Nell off the tracks before the train comes. This problem is difficult, since it involves trying to prevent an event which is not the result of actions of the planner. As far as we know, no implemented system can solve this problem, though McDermott[1982,1983] describes an approach.

We also illustrate how Wilensky[1983]'s *Raincoat* problem could be solved. In this example, the planner wants to go outside to fetch a newspaper. It is raining so the planner expects to get wet. Since the planner does not like to get wet, a prevent plan is generated to prevent the planner from getting wet. This will lead to a planning conflict and a resolve-conflict goal to resolve the planning conflict. In this case, the conflict is between the plan to fetch a newspaper, the plan to keep from getting wet, and the assumptions that the planner will be outside in the rain without a raincoat. This conflict is resolved by making the persistence of not wearing a raincoat terminate before the planner goes outside. To do this, the planner puts on a raincoat first.

1.6 Limitations and World Assumptions

To simplify the problem, we make some assumptions about the world and limit our representation. The restrictions which we impose are typically not necessary ones imposed by the principles of our low-level knowledge representation, and we will suggest ways in which some of them might be relaxed.

We will assume that the planner knows everything about the present state of the world. This is not strictly a consequent of our approach. It does simplify several aspects of the planner. For one thing, the planner does not have to worry about having to plan to obtain knowledge. It is also easier to tell when commands can be executed. It seems that it should be possible to extend our representation for reasoning about an uncertain past and present.

We will assume that the planner is the only agent. This does away with the need to model the beliefs of other agents.

We only represent temporal information associated with time-relations, and not durations or dates. Allen[1983a,1985a] suggests ways to perform such reasoning which is consistent with his time-interval based time-relation reasoning, so there is reason to believe that our approach should be extendable to handle such distinctions. The abilities to reason about durations seems vital for any useful planning system. Nevertheless, this ability is lacking in most existing planners.

1.7 Thesis Overview

Chapter two describes related work in AI. It provides the background for distinguishing what we are trying to do from the existing work.

Chapter three is a key chapter in which we describe our terminology for the various kinds of objects and relations used in our system, and how a domain is defined.

In chapter four we describe the details of our assumption-based reasoning mechanism.

In chapter five we describe briefly the inference loop. It also describes how

assumptions are generated during unification.

Chapter six describes the various non-planning inferences we attribute to the Projector. In it we describe how our time-relation reasoning works, the inequality and instance constraints which we impose, and how events are predicted. Our use of intersection intervals is an important extension to the time-relation reasoning provided by Allen[1983a].

Chapter seven describes how plans are generated and selected. It is a key chapter in order to understand the examples. The sections involving planning conflicts and the resolve-conflict goal are particularly important.

In chapter eight we describe how commands are chosen for execution, and how the knowledge base is updated when new fact changes are observed. It also discusses how we match hypothesized facts to actual ones.

Chapter nine contains the examples. It is probably useful to look at the *Door Example* first, since it has the simplest representation and describes the simplest problem. In this example we also go into the most detail concerning execution and replanning.

In chapter ten we present our conclusions and provide suggestions for further work.

Chapter 2

RELATED WORK

2.1 The STRIPS Planner and its Descendents

The classical planning problem is to find a sequence of actions which will transform an initial state into a state which satisfies some goals. STRIPS(Fikes[1971]) was an early planner. States in STRIPS are described by a set of simple fact terms. An action joins the state in which it is executed to a resulting state. STRIPS describes each action by its preconditions, addlist and deletelist. The preconditions are a list of facts which are required to be true in the state in which the action is executed. The resulting state is computed by taking the state in which the action was executed, deleting facts in the deletelist of the action, and adding facts in the addlist of the action. The following is a typical action definition:

```
(ACTION (PICKUP ?BLOCK)
  PRECONDITIONS: (CLEAR ?BLOCK) (HANDEEMPTY)
  DELETelist: (CLEAR ?BLOCK) (HANDEEMPTY)
  ADDLIST: (HOLDING ?BLOCK))
```

The STRIPS algorithm can be described by the following nondeterministic function. This function takes the initial-state and goal-state as arguments. It returns two values: a sequence of actions (plan) and the state which results if the sequence of actions is applied to the initial state.

```
DEFINE strips (initial-state goal-state)
  IF goal-state matches initial-state THEN
    RETURN( [ ] , initial-state) ; return the empty list and the initial state
  ELSE
    CHOOSE goal ← component of goal-state that does not match initial-state
```

```
CHOOSE action ← action whose addlist contains a term that matches goal
preconditions ← preconditions of action
plan1, state1 ← strips (initial-state, preconditions)
state2 ← state which results if action is executed in state1
plan2, state3 ← strips (state2, goal-state)
RETURN (append(plan1, [action], plan2), state3)
ENDIF
ENDDDEFINE
```

STRIPS does not record the reason why actions are proposed. It is therefore possible for later actions to defeat the purpose of earlier actions. This makes STRIPS very sensitive to the order in which the subgoals of a conjunctive goal are attempted. STRIPS tries to get around this problem by attempting every permutation of its goals. This is clearly an inadequate solution. The result is that STRIPS may produce solutions that are longer than necessary. For example a plan may have a pickup block1 action immediately followed by a putdown block1 action. There are also some very simple problems which STRIPS cannot solve.

Later planners modified STRIPS to detect and handle such bad subgoal interactions effectively without permuting the subgoals. HACKER (Sussman [1973]) created initially buggy plans using the linearity assumption; that is that the order of the goals does not matter. It then used critics to fix up the buggy plan. WARPLAN (Warren[1974], Coelho[1980]) and Waldinger[1977] used techniques of action regression and goal regression respectively. These techniques involve regressing actions or goals back in the plan sequence. (Nilsson [1980] describes regression in some detail.) They do not use heuristic rules like HACKER. INTERPLAN (Tate [1974]) introduced the notion of *goal-structure* which allowed more sophisticated reasoning about interactions than the regression techniques.

STRIPS does all its planning at a single level of abstraction. In a complex domain, such as the construction of a house, it would quickly flounder while planning how to hammer and saw individual boards. ABSTRIPS (Sacerdoti[1974]) was a system proposed to solve this problem. ABSTRIPS assigned each goal a level of criticality. The planner would solve the problem at the highest level of detail first, then successively solve it at lower levels of detail. At each level of detail, goals with levels of criticality below some value would be ignored. This methodology of solving a planning problem at successively lower levels of detail is called *hierarchical planning*. A problem with ABSTRIPS was that it had problems recovering if it detected problems at a lower level of detail which required modifying its plans at a higher level of detail.

We reject the approach of STRIPS-like planners on the grounds that a total ordering of states is not an adequate world model. (See section 2.4).

2.2 The NOAH Planner and its Descendents

NOAH (Network Of Action Hierarchies, Sacerdoti [1977]) was the first nonlinear planner. Planners like STRIPS are linear because they require that their actions be totally ordered. NOAH allows its actions to be partially ordered. NOAH is also a hierarchical planner, but it uses a different technique than ABSTRIPS.

NOAH represents the current state of planning with a *procedural net*. (the term *task network* is used by other researchers to mean the same thing). The procedural net is a graph structure whose nodes represent actions. There are levels of the net corresponding to different levels of abstraction. The actions

within a given level are partially ordered. Each action may refer to a set of child actions at the lower level that represent more detailed subactions. These subactions will achieve the effects of the parent action when executed in the specified order. There are also pointers from child actions to their parent action. NOAH actions have preconditions, addlists and deletelists like STRIPS actions.

NOAH uses critics to detect interactions between unordered nodes and add appropriate ordering constraints. We describe the major domain-independent critics below. Domain-dependent critics were also used.

The *Resolve Conflicts* critic detects if an action deletes an expression that is a precondition of an action on a parallel branch. The conflict may be resolved by requiring the endangered action to be achieved before the action that would delete the precondition.

A *double cross* occurs when two parallel actions deny each others preconditions. This kind of conflict cannot be resolved by any linearization of the parallel subplans. A *Resolve Double Cross* critic is used to resolve double crosses. The resolve double cross critic in NOAH attempts to change variable bindings to fix this problem, but gives up if the fix causes new problems.

The *Use Existing Objects* critic is used to bind variables in a plan to real objects when possible. For example, if a plan to paint the ceiling and a ladder specifies putting the ladder at some variable location Place001 to paint it, and at Under-Ceiling to paint the ceiling, the use existing objects critic would optimize the plan by replacing Place001 with Under-Ceiling.

When a plan contains two identical conditions, the *Eliminate Redundant Preconditions* critic can be used to optimize the plan by merging these two

conditions.

All critics must be applied at one level before NOAH can expand the plan to the next lower level. Nilsson [1980] describes why this is essential. NOAH starts with a single action node which represents the entire problem to be solved. It refines the plan level by level, applying critics at each level. The refinement continues until the procedural network reaches the desired level of detail (typically the most primitive level in which no nodes can be decomposed).

The algorithm (as described in Sacerdoti[1977]) is:

1. Expand the most detailed plan in the procedural net. This will have the effect of producing a new, more detailed plan.
2. Criticize the new plan, performing any necessary reordering or elimination of redundant operations.
3. Go to step 1.

NOAH is based on the philosophy of least commitment. It tries to postpone all ordering decisions as long as possible, to avoid making an early wrong decision.

Many deficiencies in NOAH were improved upon in later systems based on the NOAH paradigm. One of NOAH's problems is that it expands new levels irrevocably. It can neither backtrack through plan choices, nor through orderings imposed by critics. NONLIN (Tate[1977]) extended NOAH to handle backtracking, as did most later NOAH-based planners.

Another problem with NOAH is that it does not allow flexible specification of how long effects should hold in a plan. All effects of actions were required to hold until the end of their parent action. NONLIN (Tate [1977]) used the notion

of goal structure to remedy this problem. It modified the use of goal structure in the linear planner INTERPLAN (Tate[1974]).

-NOAH, like STRIPS, has only one type of precondition. The preconditions automatically become goals to be solved. NONLIN and SIPE (Wilkins[1984,1985]) distinguish between the goal preconditions and applicability conditions which must hold for the plan to be applicable, but which should not themselves become goals. This allows the description of actions to be performed in adverse conditions. For example, we might have a jump-out-of-window plan to exit a building, but which is only applicable if the building is on fire. The only way that NOAH could express such a plan might lead it to propose setting the building on fire in order to jump out of a window to get outside.

MOLGEN (Stefik[1981a,1981b]) uses an object constraint mechanism to allow a least-commitment approach to be used with respect to object selection. MOLGEN uses a rather inadequate representation of time, though. The properties of objects do not change. Instead objects are re-named when they change. This method works adequately in the domain of gene-splicing experiments but is not adequate for general problems.

SIPE (Wilkins[1984,1985]) is a more recent planner descended from NOAH which uses object constraints similarly to MOLGEN but is also able to reason about general fact changes. It provides an object constraint language which is based on a taxonomy of objects described in terms of their invariant properties. It also allows objects to be defined as resources. This allows the detection of conflicts between plans requiring the same resource. NOAH, like STRIPS, is a purely syntactic planner. NOAH cannot make inferences from its facts and its

actions must directly specify all fact additions and deletions. SIPE adds some deductive capabilities to NOAH. Not all additions and deletions of an action need to be explicitly stated, but instead some can be inferred by deductive operators. SIPE also allows limited use of existential variables and disjunction in the conditions part of actions, and universal variables in the effects part of actions. SIPE uses a method which is roughly equivalent to NONLIN for reasoning about the purpose of actions. SIPE is more restrictive than NONLIN in the partial orderings which it allows in its plans, though.

DEVISER (Vere[1983]) adds time-durations, events and inference capabilities to NONLIN. It was used in the domain of the Voyager spacecraft mission planning. It allows the description of external events as well as actions. Each event and action has a duration and a start-time window. Event definitions specify the effects which the event causes and the duration of the event. They also specify a start window and/or conditions for the event to occur. The durations of actions and events are fixed, but may be computed during planning. The effects of actions and events always occur at the end of the event. An action or event may also cause an event to immediately follow it. Using the ability to specify that events immediately follow some activity, it is possible to describe the timing of effects in a flexible manner. DEVISER can also distinguish between conditions which must hold over the entire interval of an action, and those which only need to hold at the start of the action. DEVISER allows inferences to be made which specify that if one conjunction of facts holds at a particular time, then some other fact(s) must also hold. DEVISER dynamically updates the start-time windows of actions and events during planning. It compresses the windows of the final task network to shorten the

total duration of the task-network. DEVISER is the most temporally sophisticated of the NOAH-type planners.

We will propose a system which is able to handle the standard planning problems. Like NOAH, it is a nonlinear hierarchical planner. We distinguish between applicability conditions and goal preconditions. We handle complicated subgoal interactions. We also handle object constraints to some extent. We are able to specify the effects of actions more flexibly than most NOAH-type planners. We do not support reasoning about time durations, so we are lacking some of the capabilities of DEVISER. We are able to specify external events unlike most planners. The most significant difference is that we are able to interact to prevent expected external events, unlike any of the planners described above.

2.3 Metaplanning

The term *metaplanning* is used somewhat differently by different researchers, but can be loosely defined as reasoning about the planning process. Hayes-Roth&Hayes-Roth[1979] and Stefik[1981b] define it as reasoning about decisions in the planning process. Wilensky[1983] defines it as using meta-goals and meta-plans so that the the planning process consists of finding metaplans for metagoals.

Hayes-Roth&Hayes-Roth[1979] proposed a model of *opportunistic* planning. They argued that often planning is not a top-down process of refinement, as proposed in the NOAH model. Instead they proposed that planning is a combination of top-down (goal-driven) and bottom-up (event-driven) processes. They produced studies with human subjects to

support their hypothesis that people plan opportunistically. Their planner was implemented using the blackboard model introduced by the HEARSAY speech understanding system (Ermann-et-al[1980]). The domain considered in their planner was errand-planning. The errand goals are given levels of priority, and the planner might have to abandon some of them. An example of event-driven planning would be to adopt a strategy of grouping errands by location if the planner notices that several errands are in the same part of town. The emphasis in their work was on helpful goal interactions. Their temporal representation was not discussed, and seems to be rather domain-dependent.

Stefik[1981b] defines metaplanning as the approach of organizing the decision-making knowledge into a layered control structure which separates decisions about the planning problem from decisions about the planning process. MOLGEN (Stefik[1981a,1981b]) is a planner which was designed using this philosophy and applied to the domain of planning gene cloning experiments in molecular genetics. The design philosophy behind MOLGEN is similar in many respects to that of Hayes-Roth&Hayes-Roth[1977] (see Stefik[1981b] for a comparison).

NASL (McDermott[1978]) uses domain-dependent meta-rules to help choose between potential plans for a given goal. These meta-rules can either rule-in, rule-out, or rule-together plans. Ruled-in plans are preferred over plans neither ruled-in nor ruled out, which are preferred over ruled-out plans. Ruled together plans are used together as a single plan to solve a goal. NASL also allowed the description of *policies*. A policy is a goal that requires some fact(s) to hold over the entire interval of a plan. This is not possible to express within NOAH, where goals only have to hold for an instant of time. NASL did

not distinguish between planning and execution. Instead it would always execute the first pending task. Complex tasks would be executed by decomposing them, and primitive tasks by simulating their effects. NASL therefore missed some solutions that a planner which plans before executing (like NOAH) would find. NASL was implemented in a rule-based system and was used in the domain of electronic circuit design.

Schmidt[1985] and Sridharan[1982] describe the PLANX10 planner. They are particularly concerned with metapanning where there is incomplete knowledge about the world. We will not address this issue.

Wilensky[1983] proposes that the process of planning be itself considered planned activity, which involves the use of meta-plans and meta-goals. The same mechanisms could be used for planning with these meta-plans and meta-goals as are used for regular plans and goals. Unlike Hayes-Roth&Hayes-Roth[1979] and Stefik[1981b], he avoids a layered approach. He argues that the same basic planning mechanisms should be used to find meta-plans for meta-goals as to find regular plans for regular goals.

There are three components to his theory:

- A theory of planning: how an intelligent agent determines and executes a plan of action.
- A theory of understanding: how an understander comes to comprehend the behavior of another.
- A theory of plans: which describes knowledge about planning used for both these tasks.

Wilensky argues for a declarative approach with respect to representing plans so that the knowledge about them can be used for both planning and

understanding purposes. His work is an extension of earlier work in commonsense reasoning and natural language understanding (e.g. Schank[1977]). Allen[1979] and Cohen[1979] suggest that sophisticated natural language understanding requires being able to simulate the goals and plans of the other agent. Other work suggests that planning systems and understanding systems should use the same representation (e.g. Charniak[1981], Charniak&McDermott[1985]). We will not be concerned with natural language understanding in this thesis, but agree with the philosophy of making the representation declarative.

Conventional planners do not worry about where their goals come from. They are given one typically complex goal to solve. Wilensky argues that in the real world, at least as much work is involved in determining what one wants to do as in determining how to do it. The ability to infer goals is important both for autonomous robots and for text comprehension. An autonomous robot will have to generate its own goals to respond to the changing environment around it. To understand other agents, it is important to infer their goals (Allen[1979] and Cohen[1979]).

When a system can generate its own goals, there will typically be several top-level goals. The concept of abandoning one top-level goal to solve another can be introduced. This concept is not in the vocabulary of NOAH-type planners, which only have one top-level goal.

Plans can also be proposed to solve several independent goals. For example, if it needed enough items, a planner could decide to do its entire week's shopping. This is similar to the type of event-driven planning proposed by Hayes-Roth&Hayes-Roth[1977].

Wilensky notes that some of the general-purpose critics used by NOAH ignore the true complexities of plan interactions. For example, the eliminate redundant preconditions critic might notice that there are two preconditions of have-paint in the plan to paint the ladder and the ceiling. It then modifies the plan so that the have-paint goal need only be solved once. This ignores a number of complexities, though. If we merge these preconditions we should get a new goal to get enough paint for the ladder and for the ceiling, which is not identical to either of the original have-paint goals. If we were to merge enough have-paint conditions, it might change the plan to get the paint from walking to the store to having to take a truck. Critics would be unable to generate such suggestions. The critics used in NOAH depend on the conditions of plans being simple.

The critics used in NOAH also do not record the reasons why they make the modifications that they do. Modifications may be made for efficiency reasons, or to resolve conflicts. These modifications may result in later conflicts. The critics which try to remedy these later conflicts do not have access to the reasons why the plan is in the state it is in. With a metapanning approach, all changes are justified as parts of meta-plans. When a conflict is detected, all the plan and meta-plan choices underlying the conflict can be examined.

When a planning conflict is detected, a resolve-conflict goal is generated. Wilensky proposes some methods for resolving such conflicts which are much more flexible than those provided with NOAH-like critics. This is because when tasks conflict, it is often due to some persistence of background facts or other assumptions. With Wilensky's model, we can generate a plan to change these background circumstances, and still pursue all the tasks mentioned in the conflict. Wilensky uses a Change-Circumstances plan to do this. The planner

is also able to decide to not do one of the plans in the conflict. Another solution is to restate a goal. For example, a goal of staying dry may be restated as a goal of not getting too wet. We will not investigate this type of solution to resolve-conflict goals.

Conventional planners do not reason about bad situations to avoid or about constraints to maintain over an interval. NASL is an exception to some extent. This ability is important in order to avoid generating a plan that causes bad side-effects. An example might be a robot in a factory that would want to avoid hurting people as it moves around. In Wilensky's model, a bad situation gives rise to a preservation goal. There is then a planning conflict between the preservation and the plan which lead to the bad situation. A resolve-conflict goal is generated to handle the planning conflict.

Wilensky describes a theory of resources, which can be used to reason about various types of positive and negative resources. We will not address the topic of resources in this paper. He also discusses the method of evaluating alternate simulated plans for a goal.

PANDORA (Faletti[1982], Wilensky[1983]) is an implementation based on these ideas, but it seems to fall short in many respects. For one thing, its representation of time is rather adhoc. The order in which actions are to be executed is dependent on their order in a plan queue. It is therefore a linear planner. It also seems that PANDORA can only prevent bad facts which are the result of its own actions. Wilensky[1983] does not give any examples of how problems addressed in the conventional planning literature would be solved by PANDORA. He does not address the standard subgoal interaction problem, which is the main focus of much planning research. His examples also typically

do not require sophisticated use of planning variables.

McDermott[1982,1983b,1985] proposes a large vocabulary of action types, and thus could be considered to discuss meta-planning. For example, he describes actions such as *avoid*, *allow* and *prevent*. These are all described within the framework of his temporal logic, which we will describe in the next section.

Allen[1984] also describes a more sophisticated representation for actions within the context of a temporal logic. He is more concerned with natural language issues which do not concern us, though. In Allen[1983b], he describes how standard nonlinear planning could be performed using his representation.

Like Wilensky, our planning is guided by metaplans and metagoals. Our vocabulary of metaplans and goals is much more limited than Wilensky's. We do not consider multi-agent situations, represent resources, or evaluate alternate scenarios. Instead we concentrate mainly on how to solve the resolve conflict goal. We propose a temporal representation and an assumption-based reasoning methodology which make clearer the interpretation of the metaplans and metagoals that we do have. In particular our approach makes the interpretation of our resolve-conflict goal clearer than the description by Wilensky. We will also show how more standard planning problems can be solved with our methods. Our thesis is based on the premise that handling complicated interactions with a changeable environment is a difficult problem, and though adhoc techniques like that used in PANDORA may work in special cases, they will not work in general cases. Furthermore it will be difficult to describe the cases which such systems would be able to handle.

We restrict our vocabulary of action types to a relatively small set and do not address most of the complexities described in McDermott[1982, 1983b, 1985]. We also do not address planning control issues such as considered by McDermott[1978], Hayes-Roth & Hayes-Roth[1979], and Stefik[1981a]). Our emphasis is on representational issues. The combination of our meta-planning approach and our ability to generate plans from facts might lend itself to such extensions, though.

2.4 Temporal Representations

One class of temporal representations describes time in terms of snapshots of the universe. These snapshots are frequently called *states*. States are typically considered instantaneous. They could be considered to occur over an interval of time but would have to describe an interval over which nothing changed in the universe. Facts are associated with all the states in which they are true. Events describe fact changes. The function expression $R(e, s)$ returns the state resulting from event e happening in the state s .

There are a large number of theories and systems which are based on this temporal model. STRIPS and the linear planners discussed in section 2.1 are some examples. The Situational Calculus (McCarthy[1969], Hayes[1973]) is a first order logic representation which uses state variables. They call states *situations*. Another term used is *possible world*. There are various other methods using state variables, modal logics or temporal logics based on this model of time. (e.g. Haas[1985], Hayes[1973], Kautz[1982], Moore[1977], Rosenschein[1981], Turner[1984]). There is considerable formal basis to these approaches. Kautz[1982] describes a planner which can reason about

disjunctions and quantification in a sophisticated way. Moore[1977] describes a scheme for a system to reason about its own knowledge and create knowledge goals.

This temporal model necessarily imposes a number of restrictions including the following:

1. Events are instantaneous.
2. No two events can occur at the same time.
3. There is no way to describe continuous change.
4. Events must be totally ordered.

Allen[1984], Forbus[1984], Hayes[1979,1985], McCarthy[1977] and McDermott[1982] all criticise this temporal model as insufficient.

The nonlinear planners derived from NOAH relax conditions 2 and 4 above by distributing state information over a partially ordered network of actions. DEVISER (Vere[1983]) explicitly allows events to overlap. Though DEVISER is a step in the right direction, NOAH-like nonlinear planners generally manipulate time in a manner which require strong assumptions to be made about the environment.

Allen[1984] proposes a temporal logic with time-intervals as the primitive temporal units. He does away with the notion of states altogether. Facts are associated with time-intervals, as are events. The time relation between time-intervals is described as one of seven primitive mutually exclusive time relations and their inverses. Temporal uncertainty can be expressed by describing the time-relation between time-intervals as being one of a subset of alternate primitive time-relations. Allen[1983a] describes how to compute transitive temporal relations. Allen[1983b] describes how one might do simple

NOAH-like planning using his representation. His representation has the advantage that time-intervals can be constrained to be disjoint if desired. Point-based methods require that such intervals be ordered one way or the other. Allen's representation therefore does not require as many choice points in planning.

McDermott[1982] proposes an extension of the situational calculus. His temporal logic is still based on states, but where an infinite number of states occur between any two states. Facts are associated with infinite sets of states. He therefore does away with the notion of a *next* state. Events occur over a time interval, described by two endpoint states. States are ordered with $<$ and $\leq$ relations. An implementation of his ideas (Dean[1984]) associates facts with intervals designated by pairs of endpoints, and so is closer to the approach of Allen[1984].

The temporal logics of McDermott[1982] and Allen[1984] do not distinguish between present, past and future, unlike some temporal logics (Turner[1984]). Instead *now* is simply a variable that is frequently updated. It is possible to determine whether some fact or event is past, present or future by comparing it to the current *now*.

Our approach uses a temporal representation based on that of Allen[1983a,1983b,1984,1985a]. In addition we introduce the notion of intersection intervals, which can be used to describe the complete time-interval of temporal conjunctions. Charniak&McDermott[1985] note that the inferences which this allows, which they call *overlap chaining*, cannot be performed by existing systems. Our method cannot be applied in a point-based scheme, but only in an interval-based scheme.

2.5 Causality and the Frame Problem

The *frame problem* (McCarthy[1969], Hayes[1973]) is essentially how to describe what does and does not change when an event occurs. The problem was initially posed in terms of the state-based model. Given some state s and an event e , what is the resulting state $R(e, s)$? It is not sufficient to describe what does change when e occurs, but also what does not change. There were various methods proposed for describing what does not change. The simplest (and most unwieldy) is to explicitly mention all the things that do not change when an event occurs. Such assertions are called *frame axioms*. STRIPS solved the problem with its addlist and deletelist. All facts in the old state not mentioned in either of these lists could be carried into the new state.

McDermott[1982] has difficulty inferring facts because it is not sufficient to infer that a fact holds for a single state in his logic. To remedy this problem, he introduces the notion of fact persistences. Fact causal rules have to specify the expected duration that the caused fact is expected to persist. Once the duration is up, the system simply loses information as to whether the fact holds. This approach is somewhat awkward. McDermott does make the valid point that we do seem to lose information about the truth of facts after some duration, but the length of this duration does not seem to be an intrinsic property of the causal rule. With an interval-based approach if we infer a fact we will automatically infer that it holds over a finite interval. Actually in an implementation of part of McDermott[1982]'s scheme (Dean[1984]) facts are associated with intervals defined by endpoints, so this problem does not occur.

DEVISER (Vere[1983]) is able to describe sophisticated events, allowing for time-delays before effects occur. There are some similarities between the

causal rules of Vere[1983] and McDermott[1982]. Both have conditions which must hold for some duration before the effect will occur. Our representation of events is influenced the most by these two models of causality.

Allen[1984] is most interested in distinctions relevant to natural language understanding, such as whether an agent intended to perform some action. We did not find his ideas useful with respect to the problem of how to infer events.

Allen's interval-based framework suggests a way to handle the frame problem, which is essentially a persistence problem. If we want to know whether some fact will persist until some given time-interval, we need only check the time-relation between them. This will be a disjunction of possible primitive relations. If these relations require that the intervals must be disjoint, then we know that the time-intervals do not meet. If these relations permit them to intersect, the planner can assume that they will. For example, the planner might want to ask himself whether his car will still be in its parking spot after he finishes shopping. The various events involved in shopping will not mention changes in the location of the car. The planner does not expect that some other event will lead to it changing position. The car-in-lot time-interval would therefore be asserted to be before, to meet, to be finished-by, or to contain the interval of arriving back at the parking spot. It is clearly possible that the car will still be there. The planner can therefore assume that it will be. Note that it is believed to be *possible* that the car will be gone when the planner gets back (if the relation is *before*) or that the car will be pulling out as the planner gets back (if the relation is *meets*). Allen[1983a] mentions this technique and notes that some sort of truth-maintenance is necessary, but does not propose a solution. We propose a method for assumption-based reasoning to handle such

persistence reasoning.

The representations of McDermott[1982] and Allen[1984] allow actions to take time and even overlap. They also allow continuous change to be modelled. These abilities are beyond the capabilities of the linear state-based representations. Allowing events to overlap does not automatically solve the problem of how to reason about the effects of events that overlap, though. The following example is due to Allen[1984]. Consider that there are two robots. One is pushing left on a block, and the other is pushing right on the same block. The block is not moving. (Clearly this could not be represented in a linear state-based temporal model) We will call these actions PUSHL and PUSHR. If we were to describe PUSHL and PUSHR independently we might propose the causal rules that PUSHL makes a block go left, and PUSHR makes a block go right. Such cause-effect rules clearly fail in this example. To solve this problem effectively we need to be able to describe the influence that PUSHL and PUSHR have on blocks. To determine the movement of a block, we would add up the influences on it. In the above example, we would find that the actions cancelled out, and predict that the block would not move.

Forbus[1984] describes a qualitative process theory which addresses the problem of how to reason about the influences that processes have on quantities. It is an extension of the Naive Physics proposals of Hayes[1979]. These proposals involve describing the course of events of the world in terms of *histories*, which are spatially bounded facts. Forbus has a representation in which facts and physical objects are represented with the same knowledge representation structure. Facts are therefore spatially bounded, while physical objects have lifetimes. Forbus[1984] bases his temporal representation on

Allen[1984]'s time-intervals. Forbus[1983, 1984] and Hayes[1979,1985] make interesting observations about time, space and causality and the limitations of current AI systems with respect to all three. We were influenced by Forbus' ideas and representation. Other papers on qualitative physics (in the same volume of Artificial Intelligence magazine as Forbus[1984]) include DeKleer[1984b] and DeKleer[1984c].

We will not address the problem of reasoning about influences, and instead restrict ourselves to a cause-effect model. Our point is that within the representational framework which we describe, such extensions might be possible. The situational calculus lacks the representational ability to even represent these types of interactions.

There is another aspect to the frame problem. This is simply that it is not possible to make enough qualifications to an event description so that it will always work. There is always an exception to any rule that is defined. This aspect of the problem requires some form of nonmonotonic reasoning so that we can allow our event rules to fail occasionally. This is not a problem for planners which only generate plans, but do not execute them. We are interested in executing plans, though, and allow the environment to behave unexpectedly.

2.6 Nonmonotonic Reasoning

Nonmonotonic reasoning refers to reasoning in which subsequent beliefs may make previous beliefs invalid. The beliefs of an agent therefore do not grow monotonically as do assertions in classical logic. There are a number of formal theories of nonmonotonic reasoning. (e.g. McDermott[1980a, 1981],

Reiter[1980], McCarthy[1980] and Moore[1985]).

There are a number of implementations that support nonmonotonic reasoning, typically called Truth Maintenance Systems (Doyle[1979]). Doyle[1979]'s TMS keeps track of the justifications underlying each belief. Beliefs are derived by a user program. If a contradiction is discovered, one of the assumptions underlying the contradiction must be revoked. This is done by justifying the negation of the revoked assumption by the other assumptions in the contradiction being IN. The process of undoing assumptions to recover from contradictions is called dependency-directed backtracking. It is more efficient than simple chronological backtracking. The TMS labels each belief node as IN or OUT. The IN nodes are the currently believed nodes. The OUT nodes are not believed (but not necessarily believed to be false). The TMS attempts to keep the IN nodes consistent. Assumptions can be made by justifying a node by another node being OUT.

McAllester[1978,1980] describes a TMS which is somewhat simpler than Doyle[1979]'s. It is based on a three-valued logic. It has a simpler method of handling contradictions than Doyle[1979]'s. It also differs in how assumptions are handled. It also requires global consistency.

DeKleer[1984a, 1986] describes an approach in which each derivation of an assertion is composed of the assertion, its justifications and the set of assumptions that underly the derivation. We can ignore the justifications part, since it has limited importance in the scheme. The assumption set represents a conjunction of assumptions. Assumptions are essentially assertions whose truth is not known. This is a different use of the term than Doyle[1979]. When an inference is made using some rule, the consequent derivation's assumption

set is equal to the union (conjunction) of the assumption sets of the antecedents. When a contradiction is derived under some assumptions, the assumption set is marked nogood. Any derivations based on assumptions that are a superset of a nogood assumption set can be deleted. DeKleer does not perform any backtracking. Instead his system finds all solutions to the given problem in parallel. There is no cost associated with switching contexts, since there is no current consistent context. Contexts are defined by the assumption sets of derivations. DeKleer[1984] notes some deficiencies with the Doyle[1979] style TMS. Besides the cost associated with switching contexts in Doyle's TMS, there are some problems with respect to missing some derivations if an assumption is revoked due to backtracking, then re-instated. DeKleer[1986] expands on the issues discussed in DeKleer[1984a], describing an assumption-based TMS system. It also provides a comprehensive review of related TMS systems.

McDermott[1983] proposes a scheme which fuses data-dependencies (Doyle[1979]) and contexts (data pools). It is a fairly complex scheme which seems to inherently support a branching future scheme. It has similarities with DeKleer[1984]'s approach. It allows switching between contexts to be performed more efficiently than Doyle's TMS. It requires a current consistent context, though. Its treatment of contradictions also differs from DeKleer. Under McDermott[1983]'s approach, negative assumptions can be added to derivations. We will give an example of this in the next section.

We consider that there are two kinds of assumptions: *guesses* and *choices*. The planner has control over choices. Though he cannot always make a choice occur, he can typically make a choice false. An example of a planning

choice is the choice of which plan to use to solve a goal. Guesses include the assumptions that causal rules will work and persistence assumptions. The planner has no direct control over these assumptions. In most planners, where all events are under the planner's control, all assumptions are choices. It is therefore reasonable to keep a consistent set of assumptions. We allow external events, so cannot control the truth of all assumptions underlying derivations. When there is considerable uncertainty as to what will occur, it may not be reasonable for the planner to randomly choose which guesses should be revoked in the case of a contradiction. Our planner wants to detect bad facts, and if it revokes the wrong guesses, it will not infer that certain bad facts are possible.

DeKleer[1984a, 1986]'s system seems well-suited to the problem of handling guess assumptions - it does not require global consistency and allows us to investigate alternate possible futures in parallel. When a contradiction occurs, it is not necessary to revoke any assumptions to maintain consistency. There are three problems with respect to applying DeKleer's methods to our problem. First, we do not want to investigate all plans for all goals in parallel. This would be prohibitively expensive. Second, we need to finally decide which actions we actually intend to do, so that we know which actions to execute. It seems useful for these intended actions to be consistent with each other. Third, our causal rules are not certain, since we want to allow them to be able to be contradicted by observations in the actual world, but we would like to have confidence in these causal assumptions. If we used a straightforward implementation of DeKleer[1984a]'s approach, we would end up making derivations based on persistence assumptions which contradict our event

expectations.

For these reasons we use a modified version of DeKleer[1984a]'s approach, which allows global inconsistency, but in which we can mark certain assumptions as *unlikely* (roughly equivalent to OUT). The assumption underlying plans that are not currently selected will be unlikely, as will persistence assumptions which conflict with expected events. We also distinguish between two degrees of *not unlikely* belief, so that we can maintain consistency between the assumptions underlying the task network.

2.7 Modelling Future Uncertainty

The Situational Calculus/Possible Worlds approach to modelling future uncertainty is to have a branching future. McDermott[1982] integrates a branching future model into his temporal logic, as well. McDermott[1983] describes the techniques which he proposes could be used to implement this.

Allen[1984] criticizes the branching future model. Instead, he proposes that the same general assumption-based reasoning techniques be used to reason about all types of uncertainty, including uncertainty about the past as well as uncertainty about the future. Branching future methods do not allow reasoning about past uncertainty. Reasoning about an uncertain past is clearly important for a natural language understanding system, and more generally for any system which has incomplete knowledge about the present. The process of trying to understand what is true now, or how the present situation came about cannot be handled effectively with branching future models.

We will assume for the purposes of this thesis that the system has complete knowledge about the present, but we do not want to make a design decision

which will fundamentally limit our model. Allen's temporal logic also does not seem to lend itself to a branching future model. One of the strengths of Allen's logic is the ability to constrain two time intervals to be disjoint. If a branching future mechanism were used, it seems we might have to consider each alternative ordering separately.

One feature of the branching future approach is that *negative assumptions* have to be added to assertions in a branch when a new offshoot branches off from a point earlier in the branch. For example, if the planner expects that Nell will be run over by a train, he can design a plan to save her. The branch describing Nell being run over by a train will now have the added condition that the prevention plan does not succeed.

We will use DeKleer[1984a]'s general assumption-based reasoning techniques. With DeKleer's approach, there is no need to add negative assumptions to derivations. DeKleer's method also suggests a better way for detecting and handling planning conflicts.

2.8 Prevention

Preventing hypothesized events from actually occurring is a hard problem. It is beyond the ability of all conventional planners. McDermott[1982] is one of the few who have addressed this problem in detail. Haas[1985] also discusses the problem, though he does not go so far as to propose exactly how to prevent something. Wilensky[1983] discusses the problem of resolving goal conflicts. He only seems to only be able to prevent situations which the planner causes, though. Allen[1984] gives an example of hiding a coat, which expresses prevention in terms of the durations of beliefs. He does not address the

problem of general prevention techniques.

McDermott [1982] describes a trap that a planning system using a naive method of prevention might get itself into:

Say a problem solver is confronted with the classic situation of a heroine, called Nell, having been tied to the tracks while a train approaches. The problem solver, called Dudley, knows that

"If Nell is going to be mashed, I must rescue her."

(He probably knows a more general rule, but let that pass.) When Dudley deduces that he must do something, he looks for, and eventually executes, a plan for doing it. This will involve finding out where Nell is, and making a navigation plan to get to her location. Assume that he knows where she is, and she is not too far away; then the fact that the plan will be carried out will be added to Dudley's world model. Dudley must have some kind of data-base-consistency maintainer to make sure that the plan is deleted if it is no longer necessary. Unfortunately, as soon as an apparently successful plan is added to the world model, the consistency maintainer will notice that "Nell is going to be mashed" is no longer true. But that removes any justification for the plan, so it goes too.

The problem in this example is that Dudley is justifying his task to prevent Nell being mashed by the assertion that Nell will be mashed. This is the naive way to handle the problem using justification-based reasoning. McDermott[1982] proposes that the plan to save Nell be justified by the assertion that Nell will be killed in the branch of the future in which the planner does not execute a plan to save her.

Our approach is based on the approach of Wilensky[1983] (though it seems that he would not be able to solve this example). A prevent-fact-plan (which corresponds to Wilensky's preservation goal) is generated and conflicts with Nell getting killed. A contradiction is detected between this plan and Nell getting killed. A resolve-conflict-goal is produced with arguments of the prevent-fact-plan and the assumptions under which Nell was killed. In this example, these assumptions might be that Nell would still be on the tracks when the train arrives, and that the train does arrive. A Resolve-Conflict goal

is solved by making one of its arguments false. In this case, the planner might solve the conflict using a plan to make this time-relation not hold. That is, the planner would generate a plan to make Nell stop being on the tracks before the train arrives. If the planner finds that Nell escapes somehow, the conflict is solved for free. Otherwise a positive plan of action is used.

2.9 Execution and Replanning

Most planners only consider how to generate plans, and not how to execute them. Execution brings with it the possibility that not all actions may work as expected. Unexpected events may also occur. Execution-monitoring is required to patch up the task network if what really occurs does not match expectations. Note that the unexpected occurrences could be either helpful or harmful. The process of revising plans during execution to take unexpected events into account is often called *replanning*.

The PLANEX system (Fikes[1972]) was used to monitor the execution of STRIPS plans that were represented in *triangle tables*. PLANEX uses the triangle table representation to determine the latest point in the plan where execution could begin in the current situation. If unexpected events create a situation in which restarting the plan from some other point than the current point is possible, PLANEX does this. This could involve either redoing previous actions or skipping actions that were planned. This is a particularly primitive type of replanning. It is unlikely that events in a complex world could be handled in this way.

NOAH (Sacerdoti[1977]) was designed for be used within the context of a larger system called the Computer-Based Consultant. The CBC project was

intended to produce a system that could fill the role of an expert in the cooperative execution of complex tasks with a relatively inexperienced human apprentice. The NOAH system was built to serve as the problem solving and execution monitoring component for dealing with assembly and disassembly tasks. NOAH has some replanning abilities. It does not allow the input of arbitrary predicates during execution, though, so the general problem we would like to address does not arise. It allows a node that has just been executed to be planned for again if it fails.

SIPE (Wilkins[1985]) allows arbitrary expressions to be input as the effects of an action. It is able to take advantage of helpful effects, and repair plans to fix harmful effects. It seems to have the most sophisticated replanning capabilities of current planners, which includes an ability to specify domain-dependent error-recovery information. It has the usual NOAH temporal model, however. It models unexpected effects by representing the action as occurring as expected, with a *Mother-Nature* event immediately following which causes the real effects.

We want to allow similar replanning capabilities to those of SIPE, but within a more sophisticated temporal representation. Most of these capabilities simply fall out of the rest of our techniques. This is because our methods for resolving conflicts and generating plans work equally well when the system is executing or just planning. We are able to model unexpected events which occur whether or not the planner is executing any actions. Unexpected events may even occur while the planner is in the midst of planning.

The added complexities which execution adds are subtle, but important. It requires us to be able to reason about *actual* facts, actions and events, and

not simply hypothetical ones. Haas[1985] and Hayes[1973] discuss the importance of the distinction between possible and actual events. It also requires us to be able to revise the beliefs of the planner when unexpected events occur. DeKleer's assumption-based approach suggests that when something is known it should be derived under the empty assumption set. *Observed* facts are asserted in this way in our proposed system.

Chapter 3

KNOWLEDGE REPRESENTATION

3.1 Introduction

In this chapter we describe the types of assertions that can be made, and how to define a domain. A notable feature of our representation is the similar way in which physical objects, facts, actions and events are described. There are two types of entities in our knowledge representation: *objects* and *relations*. There is a hierarchy of object types. The following five are the highest level types in the object hierarchy:

1. *Facts*. A fact is a partial description which holds over an interval of time. It can also be considered to hold over any subinterval of its complete interval. For example, a fact might describe a particular time-interval over which block1 was on block2. *Conjunction facts* are composed of sub-facts and describe a conjunction of facts holding over an interval.
2. *Actions*. There are three types of actions. *Commands* are primitive actions which can be directly executed by the planner. *Plans* are complex actions composed of sub-actions, sub-facts and sub-events. They are decomposable, but not directly executable. *Goals* are actions which are neither executable nor decomposable, but instead require a plan to be selected to solve them. We provide a small number of domain-independent goal types.
3. *Events*. An event describes a change in the environment. Events have conditions composed of facts and/or commands and the time-relations

between them. An event also specifies fact and time-relation effects. It is expected that if the conditions occur that the effects will occur. The event occurs if and only if its conditions and effects occur. We allow that the conditions may occur without the effects occurring.

4. *Physical Objects*. A physical object is described by its type and perhaps one or more arguments describing its invariant property values.
5. *Values*. Values are simple objects which have only a type. For example, red might be a value of type colour-value.

We can illustrate the object hierarchy with the following diagram:

We have a small number of relations that hold between objects. We will show relation predicate names in upper case to distinguish them from object type names. We have two kinds of relations: constraint relations and non-constraint relations.

1. Constraint Relations

- a. Time-relation expressions assert a temporal relationship between two time-intervals.
- b. Equal and not-equal expressions assert inequalities between variables or objects.

- c. Instance and not-instance expressions assert that an object or variable either is of a particular type, or is not of a particular type.
- 2 Non-constraint relations.
- a. Planfor expressions assert that a plan is applicable to solve a goal.
 - b. A bad-fact expression asserts that a fact is undesirable. The planner will try to prevent such a fact if possible.
 - c. A planning-conflict expression is used to describe a planning conflict. The detection of planning conflicts leads to the generation of resolve-conflict goals to solve them.

3.2 Objects, Constants and Variables

An object is a specific fact, action, event, physical object or value that has an extension in the real world. We can describe objects using the following syntax:

`(<type> <name> (<argument-name1> <object1>) ... (<argument-namen> <objectn>)`

This syntax is essentially the same as that used by Wilensky[83] in the PEARL knowledge representation language. We can partially describe an object by not specifying all its arguments.

Each *constant* is known to refer to a distinct object. We can accumulate temporal, inequality and type constraints on constants. The system is initialized with definitions of constants describing the initial facts which hold in the world, as well as the physical objects which exist. Value constants are defined explicitly in the domain declaration rules. New constants are created to

describe facts which are observed to occur. Constant names start with a letter.

A *variable* describes a hypothesized object. We can accumulate temporal, inequality and type constraints on variables. *Choice variables* refer to objects which the planner attempts to bind. Physical object and value choice variables correspond to the standard planning variables found in the AI literature. They correspond to arguments of a plan which are not made specific by plan definition. Fact choice variables describe facts the planner wants to achieve. The planner chooses the bindings of choice variables as part of the planning process. *Non-choice variables* describe hypothesized events, facts and actions. Non-choice fact variables describe facts which describe the effects of events. We will refer to unbound variable instances using names that start with the character \$.

We could describe the constant block1, and the variable \$block2 as follows:

(block block1 (colour red))

(block \$block2 (colour blue))

We can describe the variable fact \$on-23 which describes an interval over which block1 is on \$block2 as follows:

(on \$on-23 (top block1) (bottom \$block2))

3.3 Time Relations

Each fact or action has one associated time interval. It is possible for a time-interval to be shared by more than one fact or action. We describe the time-relations between facts and actions by asserting the time-relations between their time-intervals. There are no state (possible world) variables. Allen[1983b] identifies seven primitive mutually exclusive interval relations and

their inverses. The following table illustrates these time-relations.

	Relation	Inverse Relation
AAAA BBBB	A equal B	B equal A
AAAA BBBB	A before B	B after A
AAAA BBBB	A meets B	B met-by A
AAAA BBBB	A overlaps B	B overlapped-by A
AAAA BBBBBB	A starts B	B started-by A
AAAA BBBBBBBB	A during B	B contains A
AAAA BBBBBB	A finishes B	B finished-by A

We could define temporal relations using these relations as the predicate, e.g. (before time1 time2). Allen[1983b] proposes that the relation between two times be described as one of a set of possible relations. Note that the relations given above are mutually exclusive. The relation *time-relation* asserts that a particular relation holds between two time intervals. It has the form:

(TIME-RELATION <time1> <time-relation-value> <time2>)

Strictly speaking, the <time1> and <time2> arguments of time-relation expressions should be time-intervals. We will allow either argument to be an action or fact, where it is understood that it is the time-interval of the action or fact that is being related, and not the object itself. The time-relation-value argument is a set of possible mutually exclusive relations. For example, we could say:

(TIME-RELATION time1 [before after meets] time2).

This is logically equivalent to:

(before time1 time2) $\vee$ (after time1 time2) $\vee$ (meets time1 time2)

but it is more computationally efficient to express this disjunction using our time-relation relation. In section 6.1.1 we will describe the temporal inferences which we can make using this representation.

We will frequently use names to refer to time-relation values instead of enumerating the set of primitive time-relations. For example, we will define :

intersects =

[equals starts started-by during contains finishes finished-by overlaps overlapped-by]

disjoint = [before after meets met-by]

in = [starts during finishes]

in-or-equal = [equals starts during finishes]

starts-while = [equals starts started-by finishes overlapped-by]

starts-before = [before meets overlaps started-by contains finished-by]

The time-relation assertion does not allow arbitrary disjunctions to be specified.

For example, we cannot express:

(before time1 time2) $\vee$ (after time2 time3)

with a single time-relation assertion. The restricted form of temporal disjunction described by the time-relation assertion is useful for most purposes, however. For example it is possible to constrain two time-intervals to be disjoint with the expression:

(TIME-RELATION time1 [before after meets met-by] time2).

An endpoint-based scheme would express this as:

end(time1) $\leq$ start(time2) $\vee$ end(time2) $\leq$ start(time1)

and would treat this as a general disjunction. Planners that want to constrain two facts to be disjoint and use point-based models typically treat this disjunction as a choice point. They make one or the other constraint, then if this leads to a problem, they might try the other ordering instead. Using Allen's time-interval based method, we can directly constrain them to be disjoint and do not need this choice point. Conventional NOAH-type planners can be treated as point-based models. (For example, see Charniak[1985]).

The distinctions between past, present and future time-intervals are not built into the logic. Instead, there is always a single time-interval designated the current now interval. We can determine whether a fact or action is past, present or future by testing its time-relation with this interval.

It is worth noting that our proposed system only allows the relations between time-intervals to be specified, and not their durations. It seems that time-relation reasoning using time-intervals is the most fundamental form of temporal reasoning, and should form the basis for further temporal reasoning extensions. Allen[1983a,1983b,1985] and Vilain[1982] describe methods for reasoning about dates, time durations and time-points, using an interval-based approach.

3.4 Defining Object Types

We can define object types which extend the object hierarchy described in figure 3.1 using a specialized syntax. The argument names of the object types defined are specified by the "?"-names mentioned in the patterns of the definition. The following example will illustrate how object types are defined. The types block, on, surface, table and colour-value could be defined as follows:

```
(define-value colour-value
  range: {red yellow blue})

(define-physical-object surface
  properties:
    (colour-value ?colour))

(define-physical-object block
  isa: surface)

(define-physical-object table
  isa: surface)

(define-fact on
  arguments:
    (block ?top)
    (surface ?bottom)
  restrictions:
    (NOT-EQUAL ?top ?bottom))
```

The isa: specifier is used to define one object type as a case of another type. The object type defined inherits all the arguments of the object type given as the isa: argument, and all the type and inequality restrictions imposed on them. Every fact instance of the more specific type is necessarily an instance of the less specific type. The above definitions therefore imply that:

```
(ISA surface physical-object)
(ISA block surface)
(ISA block physical-object)
(ISA table surface)
(ISA table physical-object)
(ISA on fact)
```

We will reconsider the definitions of block1, \$block2 and \$on-23:

```
(block block1 (colour red))
(block $block2 (colour blue))
(on $on-23 (top block1) (bottom $block2))
```

The following follows from these definitions (we will ignore the ~~no~~ instance assertions):

```
(INSTANCE block1 block)
```

```
(INSTANCE block1 physical-object)
(INSTANCE $block2 block)
(INSTANCE $block2 physical-object)
(INSTANCE $on-23 on)
(INSTANCE $on-23 fact)
```

```
(NOT-EQUAL block1 $block2) ; from on definition and $on-23.
```

3.5 Defining Value Types

- We can define ranges of values with the define-value declaration.

```
(define-value <value-type>
  isa: <super-type>
  range: <set of possible values>)
```

The following are examples of value definitions:

```
(define-value wetness-value
  range: {dry wet})
```

```
(define-value colour-value
  range: {yellow blue red green purple orange})
```

We say that blue is a value, and colour-value is a value type. The values described in the range of a value definition are always constants, so are disjoint.

3.6 Defining Physical Object Types

Physical objects can specify invariant properties, but will frequently only specify a type. Properties of a physical object that can change have to be described using facts. We allow an invariant property of an object type to be defined this way only because it is more efficient than defining a fact type. The capability for defining invariant properties is not semantically necessary, since properties can always be defined using facts.

We will not associate physical objects with time-intervals in this thesis.

Instead we implicitly assume that physical objects are neither created nor destroyed. Giving physical objects time-intervals would not be very difficult, though. We would merely have the additional inference that all facts and commands have to be temporally in the time-intervals of their physical object arguments. We could then write events to create and destroy physical objects. The main reason that we do not pursue this is that when objects are created and destroyed, we frequently have to reason about fragments of objects (e.g. of a broken vase) or even liquids (e.g. from a dropped egg). This is difficult. (Hayes[1979, 1985], Forbus[1984]).

3.7 Defining Action and Fact Types

Actions and facts have time-intervals associated with them. All actions except goals (i.e. facts, commands and plans) also have propositional arguments. These are the physical object and value arguments of the fact or action. They are specified by the *arguments*: specifier in the definition of the fact or action type. Propositional arguments have several uses.

1. They provide a high-level description of the fact or action which is not in terms of sub-events, sub-facts or sub-actions. Plans can be briefly described by their propositional arguments.
2. They also allow us to constrain the time-intervals of two facts or actions whose type and propositional arguments match. Such objects must be before, after or equal to each other to enforce the constraint that the time-intervals of objects be their complete time-interval.

For example:

(on on-21 (top block1) (bottom block3)) $\wedge$ (on on-24 (top block1) (bottom block3))

→
(TIME-RELATION on-21 [equal before after] on-24)

Type restrictions on the arguments are imposed in the arguments: part The restrictions part can be used to impose additional not-equal, equal, instance and not-instance constraints.

3.8 Defining Fact Types

A fact is a partial description of the world which holds over an interval, and can be said to hold over any subinterval as well. Essentially a fact describes some unchanging (or continuously changing) state of the world. We can describe a conjunction of facts which holds over an interval using the conjunction fact and. Conjunction facts have two subfact arguments. The time-interval of the conjunction fact is constrained to be the intersection of the time-intervals of its sub-facts. Conjunctions representing the conjunction of three or more simple facts can be built using conjunction facts as sub-facts.

For example, we might have a conjunction fact describing an interval over which an agent was outside while it was raining. If the outside fact were called outside1 and the raining fact were raining1, we could describe the conjunction fact out-in-rain1 as follows :

```
(and out-in-rain1
  (subfact1 outside1)
  (subfact2 raining1))
```

Out-in-rain1 requires that outside1 temporally intersects raining1. i.e.

```
out-in-rain1 ↔ (TIME-RELATION raining1 intersects outside1)
```

We will describe how the time-interval of the conjunction fact is constrained to be the intersection of the time-intervals of its subfacts in section 6.1.1.

3.9 Defining Mutually Exclusive Facts

The define-exclusive-facts definition allows us to assert that two facts cannot both occur at the same time. If two matching facts are found, they are made temporally disjoint. The form of a define-exclusive-facts definition is as follows:

```
(define-exclusive-facts
  restrictions: <restrictions>
  facts: <fact1> <fact2>)
```

For example, in the blocks world only one block may be on another block at the same time. We can therefore define the following domain constraint:

```
(define-exclusive-facts
  restrictions:
    (NOT-EQUAL ?block1 ?block2)
  facts:
    (on (top ?block1) (bottom ?block3))
    (on (top ?block2) (bottom ?block3)))
```

3.10 Defining Command Types

A command is an action which the planner can directly execute. Unlike the primitive actions described in most planners, we do not define preconditions or effects of commands in their definitions. This is because the effect of a command will depend on the context in which it is executed. We do not model commands as succeeding or failing. It is only the plan which uses them which succeeds or fails depending on the effects of the command. Our use of the term command is similar to that of Haas[1985]. The only way in which the planner can physically change its environment is through the execution of commands. Our definition of commands is similar to our definition of facts.

```
(define-command <command-type>
```

isa: <super-type>
arguments: <arguments>
restrictions: <restrictions>)

In addition to the domain-specific physical commands, we have the bind-command meta-command which binds a choice variable. We will discuss this command in section 7.3.

Our temporal representation can represent overlapping commands, but without duration information it is not practical to reason about executing overlapping commands. For this reason we will assume that only one command can be executed at a time.

3.11 Goal Types

A goal is an action which is neither executable nor decomposable. It is the planner's task to find a plan which can lead to the goal being realized. The time-interval of a goal is made equal to the time-interval of the plan that solves it. We use the following types of goals:

1. Achieve goals.
2. Bind goals.
3. Resolve-conflict goals.
4. Make-disjoint goals.

Achieve goals are the most common goals. An achieve goal has a goal fact argument which describes a choice-variable fact which should be achieved. The achieve goal is solved (and occurs) if and only if this fact occurs. Our achieve goal is more powerful than the usual achievement goals, which specify that a goal fact need only hold for an instant. We can arbitrarily constrain the time-interval of the goal-fact of an achieve goal within a plan.

A bind goal has a single variable argument. The bind goal is solved when its variable argument is bound to a variable or constant that is known to describe an actual object. Planners generally produce plans containing choice-variables and leave the problem of finding bindings for these variables for a second execution step. We integrate this process explicitly into the planning process using instantiate goals.

The resolve-conflict goal is an important goal. It is used to resolve a planning conflict described by a set of assumptions. It seems unlikely that the user would want to directly define plans to solve resolve-conflict goals, though the user might define domain-dependent plans to solve make-disjoint goals, which allow methods for resolving conflicts to be specified indirectly.

The make-disjoint goal has two facts as arguments and is solved if the facts are prevented from temporally intersecting. These goals are produced to solve resolve-conflict goals where the conflict includes an intersection time-relation constraint. We provide a plan to solve a make-precede goal by terminating the persistence of one argument before the other starts. The make-disjoint goal illustrates how a planner can prevent a temporal constraint from holding. It does not provide the means to solve all types of temporal constraints found in planning conflicts, nor will we provide a complete set of plans to solve it. It seems that to prevent temporal constraints in a sophisticated way requires duration reasoning.

3.12 Defining Event Types

Event definitions have the form:

```
(define-event <event-type>
```

conditions: <conditions>
effects: <effects>)

Event type definitions define the expectation that if some conditions occur, some effects will result. An event occurs if and only if all its conditions and effects hold/occur. The conditions can specify commands and/or facts and the time-relations between them. The effects specify fact(s) and the time-relations between them and the condition objects. We do not associate a time interval with an event. This has not seemed necessary since we can describe when an event occurs in terms of the time-intervals of its conditions and effects. Many events are instantaneous. If we did associate events with times, we would have to introduce time-points into our representation. See Vilain[1982], Allen[1985a, 1985b], and Forbus[1984] for treatments of time-points consistent with a time-interval based representation.

The causal rules which event type definitions describe are assumed to work most of the time. In addition we assume that no two event type definitions (or set of type event definitions) will be contradictory. That is, if their effects are mutually exclusive, their conditions must be contradictory. We allow event rules to fail occasionally.

Our event declaration is quite general. This is primarily because it is not yet entirely clear what sort of restrictions we would like to make on it. There are a few basic principles which any event type definition must satisfy, though:

1. Objects defined as effects cannot be mentioned in the conditions part.
2. All condition objects must start before all effect objects.
3. A condition fact whose persistence is terminated by an effect fact should be constrained to meet the fact which terminates it.

We will use two kinds of event types in this thesis: command-events and non-command-events. Command-events contain a single command in the conditions part.

Our first example will illustrate how commands can have different effects in different contexts. We have a flip-switch command which will make a switch become on if it is off and off if it is on.

```
(define-value switch-value
  range: (switch-on switch-off))

(define-physical-object switch)

(define-fact switch-state
  arguments:
    (switch ?switch)
    (switch-value ?state))

(define-exclusive-facts
  restrictions:
    (NOT-EQUAL ?state1 ?state2)
  facts:
    (switch-state
      (switch ?switch)
      (state ?state1))
    (switch-state
      (switch ?switch)
      (state ?state2)))

(define-command flip-switch
  arguments:
    (switch ?switch))

(define-event flip-switch-on
  conditions:
    (switch-state ?switch-off
      (switch ?switch)
      (state switch-off))
    (flip-switch ?flip-switch
      (switch ?switch))
    (TIME-RELATION ?flip-switch in-or-equal ?switch-off)
  effects:
    (switch-state ?switch-on
      (switch ?switch)
      (state switch-on))
```

```

      (TIME-RELATION ?flip-switch {meets} ?switch-on))

(define-event flip-switch-off
  conditions:
    (switch-state ?switch-on
      (switch ?switch)
      (state switch-on))
    (flip-switch ?flip-switch
      (switch ?switch))
    (TIME-RELATION ?flip-switch in-or-equal ?switch-on)
  effects:
    (switch-state ?switch-off
      (switch ?switch)
      (state switch-off))
    (TIME-RELATION ?flip-switch {meets} ?switch-on))

```

The time-relation expression in the conditions part of these event definitions indicate that the initial switch position must hold over the entire interval of the command. The effect time-relation asserts that the command temporally meets the caused switch position. Given that the initial and final switch positions describe mutually exclusive facts (from the exclusive-facts definition) and the time-relation condition between the command and initial switch state holds, it will be inferred that the initial switch state temporally meets the final switch state.

We consider two types of condition facts, distinguished by the time-relation-value of the condition time-relation between the command and the fact:

1. The condition must hold over the entire interval of the command,
time-relation-value = in-or-equal = [equals starts during finishes]
2. The condition must hold over an interval at the start of the command,
time-relation-value = starts-during
= [equals starts started-by during overlapped-by]

Various other condition relations could be expressed, such as having the

condition hold over the end of the command, or during the middle. To reason properly about these cases would probably require duration reasoning. It might also be useful to decompose commands into sub-commands and mention the sub-time-intervals explicitly. We will stick with our two simple forms.

We have two kinds of non-command events. The first type says that if a conjunction of facts hold, then there will be some effect after some delay. The second type describes an instantaneous change. Our first example is from our version of Wilensky[1983]'s raincoat example. It asserts that if an agent is dry, outside in the rain and not wearing a raincoat, he will get wet. There is some delay before he gets wet.

```
(define-physical-object person)

(define-value wetness-value
  range: {dry wet})

(define-fact wetness
  arguments:
    (physical-object ?object)
    (wetness-value ?value))

(define-value place
  range: {inside outside})

(define-fact location
  arguments:
    (physical-object ?object)
    (place ?value))

(define-fact not-wearing-a-raincoat
  arguments:
    (person ?agent))

(define-event get-wet-in-rain
  conditions:
    (and ?outside-in-rain-without-raincoat
      (subfact1
        (and
          (subfact1
            (location
```

```

      (object ?person)
      (value outside)))
    (subfact2
      (weather ?raining
        (value raining))))))
  (subfact2
    (not-wearing-raincoat
      (agent ?person))))
  (and ?dry-outside-in-rain-without-raincoat
    (subfact1
      ?outside-in-rain-without-raincoat)
    (subfact2
      (wetness
        (object ?person)
        (value dry))))))
  effects:
    (dryness ?person-wet
      (object ?person)
      (value wet))
    (TIME-RELATION ?dry-outside-in-rain-without-raincoat [meets] ?person-wet)
    (TIME-RELATION ?outside-in-rain-without-raincoat [overlaps] ?person-wet))

```

The event asserts that the person will remain wet over the complete interval in which he is outside-in-rain-without-raincoat, and still be wet after this interval is complete. (This is due to the second time-relation effect constraint.).

We can also define event types in which there is no delay. That is, in which a fact change occurs the instant some conditions hold. For example, we can create an event which states that a lightbulb will start shining as soon as its switch is on and the power is on. This event will allow the system to predict that if the switch is flipped on when the power is on that the bulb will start shining. It can also be used to predict that if the power comes on when the switch is on that the bulb will shine.

```

(define-value brightness-value
  range: {dark bright})

(define-situation bulb-brightness
  arguments:
    (lightbulb ?bulb)
    (brightness-value ?brightness))

```

```

(define-value power-state-value
  range. (power-on power-off))

(define-situation power-state
  arguments:
    (power-state-value ?state))

(define-event light-on-event
  conditions:
    (bulb-brightness ?bulb-dark
      (bulb ?bulb)
      (brightness dark))
    (and ?switch-on-and-power-on-for-bulb
      (subfact1
        (and
          (subfact1
            (switch-state
              (switch ?switch)
              (state switch-on)))
          (subfact2
            (power-state
              (state power-on))))))
      (subfact2
        (switch-for-bulb
          (switch ?switch)
          (bulb ?bulb))))))
    (TIME-RELATION
      ?bulb-dark
      meets-or-intersects
      ?switch-on-and-power-on-for-bulb)
  effects:
    (bulb-brightness ?bulb-light
      (bulb ?bulb)
      (brightness light))
    (TIME-RELATION ?bulb-dark [meets] ?bulb-light)
    (TIME-RELATION
      ?switch-on-and-power-on-for-bulb
      [equals starts started-by]
      ?bulb-light))

```

This event definition allows the persistence of the ?bulb-light effect to terminate while ?switch-on-and-power-on-for-bulb still holds. We could say that ?bulb-light holds over the complete interval of ?switch-on-and-power-on-for-bulb by making the second time-relation effect have the relation-value [equals starts]. We

could say that ?bulb-light holds only as long as ?switch-on-and-power-on-for-bulb holds by making the second time-relation effect have the relation-value [equals]. In this case we might want to infer a bulb-dark fact effect which asserts that the bulb will be dark immediately following ?switch-on-and-power-on-for-bulb.

3.13 Defining Plan Types

A plan is a complex action which is not directly executable, but can be decomposed into subactions. A plan definition has the following form:

```
(define-plan <plan-type>
  arguments: <propositional-arguments>
  conditions: <conditions>
  actions: <subactions>
  events: <subevents>
  protections: <protections>)
```

The time-interval of the plan is defined to be the minimum interval containing all the time-intervals of the sub-actions. The conditions of the plan are facts which must hold in the environment and which the planner does not cause. The subevents of the plan specify the causal links between parts of the plan. A plan may also specify assertions that are to be protected. Typically for most plans the time-relation protections do not have to be explicitly defined in the plan definition, since they follow from the definitions of the subevents of the plan. Only the protections not described by the sub-events need to be explicitly defined. The plan occurs if and only if all its subactions, conditions and subevents occur, and its protections hold.

A plan to solve an achieve goal will typically include as one of its parts the fact which represents its purpose. When a plan is generated to solve an achieve goal, this fact will be instantiated to the goal-fact of the achieve goal. It

is only by including events in plans that we are able to mention this fact within the plan. Plans are usually based on some causal rule(s) which lead the planner to believe that the actions will lead to some result. Our plans make the causal structure of the plan explicit.

The following example illustrates a plan definition.

```
(define-physical-object surface)
```

```
(define-physical-object table  
  isa: surface)
```

```
(define-physical-object block  
  isa: surface)
```

```
(define-fact clear  
  arguments:  
    (surface ?surface))
```

```
(define-fact on  
  arguments:  
    (block ?top)  
    (surface ?bottom))
```

```
(define-exclusive-facts  
  facts:  
    (on (bottom (block ?block)))  
    (clear (surface ?block)))
```

```
(define-exclusive-facts  
  restrictions:  
    (NOT-EQUAL ?block1 ?block2)  
  facts:  
    (on (top ?block) (bottom ?block1))  
    (on (top ?block) (bottom ?block2)))
```

```
(define-exclusive-facts  
  restrictions:  
    (NOT-EQUAL ?block1 ?block2)  
    (block ?block)  
  facts:  
    (on (top ?block1) (bottom ?block))  
    (on (top ?block2) (bottom ?block)))
```

```
(define-command move-block-command  
  arguments:
```

```

(block ?block)
(surface ?to))

(define-event move-block-event
  conditions:
    (clear ?clear-block
      (surface ?block))
    (clear ?clear-to
      (surface ?to))
    (move-block-command ?move-block-command
      (block ?block)
      (to ?to))
    (TIME-RELATION ?move-block-command in-or-equal ?clear-block)
    (TIME-RELATION ?move-block-command in-or-equal ?clear-to)
  effects:
    (on ?on-block-to
      (top ?block)
      (bottom ?to))
    (TIME-RELATION ?move-block-command [meets] ?on-block-to))

(define-plan move-block-plan
  arguments:
    (block ?block)
    (surface ?to)
  actions:
    (achieve ?achieve-clear-block
      (goal-fact (clear ?clear-block
        (surface ?block))))
    (achieve ?achieve-clear-to
      (goal-fact (clear ?clear-to
        (surface ?to))))
    (move-block-command ?move-block-command
      (block ?block)
      (to ?to))
  events:
    (move-block-event ?move-block-event
      (move-block-command ?move-block-command)
      (clear-block ?clear-block)
      (clear-to ?clear-to)
      (on-block-to (on ?on-block-to
        (top ?block)
        (bottom ?to)))))

```

We will illustrate one use of explicit plan protections with the following simplistic example. To get from home to office, the planner has to get from home to bus-stop1, take a bus to bus-stop2, then get from bus-stop2 to office. There is a

take-bus-from-bus-stop1-to-bus-stop2 event which asserts that being at bus-stop1 will get one to bus-stop2. Self is the planner; a person.

```
(define-value place
  range: { home bus-stop1 on-bus bus-stop2 office })

(define-fact location
  arguments:
    (physical-object ?object)
    (place ?value))

(define-physical-object person)

(define-event take-bus-from-bus-stop1-to-bus-stop2
  conditions:
    (location ?person-at-bus-stop1
      (object (person ?person))
      (place bus-stop1))
  effects:
    (location ?person-on-bus
      (object ?person)
      (place on-bus))
    (TIME-RELATION ?person-at-bus-stop1 [meets] ?person-on-bus)
    (location ?person-at-bus-stop2
      (object ?person)
      (place bus-stop2))
    (TIME-RELATION ?person-on-bus [meets] ?person-at-bus-stop2))

(define-plan go-to-office
  conditions:
    (location ?self-at-home
      (object self)
      (place home))
  actions:
    (achieve ?achieve-self-at-bus-stop1
      (goal-fact (location ?self-at-bus-stop1 (object self) (value bus-stop1))))
    (achieve ?achieve-self-at-office
      (goal-fact (location ?self-at-office (object self) (value office))))
  events:
    ( take-bus-from-bus-stop1-to-bus-stop2
      (person-at-bus-stop1 ?self-at-bus-stop1)
      (person-at-bus-stop2 (location ?self-at-bus-stop2
        (object self)
        (value bus-stop2))))
  protections:
    (TIME-RELATION ?self-at-home starts-before ?achieve-self-at-bus-stop1)
    (TIME-RELATION ?self-at-bus-stop2 starts-before ?achieve-self-at-office))
```

The first protection states that ?self-at-home must be true before the ?achieve-self-at-bus-stop1 action can start. The second protection states that the ?achieve-self-at-office action should start after the planner gets to bus-stop2. This will constrain the plan selected for ?achieve-self-at-office (and all its parts) to start after the planner is at bus-stop2 as well.

3.14 Define-Planfor

The assertion (PLANFOR goal1 plan1) means that plan1 is proposed to solve goal1. define-planfor definitions are used to infer planfor assertions. A define-planfor definition has the following form :

```
(define-planfor
  goal: <goal>
  conditions: <conditions>
  plan: <plan>)
```

The goal describes the goal which the plan can be used to solve. The conditions specify the applicability conditions of the plan. These conditions could include the condition arguments of the plan. (We will describe the conditions arguments of planfor definitions in more detail in section 7.1) For example, we might have the following planfor definitions for the plans described in the previous section:

```
(define-planfor
  goal:
    (achieve
      (goal-fact (on ?on)))
  plan:
    (move-block-plan
      (on-block-to ?on)))
```

```
(define-planfor
  goal:
    (achieve
```



```

        (location ?self-at-office (object self) (value office)))
conditions:
  (location ?self-at-home (object self) (value home))
plan.
  (go-to-office
    (self-at-home ?self-at-home)
    (self-at-office ?self-at-office)))

```

The go-to-office plan may seem to be recursively calling the same achieve goal to solve itself, since the plan to solve the ?self-at-office achieve goal is solved by another achieve goal with the same goal-fact. They are not the same goal, though, since the time-intervals of the plans to solve them are temporally unequal. (One temporally contains the other).

We can also use planfor definitions to generate top-level plans. Dummy top-level supergoals are generated for each top-level plan. This is slightly different from Wilensky[1983]'s model, in which situation-goal rules generate top-level goals explicitly. The differences are minor, though, since the top-level plan produced will invariably contain one or more goals. For example, we could generate a plan containing a goal to know what is going on in the world when it is morning.

```

(define-value time-of-day-value
  range: [morning afternoon night])

```

```

(define-fact time-of-day
  arguments:
    (time-of-day-value ?value))

```

```

(define-fact know-whats-going-on)

```

```

(define-plan top-level-know-plan
  conditions:
    (time-of-day ?morning-time
      (value morning))
  actions:
    (achieve ?achieve-know-whats-going-on
      (goal-fact (know-whats-going-on ?know-whats-going-on)))
  protections:

```

```
(TIME-RELATION ?know-whats-going-on starts-during morning-time))
```

```
(define-planfor
  goal: top-level
  conditions:
    (time-of-day (value morning))
  plan:
    (top-level-know-plan
      (morning-time ?morning-time)))
```

This planfor definition will generate a top-level-know-plan for every morning fact it detects. A top-level-know-plan contains a single achieve goal which attempts to determine what is going on before the end of the morning.

3.15 Defining Bad Facts

We can define a fact description to be undesirable using a define-bad-fact definition. The form of a bad fact definition is:

```
(define-bad-fact
  fact: <fact>
  restrictions: <restrictions>)
```

When a fact fact1 unifies with the bad fact description we infer (BAD-FACT fact1). The planner will try to prevent any fact that is asserted to be bad in this manner. For example, in the raincoat example the planner does not want to get wet so we define:

```
(define-bad-fact
  fact:
    (dryness
      (object self)
      (value wet)))
```

self refers to the planner, and would be defined just like any other object. If the following fact is inferred:

```
(dryness wet1 (object self) (value wet)),
```

the following bad-fact relation will also be inferred:

(BAD-FACT wet1).

Chapter 4

ASSUMPTION-BASED REASONING

4.1 The Fundamentals

The description of assumption-based reasoning described here is based on the model of DeKleer[1984a, 1986], though our terminology is slightly different. We will express that an assertion is derived under some assumptions as:

(Derived *assertion* *assumption-set*)

The assumption set is a conjunction of assumptions. Assumptions are assertions whose truth is unknown. The assumption set must represent sufficient conditions for the derived assertion to hold. If the assumption set is the empty set, the assertion is known. Once a derivation is made, its assumption set will never change. (This is unlike McDermott[1983]'s approach, in which new negative assumptions can be added). A derivation is removed from the knowledge base if its assumptions are known to be contradictory, or if a more specific derivation is made. Some kinds of assertions can have more than one derivation.

A basic assumption-based reasoning operation is to find the union (conjunction) of two given assumption sets. We can compare two assumption sets by finding their union, then comparing it to each of the original assumption sets. If either is equal to the union, it is a superset of the other. When a new assumption set is created, it is compared against existing assumption sets to see if an equal one already exists which can be used instead. It is important to make assumption sets unique in this way. If one assumption set is a subset of

another assumption set, it describes assumptions that are equal to or less restrictive than those of the other.

A derivation is as specific as another derivation if its assertion is as specific as the other derivation's assertion, and its assumption set is a subset of the other derivation's assumption-set. When we say that a derivation is as specific as another derivation, what we mean is that the derivation is as parsimonious as the other derivation. For example, the derivation:

(Derived P {a1 a2})

is more specific than the derivation:

(Derived P {a1 a2 a3}).

For most kinds of assertions, an assertion is as specific as another assertion only when they are equal. Time-relation constraints are the only exception in our proposed system. A time-relation constraint is as specific as another time-relation constraint if its time-relation value describes a set of possible time-relations which is a subset of the possible time-relations described by the other constraint. For example, the constraint :

(TIME-RELATION t1 [before] t2)

is more specific than the constraint:

(TIME-RELATION t1 [before meets] t2).

It is therefore the case that the derivation:

(Derived (TIME-RELATION t1 [before] t2) {a1 a2 a3})

is more specific than the derivation:

(Derived (TIME-RELATION t1 [before meets] t2) {a1 a2 a3}).

Neither of the following two derivations is more specific than the other:

(Derived (TIME-RELATION t1 [before] t2) {a1 a2 a3})

(Derived (TIME-RELATION t1 [before meets] t2) {a1 a2})).

When a new derivation is made, it is compared with existing derivations. If an existing derivation is as specific as the new derivation, the new derivation will not be added. Otherwise, if the new derivation is more specific than existing derivations, the less specific derivations are removed from the knowledge base. This ensures that we do not keep a derivation that is less specific than another derivation.

If we have two derivations, we can derive the conjunction of their assertions under the union (conjunction) of their assumption sets. For example, if the knowledge base contains:

(Derived B {a1 a3})
(Derived C {a2 a3})

and the system has the rule:

$B \wedge C \rightarrow D,$

the system can derive:

(Derived D {a1-a2 a3}).

Certain types of inference rules will derive the consequent under more assumptions than the union of the assumptions of the antecedent.

If a contradiction is detected, the assumption set underlying the contradiction is asserted to be *nogood*. The system keeps track of the most specific assumption sets that are *nogood*. Only if the assumptions of the contradiction are not a superset of an existing *nogood* assumption set does the new assumption set get added to the existing *nogood* sets. If an assumption set is added to the *nogood* sets, all *nogood* sets that are a superset of it can be removed. In this way, the system only keeps track of the minimum *nogood* assumption sets.

Derivations with assumption sets that are supersets of nogood assumption sets are deleted from (or not added to) the knowledge base. Such derivations are based on contradictory assumptions, so are not useful to the system. The following simple example illustrates this process. Suppose that the knowledge base includes the following derivations, and there are no nogood assumption sets.

(Derived (TIME-RELATION t1 [overlaps contains finished-by] t2) {a1 a2 a3})
(Derived P {a1 a2 a3 a4 a5})
(Derived Q {a1 a2 a4})
(Derived R {a1 a2 a3 a4})

Consider that the following derivation is then made and added to the knowledge base:

(Derived (TIME-RELATION t1 [before after meets met-by] t2) {a1 a2 a4})

This contradicts the time-relation already derived above between t1 and t2. We therefore generate the following nogood set:

(NOGOOD {a1 a2 a3 a4}).

The derivation of P is based upon an assumption set which is a superset of this nogood set, so is removed. The knowledge base now contains the derivations of two conflicting constraints. New constraints may be derived from either of these constraint derivations.

4.2 Kinds of Derivations

4.2.1 Constant and Variable Derivations

When we derive a variable or constant under some assumptions, we assert that the variable or constant refers to an actual object under those assumptions. Constants are always derived under the empty assumption set. Each variable

or constant will always have one derivation. If some of its assumptions are found to be necessarily true, a more specific derivation may replace the existing derivation. In this way, though a variable is only associated with one set of assumptions at a time, these assumptions may decrease with time. Since constants are derived under the empty assumption set, the assumption set associated with them cannot decrease, so will remain the same. We will refer to the assumption set underlying the derivation of the variable or constant as the assumptions of the variable or constant. The assumptions of a variable describe necessary and sufficient conditions for the variable to refer to an actual object. If these assumptions are found to be nogood, it becomes known that the variable cannot refer to any object.

4.2.2 Constraint Derivations

A constraint derivation defines the assumptions under which some constraint holds. It is possible to have more than one derivation of the same constraint. The assumptions of a constraint derivation must be a superset of the union of the assumptions of the arguments of the constraint. If the assumptions of a constraint derivation equal the union of the assumptions of the argument, the constraint is a *necessary constraint*. For example, given the following two fact derivations:

```
Derived
(wetness $self-wet1 (object self) (value wet))
{ a1 a2 a3 }
```

```
(Derived
(wetness $self-dry1 (object self) (value dry))
{ a2 a3 a4 })
```

and given that an object cannot be both wet and dry at the same time, the

system would derive that these facts are disjoint:

```
(Derived  
  (TIME-RELATION $self-wet1[before after meets met-by] $self-dry1)  
  { a1 a2 a3 a4 })
```

This is a necessary constraint since it is derived under the union of the assumptions of (its arguments) \$self-wet1 and \$self-dry1.

4.2.3 Other Relation Derivations

A planfor assertion asserts that a plan is applicable to solve a goal. Its single derivation depends on the assumptions of its conditions, and not of its goal or plan arguments.

A badfact derivation is based on the assumptions of its fact argument.

planning-conflict assertions describe a planning conflict, and are derived under the empty assumption set.

4.3 Making Assumptions

Our proposed system makes both variable and constraint assumptions, as well as some other special kinds of assumptions. We will examine each in turn, but it is first useful to define a restriction we make upon all assumption sets. We require that:

- Any assumption set which contains a constraint assumption must also contain all the assumptions of the arguments of that constraint.
- Any assumption set which contains a variable assumption must also contain all the assumptions of that variable.

That is, every assumption set must contain all the assumptions which the

assumptions in the assumption set depend on. We impose this constraint so that we can be certain that an assumption set which describes strictly weaker assumptions than another will be a subset of the other assumption set.

4.3.1 Variable Assumptions

A variable assumption is generated to serve as an assumption of itself. Not every variable has itself as an assumption.

Event assumptions are generated to serve as an additional assumption of an event variable. Event assumptions are used to make causal rules defeasible. An event is based on the assumption that all its conditions hold, and also on its own assumption. If the effects of an event do not occur as expected, it will cause the assumptions of the event to be found to be nogood, and hence the event derivation and all derivations that followed from it will be deleted. If we derived an event only on the assumptions of its conditions, if it did not occur as expected we would find the empty assumption set to be nogood.

When we create a *command variable*, its assumptions will always include itself.

Planning variables are variables generated by plans whose binding the planner chooses. They are initially derived under an assumption set which contains the variable itself as an assumption.

4.3.2 Constraint Assumptions

The constraint conditions described in domain definitions are *assumed*. That is, the system does not look for derivations of them because such

derivations would typically not exist. There are two steps to the process of making a constraint assumption:

1. *Testing whether the constraint is possible.* If the constraint is contradicted by a necessary constraint between the same arguments, the answer is *impossible*. The assumption cannot be made. Otherwise, if the necessary constraint between the same arguments is as specific as the constraint, the answer is *necessary*. The assumption does not need to be made; it is true under the argument assumptions alone. Otherwise, the answer is *possible*. In such a case the assumption can be made. It will be added to the assumptions of the consequent(s) produced by the rule.
2. *Projecting the consequences of the assumption.* If a rule infers a derivation that is then added, all the assumptions made as conditions of the rule must be projected. That is, for every constraint test which gave a *possible* response (as described above) we have to derive a constraint *assumption derivation*. Such a derivation has the constraint as its assertion, and the union of the constraint itself and the argument assumptions as the assumptions.

4.3.3 Other Assumptions

There are a number of other kinds of assumptions that our proposed system uses:

1. *all-plans-for* assumptions. (See section 7.9).
2. *is-likely* assumptions. (See section 7.1).
3. *not* assumptions. (See sections 7.4 and 7.6).

4. *never-added* assumptions (See section 7.10).
5. *not-selected-plan-for* assumptions. (See section 7.7).
6. *not-proposed* assumptions. (See section 7.9).

4.4 Top Level Assumptions

A top-level assumption of an assumption set is an assumption such that no other assumption in the assumption set depends on it. That is, given an assumption set A , and an assumption a which is contained in it, and an assumption set $(A - a)$ which is the assumption set formed by removing a from A , a is a top-level assumption of A if and only if:

1. There does not exist an assumption as-specific as a in the assumption set of any variable assumption in the set $(A - a)$.
2. There does not exist an assumption as-specific as a in the assumptions of the arguments of any constraint assumption in the set $(A - a)$.

For example, if we have the following derivations:

```
(Derived $v1 { $v1 $v2 $v3 (EQUAL $v2 $v3) $v2 })
(Derived $v2 { $v2 $v3 })
(Derived $v5 { $v5 })
(Derived $v6 { $v5 })
```

the top-level assumptions of the assumption:

```
{ $v1 $v2 $v3 (EQUAL $v2 $v3) (TIME-RELATION C1 intersects $v6) $v5 }
```

are:

```
{ $v1 (TIME-RELATION C1 intersects $v5) }
```

4.5 Detecting All the Nogood Assumption Sets

The nogood assumption sets describe false conjunctions, so in effect could

also be considered to be disjunctions. The system removes nogood assumption sets which are supersets of other nogood assumption sets, so avoids keeping nogood assumption sets which are obviously unnecessary. This technique alone will neither provide minimum nogood assumption sets, nor will its nogood assumption sets be able to remove all derivations which are logically based on invalid assumption sets. Consider the following two nogood assumption sets:

(NOGOOD { p q })
 (NOGOOD { (NOT p) q })

Clearly the assumption set { q } is nogood, but this would not be determined by the system we have described so far. Consider also the following nogood assumption sets:

(NOGOOD { p (NOT q) t u v })
 (NOGOOD { (NOT q) r s t })

It follows from this that { p r s t u v } is nogood. Nogood assumption sets describe negated conjunctions, and can therefore be expressed as disjunctive clauses. The two inferences made above were made using the resolution rule of inference. That is:

$$(p \vee q) \wedge (\neg p \vee r) \rightarrow q \vee r$$

The first example is the special case in which $r = q$. Note that the rule does not have to be used on expressions of the form p and $\neg p$, but instead can be used between any two statements whose disjunction is necessarily true, like (EQUAL a b) and (NOT-EQUAL a b), or time-relation constraints where the union of their time-relation-values describes a necessarily true relation constraint. It is not clear how necessary it is to perform resolution in this way for our proposed

system. It might be useful to look at the first special case at least, since it results in smaller nogood sets, whereas the general case would tend to create larger nogood sets. DeKleer[1986] discusses this topic in more detail.

We have another potential problem with respect to detecting nogood assumption sets which does not follow directly from DeKleer's approach, but rather from our insistence that variables be derived under only one assumption set. When a new variable is generated, it may be based on some constraint assumptions. If derivations of those constraints are later made, the variable is not re-derived with the assumptions of these new derivations. What this means is that we do not have every possible derivation of every assertion. Because we do not have every derivation of every assertion, we will not have every possible derivation of contradictions, and hence may miss detecting nogood assumption sets.² To remedy this, we generate new nogood assumption sets with constraint assumptions replaced by the assumptions of all their derivations. The algorithm is as follows:

```

FOR each nogood assumption set nogood1 DO
  FOR each constraint assumption constraint1 in nogood1 DO
    nogood-minus-constraint1 ← remove constraint1 from nogood1
    FOR each assumption-set1 under which constraint1 is derived DO
      IF constraint1 derived-assumptions1 THEN
        new-nogood1 ← nogood-minus-constraint1 union derived-assumptions1
        make new-nogood1 a nogood assumption set
      ENDIF
    ENDFOR
  ENDFOR
ENDFOR

```

If we make a new derivation of a constraint, and that constraint is an assumption of an existing nogood set, we have to make a new copy of the nogood set substituting the new derivation assumptions for the constraint

assumption.

4.6 Removing Assumptions from Assumption Sets

When it is determined that a constraint assumption is necessarily true (i.e. that the necessarily relation is as specific), its assumption belief will be removed as less specific than the necessary relation derivation. The assumption will not be removed from the derivations of variable derivations, because they are not rederived as assumption constraints get new derivations. The solution is simply that when a constraint assumption becomes necessarily true, or a variable assumption becomes known to be true, remove it from all assumption sets in which it occurs. This could be considered equivalent to replacing a derivation with a more specific one. We would not want to rederive the forward inferences made from such derivations, though.

4.7 Degree of Belief

Every assumption, assertion and assumption-set has a current degree of belief, which is *unlikely*, *uncertain* or *likely*. The degree of belief in an assumption set is the minimum degree of belief of its assumption elements. The degree of belief of an assertion is the maximum degree of belief in an assumption set which derives it. The degree of belief of an assumption roughly describes the degree of belief in the statement; *if all the necessary assumptions for this assertion hold, the assertion will hold*. For example, event assumptions are always *likely* because it is strongly assumed that if the conditions of an event occur, that the event will occur. The degrees of belief do not really correspond to true degrees of belief or probability. They are intended to serve

two purposes:

1. Derivations based on unlikely assumptions do not get added to the knowledge base until their assumptions become uncertain or likely. This allows a measure of control - the planner can mark the assumptions of currently unselected plans as unlikely to keep from having to investigate their consequences until they are explicitly selected. We will discuss this issue further in section 5.1.
2. All likely assumptions are globally consistent, and in particular are consistent with the task network. We require that the assumptions of the plan for conditions for plans be likely. This will ensure that the total task network remains likely. We discuss this issue further in section 7.3.

We can describe the degrees of belief in various assumption types as follows:

1. Command assumptions and choice variables are likely only if the plan they are part of is selected. Otherwise they are unlikely.
2. An event assumption is always likely.
3. A constraint assumption is unlikely if a *not* assertion with it as an argument is likely. It is likely if it is protected, or if its assertion is derived to be likely under assumptions not including itself. Otherwise a constraint assumption is uncertain.
4. A *not* assumption is likely if it is protected by a plan or its assertion is derived to be likely under assumptions not including itself. It is unlikely otherwise.
5. An *is-likely* assumption is likely if its argument assertion is likely. It is unlikely otherwise.

6. A *not-selected-plan-for* assumption is likely if the given plan is not selected for the given goal, and unlikely otherwise.
7. *never-added* and *all-plans-for* assumptions are always likely.
8. A *not-proposed* assumption is likely if its plan argument is not currently proposed for any goal. That is, there is no likely plan for assertion mentioning it. The *not-proposed* assumption is unlikely otherwise.

Chapter 5

THE INFERENCE ENGINE

5.1 The Inference Loop

It seems useful to give constraint derivations priority over variable derivations. We suggest doing this by having a higher priority queue to handle constraint derivations and a lower priority queue for all other derivations. Constraint derivations detect contradictions, and it is in general desirable to detect contradictions sooner rather than later. It might also be useful to give derivations with fewer assumptions priority over those with more assumptions.

A derivation made with an unlikely assumption set is not added immediately. Instead it is cached, indexed by its assumption set. If an assumption set goes from being unlikely to uncertain or likely, all such cached derivations are put into the appropriate inference queue so that they can be inferred. We discuss under what circumstances derivations are more specific or less specific in chapter 4. The inference loop can be described as follows:

```
LOOP
  IF nonempty(constraint-derivation-queue) THEN
    derivation1 ← dequeue(constraint-derivation-queue)
  ELSE
    IF nonempty(object-derivation-queue) THEN
      derivation1 ← dequeue(object-derivation-queue)
    ELSE
      EXIT-LOOP
    ENDIF
  IF assumption-set(derivation1) is NOT NOGOOD THEN
    IF assumption-set(derivation1) is unlikely THEN
      cache derivation1
    ELSE
      IF there does not exist a derivation as-specific as derivation1 THEN
        remove all less-specific derivations than derivation1 from the knowledge base
```

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

```

        add derivation1 to the knowledge base
        put all derivations inferred by derivation1 in the appropriate inference queues
    ENDIF
ENDIF
ENDLOOP

```

An unlikely derivation that is already in the knowledge base can be used to make forward inferences. The forward inferences made from it will typically be based on a superset of its assumptions, so will also be based on unlikely assumptions. These new derivations will therefore not be added, but instead *cached*.

5.2 Pseudo Rules

We do not propose to use a general rule-based system, but in order to describe the kinds of inferences which our system makes we will use *pseudo rules*, which capture the intended meaning of these inferences. Pseudo rules are forward inferencing rules. If the conditions of the antecedent part are satisfied, the consequent part is derived. The assumptions underlying the derivation of the consequent is the union of the assumptions of the derivations of the antecedent. The following example illustrates the syntax of pseudo-rules.

```

(→
  (AND
    (EQUAL ?object1 ?object2)
    (EQUAL ?object2 ?object3))
  (EQUAL ?object1 ?object3))

```

We will define in addition some special predicates that can be used to describe when assumptions are made. The expression (assume assertion1) tests whether assertion1 is necessarily true, necessarily false or just possible. For constraint assumptions the response is necessarily true if it is a necessary

relation, necessarily false if the necessary relation contradicts it, and possible otherwise. (See section 4.3.2). For all other types of assumptions the answer is possible. When the answer to the question is necessarily true, the (assume assertion1) expression succeeds under no assumptions. When the answer to the question is necessarily false the assume expression fails. Otherwise it succeeds under the assumption of its argument.

In addition, constraint assumptions have assumption derivations made from them. (See section 4.3.2).

5.3 Unification Assumptions

When a variable is unified with another variable or with a constant in order to satisfy the antecedent of some rule, the assumption that the variable is equal to the argument it unified with is added to the assumptions of the antecedent of the rule. For example, consider that we have the following two derivations:

```
(Derived
  (on $on1 (top $block1) (bottom block3))
  { a1 a2 })
```

```
(Derived
  (on $on2 (top $block2) (bottom block3))
  { a2 a3 })
```

If two facts have equal propositional arguments, either they hold over the same interval, or one fact holds before the other does. Therefore we could make the following derivation:

```
(Derived
  (TIME-RELATION $on1 [equal before after] $on2)
  { a1 a2 a3 (EQUAL $block1 $block2) })
```

If we imposed the typical blocks world constraint that no two blocks can be on

the same block at the same time, we could also derive:

```
(Derived
 (TIME-RELATION $on1 [before after meets met-by] $on2)
 { a1 a2 a3 (NOT-EQUAL $block1 $block2) })
```

Unification can also lead to instance or not-instance constraints being added. Imagine a domain in which there are two types of blocks: heavy-blocks and light-blocks. We could define an event which asserts that if the robot holds a heavy-block, it will break a gasket.

```
(Define-event break-gasket
 conditions:
  (and ?holding-heavy-block-and-gaskets-intact
    (subfact1
      (gaskets-intact ?gaskets-intact))
    (subfact2
      (holding (object (heavy-block ?heavy-block))))))
 effects:
  (gasket-broken ?gasket-broken)
  (TIME-RELATION ?holding-heavy-block-and-gaskets-intact [meets] ?gasket-broken))
```

If the robot has a plan in which it has a block choice variable \$block1, where it is not specified whether the block should be heavy or light, and has derived that it will be holding the block:

```
(Derived
 (holding $holding1
  (object $block1))
 { $block1
   assum1 })
```

and gaskets-intact1 describes an observed fact that the gaskets are intact, the following derivations can be made:

```
(Derived
  (and $gaskets-intact-and-holding-block1
    (subfact1 gaskets-intact1)
    (subfact2 $holding1))
  { (TIME-RELATION gaskets-intact intersects $holding1)
    $block1
    assum1 })
```

```
(Derived
  (break-gasket $break-gasket1
    (holding-heavy-object $holding1)
    (gasket-broken $gasket-broken1))
  { (INSTANCE $block1 heavy-block)
    $break-gasket1
    (TIME-RELATION gaskets-intact intersects $holding1)
    $block1
    assum1 })
```

```
(Derived
  (gasket-broken $gasket-broken1)
  { (INSTANCE $block1 heavy-block)
    $break-gasket1
    (TIME-RELATION gaskets-intact intersects $holding1)
    $block1
    assum1 })
```

```
(Derived
  (TIME-RELATION $gaskets-intact-and-holding-block1 [meets] $gasket-broken)
  { (INSTANCE $block1 heavy-block)
    $break-gasket1
    (TIME-RELATION gaskets-intact intersects $holding1)
    $block1
    assum1 })
```

The planner will derive that it will break a gasket under the assumptions that \$block1 is a heavy block, and the event rule works.

Chapter 6

PROJECTION

6.1 Constraint Propagation

6.1.1 Time-Relation Inferences

If we derive that two time intervals have relation *relation1* under some assumptions and that they have relation *relation2* under a second set of assumptions, we can infer that they have the relation which is the intersection of *relation1* and *relation2* under the union of the two assumption sets. If the relation values do not intersect, we detect a contradiction.

```
(→  
  (AND  
    (TIME-RELATION ?time1 ?relation1 ?time2)  
    (TIME-RELATION ?time1 ?relation2 ?time2)  
    (set-intersection ?relation1 ?relation2 ?intersection-relation)  
    (NOT (EQ ?intersection-relation [ ]))  
    (TIME-RELATION ?time1 ?intersection-relation ?time2))
```

```
(→  
  (AND  
    (TIME-RELATION ?time1 ?relation1 ?time2)  
    (TIME-RELATION ?time1 ?relation2 ?time2)  
    (set-intersection ?relation1 ?relation2 [ ]))  
  Contradiction)
```

For example, given the following two derivations:

```
(Derived  
  (TIME-RELATION time22 [before after meets met-by] time45)  
  { a1 a2 a3 })
```

```
(Derived  
  (TIME-RELATION time22 [before meets overlaps finished-by contains] time45)  
  { a2 a4 a5 })
```


we can make the following derivation:

```
(Derived
  (TIME-RELATION time22 [before meets] time45)
  { a1 a2 a3 a4 a5 })
```

Allen[1983a] describes a method for computing the transitive relations between time intervals. It can be described by the following rules :

```
(→
  (AND
    (TIME-RELATION ?time1 ?relation1 ?time2)
    (TIME-RELATION ?time2 ?relation2 ?time3))
  (TIME-RELATION ?time1 (transitive-time-relation ?relation1 ?relation2) ?time3))
```

```
(→
  (AND
    (TIME-RELATION ?time1 ?relation1 ?time2)
    (TIME-RELATION ?time3 ?relation2 ?time2))
  (TIME-RELATION
    ?time1 (transitive-time-relation ?relation1 (inverse-relation ?relation2)) ?time3))
```

```
(→
  (AND
    (TIME-RELATION ?time1 ?relation1 ?time2)
    (TIME-RELATION ?time1 ?relation2 ?time3))
  (TIME-RELATION
    ?time2 (transitive-time-relation (inverse-relation ?relation1) ?relation2) ?time3))
```

The function *transitive-time-relation* computes the transitive time-relation between two relations expressed as sets of possible primitive relations. The transitive relation of two primitive relations is stored in a thirteen by thirteen table. To find the transitive time-relation of two complex relations, it is necessary to find the union of the transitive relations computed for each permutation of primitive pairs of relations of the two complex relations. We can describe the algorithm as follows:

```
(DEFINE Transitive-time-relation (time-relation1 time-relation2)
  result <- []
  LOOP FOR simple-time-relation1 IN time-relation1
```

```

    LOOP FOR simple-time-relation2 IN time-relation2
      result ← (union result
                (transitive-relation-table
                 simple-time-relation1
                 simple-time-relation2))
    ENDLOOP
  ENDLOOP
  RETURN result
ENDDEFINE

```

The function *transitive-relation-table* looks up the transitive relation between two of the thirteen simple time-relations in a thirteen by thirteen table. Some examples of transitive relations from this table are:

```

(transitive-relation-table before before) = [before]
(transitive-relation-table before during) = [before meets starts during overlaps]
(transitive-relation-table before after) = all thirteen possible relations (no information)

```

Allen[1983a] notes that the constraints imposed with his method only ensure consistency between any three time-intervals. Achieving global consistency would require an exponential algorithm. Nevertheless, it seems that the limited consistency which Allen's method maintains is useful for most purposes. Allen's method requires $O(n^2)$ space to store the time-relations between each time-interval. Allen[1983a,1985] suggests methods to reduce the space and time requirements of his method by using *reference intervals* and *date lines*. We will not address these issues in this thesis.

We introduce the notion of intersection intervals. We can define a time-interval to be the intersection of two sub-intervals. We can make stronger inferences using our technique than we could by using only transitive temporal constraints alone. Charniak&McDermott[1985] call an inference which is equivalent to what we do *overlap chaining*, and claim that no existing system can make the inference. Our method depends on the use of a time-interval

based temporal representation. It does not seem that the same method could be used in a time-point temporal model. Whenever a new time-relation constraint is imposed between the sub-intervals of an intersection interval, new relations are computed between the intersection interval and each sub-interval which are stronger than the ones asserted via transitive time-relations. The following rule and function definition describe the inference:

```
(→
  (AND
    (INTERSECTION-INTERVAL ?sub-interval1 ?sub-interval2 ?intersection-interval)
    (TIME-RELATION ?sub-interval1 ?relation ?sub-interval2))
  (AND
    (TIME-RELATION ?intersection-interval (Intersection-relation ?relation 1) ?sub-interval1)
    (TIME-RELATION
      ?intersection-interval (Intersection-relation ?relation 2) ?sub-interval2)))

(DEFINE Intersection-relation (relation sub-interval-number)
  result <- []
  LOOP FOR simple-relation IN relation
    result <- (union result (intersection-relation-table ?relation ?sub-interval-number))
  ENDLOOP
  RETURN result
ENDDEFINE
```

The *intersection-relation-table* function determines, given a simple relation between the two sub-intervals, what the relation between the intersection interval and each subinterval should be. The first column of the following table describes the time-relation inferred between the two sub-intervals. The second column describes the relation that can be inferred to hold between the intersection interval and the first sub-interval. The third column describes the relation that can be inferred to hold between the intersection interval and the second sub-interval. In terms of the *intersection-relation-table* function; the first column is *relation*, the second column gives the value of (intersection-relation-table *relation* 1), and the third column gives the value of

(intersection-relation-table relation 2): Note that when the sub-intervals do not intersect, there is no valid relation between the intersection interval and the sub-intervals because there would be no intersection interval.

subtime1 : subtime2	time : subtime1	time : subtime2
equal	[equal]	[equal]
during	[equal]	[during]
contains	[during]	[equal]
starts	[equal]	[starts]
started-by	[starts]	[equal]
finishes	[equal]	[finishes]
finished-by	[finishes]	[equal]
overlaps	[finishes]	[starts]
overlapped-by	[starts]	[finishes]
before	[]	[]
after	[]	[]
meets	[]	[]
met-by	[]	[]

Figure 6.1 Intersection Relation Table

The following example will illustrate why this approach gives stronger inferences than does using transitive time-relations alone. The closest way to defining an interval as an intersection interval using only time-relations is to say that the intersection interval is in-or-equal two sub-intervals. Consider if we have the following derivations:

(Derived
(TIME-RELATION time1 [equal starts during finishes] time2)
{ a1 a2 })

(Derived
(TIME-RELATION time1 [equal starts during finishes] time3)
{ a1 a3 })

(Derived
(TIME-RELATION time2 [overlaps] time3)
{ a1 a2 a3 a4 })

From these relations we can derive the following using transitive time-relation inferences:

```
(Derived
  (TIME-RELATION time1 [during finishes] time2)
  { a1 a2 a3 a4 })
```

```
(Derived
  (TIME-RELATION time1 [starts finishes] time3)
  { a1 a2 a3 a4 })
```

If we were to say that time1 is the intersection interval of time2 and time3, we could make the following more specific derivations:

```
(Derived
  (TIME-RELATION time1 [finishes] time2)
  { a1 a2 a3 a4 })
```

```
(Derived
  (TIME-RELATION time1 [starts] time3)
  { a1 a2 a3 a4 })
```

Intersection intervals are used to describe the time-interval of a conjunction-fact. To describe a conjunction of n primitive facts requires $O(n)$ extra time-intervals to build a binary tree of intersection intervals. The time-interval of a conjunction-fact situation is defined to be the intersection of the time-intervals of its sub-facts. We infer a conjunction fact based on the assumptions of its subfacts and the assumption that these subfacts temporally intersect.

6.1.2 Inequality and Instance Inferences

We propagate inequality and instance constraints as well as time-relation constraints. The rules below describe these fairly intuitive inferences. Note that we can detect contradictions when equal and not-equal assertions have the

same arguments, and when instance and not-instance assertions have the same arguments.

```
(→  
  (AND  
    (EQUAL ?object1 ?object2)  
    (EQUAL ?object2 ?object3))  
    (EQUAL ?object1 ?object3))
```

```
(→  
  (AND  
    (EQUAL ?object1 ?object2)  
    (NOT-EQUAL ?object2 ?object3))  
    (NOT-EQUAL ?object1 ?object3))
```

```
(→  
  (AND  
    (EQUAL ?object1 ?object2)  
    (NOT-EQUAL ?object1 ?object2))  
    Contradiction)
```

```
(→  
  (AND  
    (EQUAL ?object1 ?object2)  
    (EQUAL ?object2 ?object3))  
    (EQUAL ?object1 ?object3))
```

```
(→  
  (AND  
    (EQUAL ?object1 ?object2)  
    (INSTANCE ?object2 ?type))  
    (INSTANCE ?object1 ?type))
```

```
(→  
  (AND  
    (EQUAL ?object1 ?object2)  
    (NOT-INSTANCE ?object2 ?type))  
    (NOT-INSTANCE ?object1 ?type))
```

```
(→  
  (AND  
    (INSTANCE ?object1 ?type)  
    (NOT-INSTANCE ?object2 ?type))  
    (NOT-EQUAL ?object1 ?object2))
```

```
(→
  (AND
    (INSTANCE ?object1 ?type)
    (NOT-INSTANCE ?object1 ?type))
  Contradiction)
```

```
(→
  (AND
    (INSTANCE ?object ?type)
    (ISA ?type ?super-type))
  (INSTANCE ?object1 ?super-type))
```

Any two distinct constants are known to be *not-equal*.

6.2 Imposing Time-Relation Constraints Between Objects

In chapter 3 we described the *define-exclusive-facts* definition. We can describe the inference which it makes as follows:

```
(define-exclusive-facts
  restrictions: ?restrictions
  facts: ?fact1 ?fact2)
```

is equivalent to :

```
(→
  (AND
    (assume ?restrictions)
    ?fact1
    ?fact2)
  (TIME-RELATION ?fact1 [before after meets met-by] ?fact2))
```

If two facts or actions have the same propositional arguments, they should either be temporally equal, or one should be before the other. In the rule below, the relation *propositional-arguments-equal* succeeds under the assumptions that the propositional arguments of its two action or fact arguments are the same.

```
(→
  (AND
    (action-or-fact ?action-or-fact1)
    (action-or-fact ?action-or-fact2)
    (propositional-arguments-equal ?action-or-fact1 ?action-or-fact2))
  (TIME-RELATION ?action-or-fact1 [equal before after] ?action-or-fact2))
```

We will also constrain any two commands to be temporally disjoint. This is not a necessary requirement of our representation. The major reason that we impose this restriction is to make execution choices easier. We do not have to concern ourselves with a command which only has to finish before some other command. Without duration information it is hard to tell whether that command has to be performed first, or whether it can be started during the other command. If we constrain commands to be disjoint, in the above case the action that had to finish first would become constrained to start first.

```
(→
  (AND
    (command ?command1)
    (command ?command2)
    (NOT (EQ ?command1 ?command2)))
  (TIME-RELATION ?command1 [meets met-by before after] ?command2))
```

6.3 Event Generation

In chapter three we described how events are defined. An event occurs if and only if all its conditions and effects hold or occur. The event definition can be used to predict effects given that the conditions hold. We can express event definitions as inference rules as follows:

```
{Define-event <event-type>
  conditions: <conditions>
  effects: <effects>}
```

allows the inferences:


```
(→  
  (AND  
    (assume <constraint-conditions>  
      <object-conditions>  
      (assume <event>))  
    <event>)
```

```
(→  
  <event>  
  <effects>)
```

Event variables may also be generated as components of new plans. In such a case, there may be a goal-fact variable already specified as the effect of the event. Since events are based on all the assumptions of their conditions and effects, such an event would also have to be based on the assumptions of this given effect. Effect time-relations are derived between this goal-fact variable and the conditions by effect time-relations. The goal-fact variable is not re-derived under the assumptions of the event. The projector should be kept from deriving a second event with the same conditions (and a new fact-variable effect).

6.4 Handling Non-Planning Conflicts

Event inferences in our system are defeasible, as we show above. As far as the planner is concerned, though, such event inferences should be assumed to be correct. It is only if they are observed not to hold that our proposed planner will note that they fail. Any persistence assumption that is in direct conflict with an event inference should therefore be made unlikely so that no inferences will be added based on it.

When a new nogood assumption set is discovered, it will first be checked whether the nogood assumption set describes a planning conflict. If it

describes a planning conflict, the conflict will be handled by a resolve-conflict goal. If it does not describe a planning conflict, the system checks whether a constraint assumption is in direct contradiction with one or more events. If the top-level assumptions of the nogood set contain only one constraint assumption, and one or more event assumptions, the nogood set is a candidate. The low-level assumptions of the nogood set that contradict the constraint assertion are found by removing all necessary assumptions of the constraint assumption and the constraint assumption itself from a copy of the nogood assumption set. If these assumptions contain only event assumptions, we can infer the negation (not) of the constraint under these assumptions. Otherwise the top-level constraint is not in direct conflict with the event(s). Note that since event assumptions remain likely so long as they are possible, the only way that a constraint assumption negated in this way could stop being unlikely is if the event is found to be impossible.

We assume that all event definitions that have conflicting effects have conflicting conditions. If this holds we should never detect a nogood assumption-set with only event top-level assumptions. This is because if some events contradict each other, their conditions also have to contradict each other so a nogood assumption set with fewer assumptions would be detected. If we removed this restriction, we would have to reason about conflicting event expectations, and it might become necessary to make the assumptions of some events uncertain or even unlikely. It seems that the assumption-based framework provides a representation for reasoning about such conflicts, but we will not worry about this problem here. Since event rules are essentially default rules, the problem above is the problem of conflicting defaults.

6.5 Negation Inferences

(NOT p1) cannot be uncertain; it is either likely or unlikely. If (NOT p1) is likely, p1 is unlikely. We use negation assertions carefully, since if they are used incorrectly they could lead to assumptions looping between being likely and unlikely.

The following inferences can be made from NOT assertions:

```
(→  
  (AND  
    (NOT ?p)  
    ?p)  
  Contradiction)
```

```
(→  
  (AND  
    (NOT (TIME-RELATION ?time1 ?relation ?time2))  
    (complement-time-relation ?relation ?complement-time-relation  
      ?time1  
      ?time2)  
    (TIME-RELATION ?time1 ?complement-time-relation ?time2))
```

```
(→  
  (AND  
    (NOT (EQUAL ?object1 ?object2))  
    ?object1  
    ?object2)  
  (NOT-EQUAL ?object1 ?type))
```

```
(→  
  (AND  
    (NOT (NOT-EQUAL ?object1 ?object2))  
    ?object1  
    ?object2)  
  (EQUAL ?object1 ?type))
```

```
(→  
  (AND  
    (NOT (INSTANCE ?object1 ?type))  
    ?object1  
    (NOT-INSTANCE ?object1 ?type))
```

```
(→
  (AND
    (NOT (NOT-INSTANCE ?object1 ?type))
    ?object1)
  (INSTANCE ?object1 ?type))
```

6.6 Predicting Bad Facts

The `define-bad-fact` definition derives a *bad-fact* assertion when a fact is derived which unifies with the bad fact description of the definition.

```
(Define-bad-fact
  restrictions: <restrictions>
  fact: <fact>).
```

is roughly equivalent to the inference:

```
(→
  (AND
    (assume <restrictions>)
    <fact>)
  (BAD-FACT <fact>))
```

If the derived fact only unifies with the described bad fact under some assumptions, the derived fact itself should not be asserted to be *bad*. Instead a new fact non-choice variable is created with new non-choice variables corresponding to the parts of the original which only unified under assumptions. This new fact is asserted to be a bad fact. The fact variable is derived based on the assumptions of the original fact and the unification assumptions. It is constrained to be necessarily temporally equal to the fact variable or constant which triggered the rule. The new non-choice variables are derived under the assumptions of the original variable they correspond to, and the original unification assumption which was generated from that original variable. They are made necessarily equal to the original variable to which they correspond.

Chapter 7

PLAN GENERATION AND SELECTION

7.1 Generating Plans

As we described in section 3.14, the `define-planfor` definition can be used to either define a plan which should be used to solve an existing goal, or to define a top-level plan that should be generated given some conditions. The form of the definition to solve an existing goal is:

```
(define-planfor
  goal: <goal>
  conditions: <conditions>
  plan: <plan>)
```

The form of a planfor definition to generate a top-level goal is:

```
(define-planfor
  goal: top-level
  conditions: <conditions>
  plan: <plan>)
```

Top-level plans have a *dummy* top-level supergoal generated for them. The inference made in either case is the following:

```
(→
  (AND
    (was-derived <goal>)
    (is-likely <conditions>))
  (PLANFOR <goal> <plan>))
```

The `was-derived` expression indicates that a goal was derived which unified with the definition goal specification. This expression succeeds with the assumptions necessary to unify the arguments of a new goal with the goal

specification. The assumption set of the is-likely expression is a set of is-likely assumptions in which each assumption in the assumptions of the conditions is the argument of an is-likely assumption. We require that the conditions be likely because we do not want our task network to be based on inconsistent conditions, and we know that likely assertions are kept globally consistent with the task network. (See section 7.11).

The conditions of a plan type may or may not be specified in the conditions part of a planfor definition producing that type of plan. Each plan condition that is not mentioned in the planfor definition will be described by a new choice variable in any new plan variable created. The new plan variable will have bind-goals generated to bind such choice variables. When a plan condition is specified in the planfor definition, a different plan is generated for each existing fact that matches the condition.

It is when choice variables from the goal appear in plan conditions that there is the most difference between whether or not plan conditions are specified as planfor conditions in planfor definitions. If a planfor condition contains choice variables, it may unify with existing facts under equality assumptions. Hence the planfor assertion may also be based on inequality assumptions. The planfor assertion requires that all its assumptions are likely. An inequality assumption is only likely if it is protected by a plan, or a bind-command is expected to infer it. It is therefore possible for the existing facts which match the planfor conditions to be likely, but for the planfor assertion itself to be unlikely because the binding assumptions are uncertain. If a condition is not specified in the conditions part of the planfor definition, its bind-goal can be solved by a bind-plan that mentions any likely matching fact,

and the inequalities are inferred to be likely by the bind-plan. Therefore when there are choice-variables, not putting plan conditions in the planfor conditions part will allow more solutions to be found.

Varying which plan conditions are mentioned in the planfor definition affects how many plans are produced. For example, if no conditions are specified there is only one plan produced by the planfor definition for each goal. In general there is a plan produced for every combination of derivations of the planfor conditions for every goal. This is because conditions in the *conditions* part of the planfor definition are universally instantiated, while a condition that is only found in the plan is existentially instantiated.

The number of top-level plans produced by a define-planfor rule is a function of the combinations of derivations of the planfor conditions. If no planfor conditions are specified, the planfor rule will only generate one plan.

7.2 Plan Instantiation

When a new plan variable is instantiated by a planfor assertion, new choice variables are created for each propositional argument and each condition not mentioned in the planfor rule. New bind-goals corresponding to these new choice variables become subactions of the new plan.

The preconditions of the plan and its subactions have to be computed. The preconditions of an action are those facts which must hold before the action is permitted to start. When a plan has a protection time-relation that asserts that a fact should start before one of its actions, that fact becomes a precondition of the action. The preconditions of goals are made preconditions of all subplans for those goals. The preconditions of a plan are made the preconditions of all

subactions of the plan. We constrain a fact to necessarily start before the action(s) it is a precondition of. The assumptions of an action include the assumptions of all its preconditions. This asserts that the action will occur only if its preconditions do, and if the action does occur its preconditions will start before it does.

The assumptions of the plan are equal to the union of the assumptions of all its parts. The new parts must of course have their assumptions computed first. New physical value or value choice variable arguments have themselves as their only assumption. A new fact choice variable has itself as an assumption, and is also based on the assumptions of all its arguments. A command is based on its own assumption, the assumptions of its preconditions and typically the assumptions of its arguments. The bind-command is the only exception to this rule - it does not depend on the assumptions of its *variable* argument. Achieve and bind goals are based on the assumptions of their single argument and the assumptions of their preconditions. We will discuss the resolve-conflict-goal and make-disjoint-goal assumptions in section 7.7. They are typically not found in user-defined plans.

The assumptions of a subevent variable consist of itself, the assumptions of its conditions, and the assumptions of any goal-fact effect. Any (non-choice) variable effects of the event not specified in the plan are derived under the assumptions of the event. The effect time-relations are also derived under the assumptions of the event.

Assumption derivations are also made for each of the protection assumptions of the plan. Assumption derivations are discussed in section 4.3.2.

Though the plan and its parts are all derived when the plan is first created by the planfor rule, these derivations will not be added at first because the assumptions of the plan are unlikely. (see sections 4.7 and 5.1).

7.3 Solving Bind Goals and Achieve Goals

A bind-goal has a choice variable argument, which may describe a fact, value or physical object. The bind-goal is solved only if the choice variable becomes bound to a variable or constant which is known to represent an actual object. There is only one type of plan for a bind-goal, and that is the bind-plan. A bind-plan consists of a single bind-command. We can define bind-plans and bind-commands as follows:

```
(define-command bind-command
  arguments:
    (description (object ?variable))
    (object ?value))

(define-plan bind-plan
  arguments:
    (object ?variable)
    (object ?value)
  actions:
    (bind-command ?bind-command
      (variable ?variable) (value ?value)))
```

The *description* expression is used to indicate that the bind-command does not depend on its *variable* argument representing an actual object. The bind-command can only be performed when its *value* argument is known to refer to an actual object. It is derived that if the command occurs that the *variable* argument will equal the *value* argument. We can express this in the following rule:

```
(→
  (AND
    (bind-command (variable ?variable) (value ?value))
    ?variable)
  (EQUAL ?variable ?value))
```

The reason that the condition of the variable needs to be added is that inequalities must always be based on the assumptions of both arguments. According to the above rule, selecting a bind-plan makes a binding hypothesized. If this leads to a conflict, another bind-plan can be chosen instead. When a bind command is executed, it imposes the binding permanently and makes the assumptions of the variable the empty assumption set. Bind-commands can only be executed when there are no goals that require plan choices, in the same way as regular commands. This is a novel approach. PANDORA treats meta-plans differently from regular ones, executing the meta-plans when they are selected.

- Bind plans can be used to solve bind-goals as follows:

```
(define-planfor
  goal: (bind-goal
    (variable ?variable))
  conditions:
    (as-specific ?value ?variable)
    ?value
  plan: (bind-plan
    (variable ?variable)
    (value ?value)))
```

The *as-specific* expression asserts that *value* describes as specific an object as *variable*. If it succeeds, it does so under the empty assumption set. The *value* chosen must be likely.

Achieve goals have a single goal-fact choice variable argument, and are

solved when that goal-fact is known to refer to an actual object. Achieve goals can be solved using a bind-plan in the same way as bind-goals:

```
(define-planfor
  goal: (achieve (goal-fact ?goal-fact))
  conditions:
    . (as-specific ?fact ?goal-fact)
      ?fact
  plan:
    (bind-plan (variable ?goal-fact) (value ?fact)))
```

Achieve goals may also be solved by plans which involve physical action. There are two ways in which such a plan could be used to solve the achieve goal:

1. The goal-fact argument can be described as the effect of a sub-event of the plan. This is the standard way, and it is used in all the examples in our examples section.
2. The goal-fact argument can be described as the goal-fact argument of an achieve goal within the plan. For example, it might be specified that to get to location C, given the conditions that the planner is at location A, the planner should perform some action to get from A to B, then achieve being at location C. It does not seem that there would be any looping problem, due to our constraint that plans with the same propositional arguments must be equal, or before or after each other. No plan with the same propositional arguments can therefore be nested inside another. We give an example of this in section 3.14.

7.4 Generating Prevention Plans

When the planner detects a bad fact, it generates a *Prevent* plan. The bad

fact assertion and prevent plan then contradict, leading to a planning conflict which is resolved by a resolve-conflict-goal. We define the prevent plan and the define-planfor which generates it as follows:

```
(define-plan prevent
  arguments:
    (description (fact ?fact))
  protections:
    (NOT (fact ?fact)))

(define-planfor
  goal: top-level
  conditions:
    (was-derived (BAD-FACT ?fact))
  plan:
    (prevent (fact ?fact)))
```

The was-derived assertion is true if its argument assertion was ever derived under some assumptions. It succeeds with the empty assumption set. The planfor assertion therefore is not justified by the assertion that the bad fact will occur. To do so would lead to the sort of looping problem discussed by McDermott[1982]. (see section 2.8).

7.5 Planning Conflicts

Typically AI planners have detected planning conflicts via special-purpose techniques. The NOAH model then suggests that *critics* be used to remedy the conflicts. We described some of these critics in section 2.2. The problem with critics is that they are typically designed to make only one kind of patch, and do not take the global picture of the conflict into account. It is also difficult to reason about why they are making the modifications that they make.

We propose instead (like Wilensky[1983]) that planning conflicts be solved using resolve-conflict goals. This allows the planner to reason about conflicts

in a more sophisticated way. A major problem with Wilensky[1983]'s PANDORA program, however, is that it has an inadequate representation with which to represent planning conflicts. Planning conflicts are frequently due to the persistences of background facts as well as the plans themselves. Wilensky's method also seems to only be able to handle conflicts between two tasks. We propose a general formalism that can be used to represent planning conflicts which include any number of plans greater than zero, and perhaps other background assertions as well.

We define a planning conflict as a minimum conjunction of one or more plans (and perhaps constraints or other assertions) which describes a contradiction. This is a more general and stronger definition than that which is usually used. This is principally due to our use of time-relation reasoning and assumption-based reasoning. We can detect inconsistent time-relation constraints as contradictions, whereas most planners have to use special-purpose techniques. One of the strengths of our approach is that we can reason about conflicts that arise from all kinds of contradictions in the same way. Besides time-relation contradictions, inequality or instance constraints could lead to contradictions. We will also generate contradictions when a goal has no possible plans, and between a prevent plan and the fact it prevents. An added bonus of our assumption-based reasoning techniques is that only minimum new nogood assumption sets are added. Ones which are supersets of existing ones are not added, so would not be considered as planning conflicts. The system can quickly test whether a new nogood assumption set represents a planning conflict. If this is the case, the underlying planning conflict is determined. If an equal or subset planning conflict was not already

detected, the planner adds the derivation:

```
(Derived
 (PLANNING-CONFLICT <planning-conflict>)
 { })
```

Resolve-conflict goals are generated to resolve planning conflicts. The planner resolves the conflict by making one or more of the assertions in the conflict false. The planner can decide to *avoid* performing any plans found in the conflict, avoid making choice variable inequality constraints, or try to prevent a time-relation persistence from occurring.

7.6 Detecting Planning Conflicts

If we have a new nogood assumption set generated whose top-level assumptions include at least one command, choice variable or planning protection, the nogood assumption set represents a planning conflict. We process the nogood set to identify the planning conflict as follows:

1. Initialize the conflict set to be a copy of the nogood assumption set.
2. For each command and choice variable in the conflict set, remove all of the assumptions of its assertion (except itself) from the conflict set.
3. Add the plans of each remaining command and choice variable to the conflict set.
4. Remove all the assumptions of these plans from the conflict set. This will remove all commands and choice variables from the conflict, but leave all plans in the conflict.
5. Replace a negation assertion with a fact as an argument by the prevent plan that protects it.

6. Replace each is-likely assumption by its argument.
7. Remove all intersection time-relations and identify the top-level conjunction facts. Each intersection time-relation describes an intersection interval, and hence a conjunction fact, by describing the relation between its sub-intervals. The top-level conjunction facts are those described in the conflict which are not sub-facts of any other conjunction fact described in the conflict.
8. For each top-level conjunction fact having n primitive subfacts that is identified, construct n intersection time-relations. Each time-relation should describe an intersection relation between one of the primitive facts of the conjunction, and the intersection of all the other primitive facts of the conjunction. Add these constraints to the conflict set.

We then assert that the resulting conflict is a planning conflict using the planning-conflict relation.

Negation assumptions only occur as plan protections. They may either occur in prevent-fact plans, or in plans to solve resolve-conflict goals. Since we replaced all those that correspond to prevent plans by the corresponding prevent plans in step 5 above, we know that all negation assumptions that remain in the final conflict are protected by plans to solve resolve-conflict goals. We do not allow the planning conflict argument of a resolve-conflict goal to contain negation assumptions. Doing so seems to invariably lead to undesirable looping behavior, as the planner creates contorted plans such as ones to avoid avoiding plans.

If a planning conflict contains some negation assertions, we can generate one or more new planning conflicts without negation assertions by substituting

for each negation assertion all the elements of a planning conflict containing the argument of that negation assertion except the argument itself. For example if we generate:

```
(Derived
  (PLANNING-CONFLICT { a1 a2 (NOT a3) (NOT a4) })
  { })
```

and we already have the following planning conflicts detected:

```
(Derived
  (PLANNING-CONFLICT { a3 a5 a6 })
  { })
```

```
(Derived
  (PLANNING-CONFLICT { a3 a7 })
  { })
```

```
(Derived
  (PLANNING-CONFLICT { a4 a8 })
  { })
```

we can generate the following new planning conflict derivations:

```
(Derived
  (PLANNING-CONFLICT { a1 a2 a5 a6 a8 })
  { })
```

```
(Derived
  (PLANNING-CONFLICT { a1 a2 a7 a8 })
  { })
```

The rule we describe above is a special case of the resolution rule of inference. Since our conflicts are negations of conjunctions (nands), we could also represent them as disjunctions with the arguments negated. The reason that this is a special case of resolution is that we only perform resolution in order to generate conflicts without any negations (or equivalently to generate disjunctions with only negated arguments). We describe the resolution rule of

inference in section 4.5.

Clearly in the worst case we could generate many large conflicts in this way. This is to a large extent because we keep our conflict assumptions in (negated) conjunctive normal form. This makes it easier for resolve-conflict goals to work on them. If we allowed the conflicts to be represented using arbitrary nestings of conjunctions and disjunctions, we could reduce the number of conflicts but would complicate solving the resolve-conflict goal. In any case, in most of the examples solved by conventional planning systems there are a small number of planning conflicts detected. The technique described above would therefore probably only become unwieldy in examples much more complex than those typically handled by conventional planners, in which case more conventional planners would probably have at least as much trouble with them.

If a new planning conflict is equal or a superset of an existing one, it can be ignored. Otherwise, a resolve-conflict plan is generated to solve it. This resolve-conflict plan contains a single resolve-conflict goal with the conflict as its *conflict* argument. We can define the resolve-conflict plan as follows:

```
(define-plan resolve-conflict-plan
  actions:
    (resolve-conflict-goal ?subgoal
      (conflict ?conflict)))
```

The resolve-conflict goal is based on its own assumption, so the resolve-conflict plan is based on this same assumption. The plan for definition to generate a resolve-conflict plan can be defined as follows:

```

(Define-planfor
  goal top-level
  conditions:
    (PLANNING-CONFLICT ?conflict)
    (NOT (Member (NOT ?assertion) ?conflict))
  plan:
    (resolve-conflict-plan
      (conflict ?conflict)))

```

It does not permit the resolve-conflict-goal to solve conflicts containing negation assumptions.

7.7 Solving Resolve-Conflict Goals

In this section we will describe some plans which can be used to solve resolve-conflict goals. We do not mean to suggest that the set of plans given is complete, but it allows the planner to reason about a number of conflicts which arise. For example, we will only consider preventing time-relation conditions in the conflict whose time-relation value is *intersects*, since these are the easiest to solve. Each plan for a resolve-conflict goal is directed at a single assertion in the conflict, except one plan to take advantage of a helpful solution when an assertion in the conflict is already unlikely. The plans directed against a single conflict assertion protect the negation of the assertion they are directed against. They have this argument as their *conflict-assertion* argument.

A plan is proposed to avoid performing each plan in the conflict. Selecting such an avoid plan forces the Plan Selector to make another choice for the supergoal of the avoided plan, if the goal is still likely. (If there is no other selectable plan it will generate a contradiction, and a new planning conflict. We discuss this in section 7.10. We can define the avoid plan, and how it can be used to avoid a plan to solve the resolve-conflict goal as follows:

```
(Define-plan avoid
  protections:
    (NOT ?conflict-assertion))
```

```
(Define-planfor
  goal: (resolve-conflict
    (conflict ?conflict))
  conditions:
    (Member (plan ?plan) ?conflict)
  plan:
    (avoid (conflict-assertion ?plan)))
```

Instance or inequality assertions can also be avoided by protecting their negation, provided that at least one of the arguments is a choice variable.

```
(Define-planfor
  goal: (resolve-conflict
    (conflict ?conflict))
  conditions:
    (Member (EQUAL ?x ?y) ?conflict)
    (OR (choice variable ?x) (choice variable ?y))
  plan:
    (avoid (conflict-assertion (EQUAL ?x ?y))))
```

```
(Define-planfor
  goal: (resolve-conflict
    (conflict ?conflict))
  conditions:
    (Member (NOT-EQUAL ?x ?y) ?conflict)
    (OR (choice variable ?x) (choice variable ?y))
  plan:
    (avoid (conflict-assertion (NOT-EQUAL ?x ?y))))
```

```
(Define-planfor
  goal: (resolve-conflict
    (conflict ?conflict))
  conditions:
    (Member (INSTANCE ?variable ?type) ?conflict)
    (choice variable ?variable)
  plan:
    (avoid (conflict-assertion (INSTANCE ?variable ?type))))
```

```

(Define-planfor
  goal: (resolve-conflict
        (conflict ?conflict))
  conditions:
    (Member (NOT-INSTANCE ?variable ?type) ?conflict)
    (choice-variable ?variable)
  plan:
    (avoid (conflict-assertion (NOT-INSTANCE ?variable ?type))))

```

If the negation of an assertion of a conflict is likely and it is not due to a plan selected for that conflict, there is a helpful solution. A no-op plan can then be used to solve the resolve-conflict goal while these conditions hold. Note that this plan is applicable whatever the reason that the assertion is false or unlikely; its negation does not have to be protected by another plan.

```

(Define-planfor
  goal: (resolve-conflict-goal ?resolve-conflict-goal
        (conflict ?conflict))
  conditions:
    (Member ?conflict-assertion ?conflict)
    (NOT ?conflict-assertion)
    (assume (not-selected-plan-for
              ?resolve-conflict-goal
              (plan (conflict-assertion ?conflict-assertion))))
  plan:
    (no-op-plan))

```

The not-selected-plan-for assumption is likely if the plan is not selected for the goal, and unlikely otherwise. We can make intersection time-relation constraints false by using the make-disjoint-plan.

```

(Define-plan make-disjoint-plan
  descriptions:
    (EQ ?conflict-assertion (time-relation ?time1 intersects ?time2))
  actions:
    (make-disjoint-goal ?subgoal (arg1 ?time1) (arg2 ?time2))
  protections:
    (NOT ?conflict-assertion))

```

```

(Define-planfor
  goal: (resolve-conflict-goal
        (conflict ?conflict))
  conditions:
    (Member (time-relation ?time1 intersects ?time2) ?conflict)
  plan:
    (make-disjoint-plan
      (conflict-assertion (time-relation ?time1 intersects ?time2))))

```

7.8 Solving Make-Disjoint Goals

The following plan can be used to solve the make-disjoint goal. It tries to achieve a fact that is mutually exclusive with one simple fact argument to terminate the persistence of that fact before the other argument occurs. The new conflicting fact is produced by looking at the define-exclusive-facts definitions.

```

(Define-plan achieve-between-plan
  conditions:
    (object ?before)
    (object ?after)
  actions:
    (achieve ?achieve (condition ?goal-fact))
  protections:
    (time-relation ?before starts-before ?goal-fact)
    (time-relation ?goal-fact starts-before ?after))

(Define-planfor
  goal:
    (make-disjoint-goal (time1 (simple-fact ?fact1)) (time2 ?time2))
  conditions:
    (new-mutually-exclusive-fact ?fact1 ?new-fact)
  plan:
    (achieve-between-plan
      (before ?fact1)
      (after ?time2)
      (goal-fact ?new-fact)))

```

7.9 Plan Selection

A plan *plan1* is *proposed* for a goal *goal1* if (PLANFOR *goal1 plan1*) is likely. *plan1* is *selectable* for *goal1* only while (NOT *plan1*) is not likely. For every plan

plan1 we make the derivation:

```
(Derived  
  (NOT plan1)  
  { (not-proposed plan1) })
```

(not-proposed plan1) asserts that all planfor assertions mentioning plan1 are unlikely. This assures that only proposed goals are selectable. In our system, the only plans that are ever proposed for more than one goal are proposed under the empty assumption set. The only plans that can be not-proposed are therefore ones that can only be proposed for one goal.

A goal *requires a plan choice* when it is likely but has no plan selected for it. The Plan Selector keeps track of all the current goals. It distinguishes between those that require plan choices and those that do not. Each goal indexes the planfor assertions which describe its potential plans. While there are no pending derivations in the inference queues, the Plan Selector is allowed to select selectable plans for goals that requires plan choices. There is only at most one plan selected for a given goal at a time.

The commands and protections of a plan remain likely while that plan is selected. When a plan becomes selected for the first time, this will add it to the Knowledge Base.

If a selected plan plan1 stops being selectable because (NOT plan1) becomes likely, that plan is no longer selected. If the super-goal of the plan is still likely, this goal will require a new plan.

If a new plan becomes selectable for a goal which already has a plan selected, the Plan Selector decides whether to select this new one instead of the current one.

7.10 Handling Blocked Goals

A goal is *blocked* if it is likely and there is no selectable plan for it. That is, the negation of each plan that was ever proposed for the goal is either likely or true. If there were never any plans proposed, it is also a case of goal blockage. A goal blockage represents a planning conflict, because the goal cannot be solved if there are no plans with which to solve it. A contradiction is asserted between the goal occurring, the negations of all the plans ever proposed, and the assumption that the plans proposed so far are all the plans that will ever be proposed. This assumption has the form:

```
(all-plans-for-goal <goal> <list-of-plans>)
```

If a new plan is proposed for the goal, this assumption becomes false, and any planning conflict it is part of is solved automatically. If the goal becomes impossible, the assumption becomes true. Otherwise the assumption is likely.

Often the only derivations of plan negations are due to *avoid* plans which protect the negation of the plan explicitly. We give special consideration to bind-plans. If there are protections or bind-commands that have imposed likely inequality constraints on a choice variable, we do not want the Plan Selector to choose bind-plans that contradict these current likely bindings. To enforce this restriction we make the following inference:

```
(→
  (AND
    (is-likely (NOT-EQUAL ?x ?y))
    (PLANFOR ?goal (bind-plan ?bind-plan
      (variable ?x)
      (value ?y)))
    (assume (never-added ?bind-plan)))
  (NOT ?bind-plan))
```

This asserts that no bind-plan that has never been added (and hence never selected) can be selected if it is currently likely that the not-equal assertion that would conflict with it is likely. Since inequalities are only likely when they are protected, this means that a bind-plan that has never been selected cannot be selected if it will result in an immediate planning conflict. The never-added expression succeeds under its own assumption if its argument has never been added to the Knowledge Base. If its argument is added, this assumption becomes false. If its argument is found to be false before it can be added, the assumption becomes true. Otherwise it remains likely.

7.11 Task Network Consistency

We want the current task network to remain consistent. That is, we want the union of the assumptions of all the currently selected plans to be consistent. Under our current approach we know that all selected plans are based on likely assumptions. The protections, commands and choice variables of a plan are likely because the plan is selected, and we require that the conditions provided in planfor rules are also likely. If we can keep all likely assumptions consistent, we can keep the task network consistent.

Actually, we are using the term *consistent* somewhat loosely here. As we noted in section 6.1.1, Allen's transitive time-relation inferences do not guarantee global consistency. If the resolution rule of inference is not used on the nogood assumption sets, the system will not know all the nogood assumption-sets that could be derived (Section 4.5). We should therefore say that we would like the task network to be consistent, but will only be sure of

keeping it *relatively consistent*.

We have formulated the degrees of belief in assumptions so that the only way to get a likely nogood assumption set is if it describes a planning conflict. We cannot have a contradiction only containing event assumptions due to our restrictions on event definitions. In order to try to maintain consistency between all likely assumptions, we only need to concern ourselves with nogood assumption sets all of whose assumptions are likely.

If we allow planfor conditions to be based on *uncertain* conditions, it becomes possible for the condition assumptions of a set of plans to conflict without the plans conflicting at the top-level. Such a contradiction therefore would not be detected as a planning conflict. The simplest solution would be to label all assumptions that are currently condition assumptions of a plan. When a contradiction is detected which consists of such assumptions, and perhaps some always likely assumptions such as event assumptions, the planner can detect that the conditions underlying its plans are inconsistent. It could then decide not to use one or more of these assumptions as condition assumptions, and mark them somehow. Planfor conditions would therefore have to make sure that none of their assumptions were marked in this way.

Our insistence on likely assumptions for planfor conditions imposes the limitation that the planner cannot take advantage of the effects of external events which are based on uncertain persistence assumptions. Another approach to solving the problem might be to make the degree of belief *likely* correspond to a believed likelihood, instead of the somewhat adhoc meaning we attribute to it. Persistence assumptions could be likely by default, and only become uncertain or unlikely due to conflicts with other assumptions.

7.12 Copying Choice Variables

There is a potential problem with our binding approach as presented when it comes to executing plans. Imagine that a plan contains a choice variable, and passes this choice variable through one or more levels of plans. A bind command in one of these lower-level plans is then executed, binding this choice variable permanently. If this lower-level plan were to lead to success of the higher plan there would be no problem. If this lower level plan fails, however, the binding that it made is not undone, and the rest of the task network has an unnecessary constraint imposed upon it.

One solution to this problem is to replace any propositional argument choice variables of a plan that are inherited from its supergoal with new choice variable *copies*. These variable copies are then constrained to be equal to the originals under the assumptions of the new plan. If the plan succeeds, any such bindings would be permanently inherited by the *originals*. If the plan does not occur, no permanent bindings would be imposed on the *originals*. It also seems to be important to make any necessary relations of the copied choice variables also be necessary relations on the copies. This could not be done by the equality constraint mentioned above, but would instead need to be handled specially as follows: whenever a new necessary constraint is imposed on a choice variable, the same necessary constraint is imposed on all its copies.

7.13 Replanning

Replanning is necessary when unexpected events occur during execution

which affect the task network. These events can be either helpful or harmful. Most of the replanning abilities of our proposed system follow naturally from the representation and the abilities described in the previous sections of the chapter. If a bad event occurs, it will typically cause one of the persistence assumptions of a plan to become impossible, and hence the plan to become impossible. This will automatically make the Plan Selector look for another plan to solve its super-goal. A positive interaction will typically cause a new plan to be proposed for some goal. The best type of helpful interaction occurs when a helpful fact matches the goal-fact of an existing achieve goal. This causes a bind-plan to be proposed for the achieve goal. If a new plan is proposed for a goal that already has a plan selected for it, the Plan Selector decides whether to select the new one, or keep selecting the old one. If a new plan is proposed for a goal that was abandoned because there were no selectable plans for it, the new plan will make the all-plans-for assumption false. This will provide a helpful solution to the resolve-conflict-goal which caused the (plan of the) goal to be avoided. The goal no longer has to be avoided, so the new plan can be considered.

The only additional ability our planner needs with respect to replanning is the ability to propose copies of partially-executed failed plans. What this involves is generating a new plan using the same planfor rule with the same planfor conditions as the failed one. Not doing this causes possible solutions to be missed.

CHAPTER 8

Command Execution and Event Detection

8.1 The Command Executor

The command executor is responsible for starting the execution of commands. It maintains two lists:

1. *The executable commands.* These command variables are derived under only their own assumption. This means that all their arguments have been bound, and all their preconditions have occurred. An executable command must be likely. There must also be no likely time-relation constraint that the command starts after *now*.
2. *The non-executable enabled commands.* These commands must also be derived under only their own assumption. Their assumption set cannot be nogood. One of the last two conditions of being an executable command must not be satisfied.

The executor is notified when the assumptions of a command variable contain only itself. The executor then puts the command variable in the appropriate list. The executor is also notified when any of the other conditions change. This could lead to a command moving from one of the lists above to the other.

When there are no pending inferences in the inference queues and there are no plans to select, the executor is permitted to make a command choice. If there is at least one executable command, it will pick a command. It first makes the command assumption known to be true. This removes it from all

assumption sets it is part of. The Event Detector is then notified that a command is starting.

Ordering choices between unordered actions are not made during the planning process. Instead temporal constraints accumulate which might lead to the actions becoming ordered. The only ordering choices are made with respect to which command should be executed next.

There are cases in which simulation of the orderings of the actions during the planning process would be useful. Such simulation might even detect inconsistencies in the global task network. As we noted in section 6.1.1, Allen[1983a]'s time-relation constraints do not guarantee global consistency. Our approach is just the simplest one, and seems to be adequate for many problems.

8.2 The Event Detector

The event detector keeps track of:

1. *The current executing command.* This is the command which is currently executing. If the planner is doing nothing, an *idle* command is executing. We constrain all commands to be disjoint, so having an *idle* command automatically constrains hypothesized commands to start after any idle interval.
2. *The current observed fact constants.* These refer to facts known to be true now.
3. *The now time-interval.*

The event detector is activated when fact change(s) occur and/or a command starts executing. Note that the executor is not responsible for

stopping commands. We assume that the *code* of the command stops itself. When a command stops, an *idle* command starts. All these changes can be described as changes to the current-executing-command and current-observed-facts of the event detector. When such changes occur, the event detector does the following:

```

oldnow ← now
now ← new time interval
infer ( (Derived (TIME-RELATION now in-or-equal current-executing-command) { }) )
infer ( (Derived (TIME-RELATION oldnow [meets] now) { }) )
LOOP FOR fact IN current-observed-facts DO
  infer ( (Derived (TIME-RELATION now in-or-equal fact) { }) )
  IF fact is a new fact THEN
    infer ( (Derived fact { }) )
  ENDIF
ENDLOOP

```

8.3 Matching Hypothetical Facts Against Observed Facts

If the system predicts that an event will occur, and that event does occur, the effects of the event should be bound to the observed facts which they describe. We can have either-choice variables or non-choice variables as the effects of events. A choice variable is the effect of an action only if the event is part of a plan, and the choice-variable is the goal-fact of an achieve super-goal of the plan.

When a non-choice fact variable is constrained to be necessarily equal to an observed fact constant, we bind it to that observed fact constant. A choice-variable can be bound to an observed fact constant when it becomes equal to that constant under its own assumptions and the assumption of an event in which it is the effect. Two facts are equal only if they are temporally equal, and their arguments are equal. Binding a variable to a constant or

variable which is known to represent an actual object causes the assumption set of the variable to become the empty assumption set.

8.4 Inferring That Event Variables Describe Actual Events

The event described by an event variable actually occurs when all its conditions and events are known to occur. This cannot be determined strictly on the basis of the assumptions of the event. The event will become derived under only its own assumption when all its conditions occur, but this does not necessarily mean that the event has occurred because some of its effects may not occur.

Chapter 9

EXAMPLES

9.1 The Door Example

9.1.1 The Problem

This example illustrates in a very simple domain the basic idea of how our planner works, and especially how it replans in the light of unexpected events. As with all our examples, we will show all objects produced, but will show only the relevant time-relation constraints derived. We do not show any of the assumption derivations of constraints. There are too many time-relation constraints imposed to show them all.

The robot is initially outside a room. It has the goal of being inside the room. The door that leads into the room is open. The planner proposes a plan to enter the room. This plan consists of a goal to achieve the door being open and a command to enter the room once the door is open. The robot proposes to solve the door-open goal with a bind-plan which takes advantage of the fact that the door is already open. At this point a complete task network has been created.

We will first show how execution proceeds if the door stays open. First the robot executes a bind-command to bind the door-open goal to the initial door

open fact. This solves the goal of the door being open. The robot then executes the command to enter the room. The robot is then inside the room and the top-level goal is solved.

If the door closes by itself before the robot executes any commands, the bind-plan fails, and an open-door plan is proposed to solve the open door goal instead.

If the door closes after the door-open goal is solved but before the enter room command is executed, the enter-room plan becomes impossible, so fails. Another plan of the same type is proposed to solve the goal of being inside the room.

The enter room plan will also fail if the command to enter the room does not have the proper effects. For example, if it does not change the location of the robot.

9.1.2 The Domain Definition

The representation used in this example is very simplistic. The fact types `inside-room` and `outside-room` refer to the position of the robot. These are the only possible locations of the robot. They are mutually exclusive: the robot is either `inside-room` or `outside-room`, but not both. There are two mutually exclusive door states: `door-open` and `door-closed`.

```
(define-fact inside-room)
```

```
(define-fact outside-room)
```

```
(define-exclusive-facts
```

```
  facts:
```

```
    (inside-room)
```

```
    (outside-room))
```

```
(define-fact door-closed)
```

```
(define-fact door-open)
```

```
(define-exclusive-facts
```

```
  facts:
```

```
    (door-closed)
```

```
    (door-open))
```

Open door command, event and plan.

```
(define-command open-door-command)
```

```
(define-event open-door-event
```

```
  conditions:
```

```
    (door-closed ?door-closed)
```

```
    (open-door-command ?open-door-command)
```

```
    (TIME-RELATION ?open-door-command in-or-equal ?door-closed)
```

```
  effects:
```

```
    (door-open ?door-open)
```

```
    (TIME-RELATION ?open-door-command [meets] ?door-open))
```

```
(define-plan open-door-plan
```

```
  conditions:
```

```
    (door-closed ?door-closed)
```

```
  subactions:
```

```
    (open-door-command ?open-door-command)
```

```
  event:
```

```
    (open-door-event ?open-door-event
```

```
      (door-closed ?door-closed)
```

```
      (open-door-command ?open-door-command)
```

```
      (door-open ?door-open)))
```

```
(define-planfor
```

```
  goal: (achieve (goal-fact (door-open ?door-open)))
```

```
  conditions:
```

```
    (door-closed ?door-closed)
```

```
  plan:
```

```
    (open-door-plan
```

```
      (door-closed ?door-closed)
```

```
      (door-open ?door-open)))
```

Enter room command, event and plan.

```
(define-command enter-room-command)

(define-event enter-room-event
  conditions:
    (outside-room ?outside-room)
    (door-open ?door-open)
    (enter-room-command ?enter-room-command)
    (TIME-RELATION ?enter-room-command in-or-equal ?door-open)
    (TIME-RELATION ?enter-room-command in-or-equal ?outside-room)
  effects:
    (inside-room ?inside-room)
    (TIME-RELATION ?enter-room-command [meets] ?inside-room))

(define-plan enter-room-plan
  conditions:
    (outside-room ?outside-room)
  subactions:
    (achieve ?achieve-door-open
      (goal-fact (door-open ?door-open)))
    (enter-room-command ?enter-room-command)
  events:
    (enter-room-event ?enter-room-event
      (outside-room ?outside-room)
      (door-open ?door-open)
      (enter-room-command ?enter-room-command)
      (inside-room ?inside-room)))

(define-planfor
  goal:
    (achieve
      (goal-fact (inside-room ?inside-room)))
  conditions:
    (outside-room ?outside-room)
  plan:
    (enter-room-plan
      (outside-room ?outside-room)
      (inside-room ?inside-room)))
```

9.1.3 Generating the Task Network

The robot is initially outside the door, which is open. The robot is executing the idle command. now1 is the initial now interval.

```
(Derived
  (outside-door outside-door1)
  { })
```

```
(Derived
  (TIME-RELATION now1 in-or-equal outside-door1)
  { })
```

```
(Derived
  (door-open door-open1)
  { })
```

```
(Derived
  (TIME-RELATION now1 in-or-equal door-open1)
  { })
```

```
(Derived
  (idle idle1)
  { })
```

```
(Derived
  (TIME-RELATION now1 in-or-equal idle1)
  { })
```

The robot has the initial goal of being inside the room. Where the goal came from is not relevant to this example.

```
(Derived
  (achieve $achieve-inside-room1
    (goal-fact $inside-room1))
  { $inside-room1 })
```

```
(Derived
  (inside-room $inside-room1)
  { $inside-room1 })
```

```
(Derived
  (TIME-RELATION now1 [before meets] $inside-room1)
  { $inside-room1 })
```

There is only one plan proposed for this goal

```
(Derived
  (PLANFOR $achieve-inside-room1 $enter-room-plan1)
  { })
```

The Plan is selected.

```
(Derived
  (enter-room-plan $enter-room-plan1
    (achieve-door-open $achieve-door-open2)
    (outside-room outside-room1)
    (inside-room $inside-room1)
    (enter-room-command $enter-room-command1)
    (enter-room-event $enter-room-event1))
  { $inside-room1
    $door-open2
    $enter-room-event1
    $enter-room-command1
    (TIME-RELATION $enter-room-command1 in-or-equal $door-open2)
    (TIME-RELATION $enter-room-command in-or-equal outside-room1)  })
```

```
(Derived
  (enter-room-event $enter-room-event1
    (outside-room outside-room1)
    (inside-room $inside-room1)
    (enter-room-command $enter-room-command1))
  { $inside-room1
    $door-open2
    $enter-room-event1
    $enter-room-command1
    (TIME-RELATION $enter-room-command1 in-or-equal $door-open2)
    (TIME-RELATION $enter-room-command in-or-equal outside-room1)  })
```

```
(Derived
  (achieve $achieve-door-open2
    (goal-fact $door-open2))
  { $door-open2  })
```

```
(Derived
  (door-open $door-open2)
  { $door-open2  })
```

The subactions of a plan are temporally in-or-equal their plan. For the enter-room-plan plan type, we can further restrict the achieve-door-open goal to start the plan, and the enter-room-command command to finish the plan.

```
(Derived
  (TIME-RELATION $achieve-door-open2 [starts] $enter-room-plan1)
  { $inside-room1
    $door-open2
```

```

$enter-room-event1
$enter-room-command1
(TIME-RELATION $enter-room-command1 in-or-equal $door-open2)
(TIME-RELATION $enter-room-command in-or-equal outside-room1)  ))

```

```

(Derived
  (TIME-RELATION $enter-room-command1 [finishes] $enter-room-plan1 )
  {
    $inside-room1
    $door-open2
    $enter-room-event1.
    $enter-room-command1
    (TIME-RELATION $enter-room-command1 in-or-equal $door-open2)
    (TIME-RELATION $enter-room-command in-or-equal outside-room1)  ))

```

\$enter-room-command1 is protected to be in-or-equal \$door-open2 and outside-room1. These are the preconditions of the command. We only allow a command to be started once its preconditions are known to hold. \$enter-room-command1 therefore only occurs if \$door-open2 and outside-room1 occur. This makes it depend on the assumptions underlying these two conditions. It will start after these conditions start.

```

(Derived
  (enter-room-command $enter-room-command1)
  {
    $door-open2
    $enter-room-command1  })

```

```

(Derived
  (TIME-RELATION $door-open2 starts-before $enter-room-command1)
  {
    $door-open2
    $enter-room-command1  })

```

\$achieve-door-open2 has only one plan proposed for it, which is chosen.

```

(Derived
  (PLANFOR $achieve-door-open2 $bind-plan1)
  {  })

```

```

(Derived
  (bind-plan $bind-plan1
    (variable $door-open2)
    (value door-open1)

```

```
(bind-command $bind-open2-open1))
{ $bind-open2-open1
  $door-open2  })
```

```
(Derived
 (bind-command $bind-open2-open1
  (variable $door-open2)
  (value door-open1))
 { $bind-open2-open1  })
```

If the \$bind-open2-open1 meta-command is executed, \$door-open2 will become equal to door-open1. It is asserted that they are temporally equal under the assumptions of \$bind-open2-open1 and \$door-open2. It would also be inferred that any arguments that they might have would also be equal under these assumptions, but in this example these facts have no arguments.

```
(Derived
 (TIME-RELATION door-open1 [equals] $door-open2)
 { $bind-open2-open1
   $door-open2  })
```

9.1.4 Normal Execution

The task network required to solve the top-level goal of getting into the room is now solved, since there are no more goals requiring plans to be chosen for them. (This also means that there are no unresolved conflicts, since the existence of unresolved conflicts would imply the existence of resolve-conflict meta-goals requiring a plan choice). The system now looks for an executable command. The command \$bind-open2-open1 is executable, since it is derived under only its own assumption. The Command Executor selects \$bind-open2-open1 to be executed. The Event Detector creates a new now

interval now2 and asserts that the facts which held in now2 still hold. (see sections 8.1 and 8.2).

```
(Derived
 (TIME-RELATION now1 [meets] now2)
 { })
```

```
(Derived
 (TIME-RELATION now2 in-or-equal outside-room1)
 { })
```

```
(Derived
 (TIME-RELATION now2 in-or-equal door-open1)
 { })
```

We infer that \$bind-open2-open1 describes an actual command by removing its assumption from all assumption sets. (see section 4.6). We also infer that it is temporally equal to now2.

```
(Derived
 $bind-open2-open1
 { })
```

```
(Derived
 (TIME-RELATION $bind-open2-open1 [equal] now2)
 { })
```

Bind-commands have the effect of making their variable argument equal to their value argument, so \$door-open2 becomes bound to door-open1 and known to describe an actual fact. We then remove its assumption from the assumption sets in which it occurs. This will make the following variables known to represent actual objects :

```
$door-open2
$achieve-door-open2
$bind-plan1
```


In addition, the following time-relation becomes known to hold:

```
(TIME-RELATION door-open1 [equals] $door-open2)
```

Since \$door-open2 is now known to describe an actual fact, \$enter-room-command1 becomes executable, since it is based only on the assumption of itself. The planner decides to execute it. This requires creating a new now interval now3. now3 is asserted to be in-or-equal the current sensory facts.

```
(Derived  
  (TIME-RELATION now2 [meets] now3)  
  {  })
```

```
(Derived  
  (TIME-RELATION now3 in-or-equal outside-room1)  
  {  })
```

```
(Derived  
  (TIME-RELATION now3 in-or-equal door-open1)  
  {  })
```

It is asserted that \$enter-room-command1 describes an actual command and starts with now3 :

```
(Derived  
  $enter-room-command1  
  {  })
```

```
(Derived  
  (TIME-RELATION now3 [starts equals] $enter-room-command1)  
  {  })
```

If the command has the effects which \$enter-room-event1 predicts, the robot will detect that it has stopped executing \$enter-room-command1 at the same time

that it detects that it is in the room. We will consider what happens if all goes as expected. First we create a new now interval, now4. We infer that door-open persists. We also create a new fact inside-room3 and infer that it now holds. This will constrain outside-room1 to meet now4, since the robot cannot be inside and outside the room at the same time. We create an idle command to indicate that the robot is no longer executing any commands. This constrains \$enter-room-command1 to be equal to now3, since commands are constrained to be disjoint.

```
(Derived
  (TIME-RELATION now3 [meets] now4)
  { })
```

```
(Derived
  (TIME-RELATION now4 in-or-equal door-open1)
  { })
```

```
(Derived
  (inside-room inside-room3)
  { })
```

```
(Derived
  (TIME-RELATION now4 in-or-equal inside-room3)
  { })
```

```
(Derived
  (idle idle2)
  { })
```

```
(Derived
  (TIME-RELATION now4 in-or-equal idle2)
  { })
```

\$inside-room1 is constrained to be met-by \$enter-room-command1, under the single assumption of \$enter-room-event1. inside-room2 is known to be met-by

~~\$enter-room-command1~~. This constrains \$inside-room1 and inside-room2 to intersect under the assumptions of \$inside-room1 and the assumption \$enter-room-event1. Given that they have equal descriptions, and the above constraint, we can bind \$inside-room1 to inside-room2. (see section 8.3).

```
(Derived
  $inside-room1
  ( ))
```

```
(Derived
  (EQUAL $inside-room1 inside-room2)
  ( ))
```

We can remove the assumption \$inside-room1 from the assumption-sets in which it occurs, since it is known to describe an actual fact. An effect of this is to make \$achieve-inside-room1 known to occur (be solved). The following assumptions also become known to necessarily hold, and can also be removed from the assumption sets in which they are found :

```
(TIME-RELATION $enter-room-command1 in-or-equal door-open1)
(TIME-RELATION $enter-room-command1 in-or-equal outside-room1)
```

If all the conditions and effects of the event become known, the event is known to occur. (See section 8.4). Its assumption is removed from all assumption sets of which it is an element.

The final result is that all the objects that were mentioned so far in this example are known to describe actual objects. The following diagram illustrates the final known time-relations between these objects:

Diagram 9.1 : The final ordering of facts and actions if no unexpected events occur.

9.1.5 Replanning

Spontaneous world changes may be detected by the planner before it has created a complete task network, or during execution. In this section, we will consider how the planner would react if different unexpected events occurred.

If the door closes by itself before any commands are executed,

\$door-open1 will be constrained to necessarily precede \$enter-room-command1. There will therefore be a contradiction between the plan to bind \$door-open2 to door-open1 (\$bind-plan1), the protection that \$enter-room-command1 be temporally in-or-equal \$door-open2, and this new constraint. We therefore get a planning conflict between \$enter-room-plan1 and \$bind-plan1. Since \$bind-plan1 is a plan for a subgoal of \$enter-room-plan1, the planner will avoid \$bind-plan1 instead of \$enter-room-plan1. The new closed door fact also satisfies the precondition of the open-door-plan definition, and an open-door-plan is proposed to solve \$achieve-door-open1.

If the door closes after \$achieve-door-open1 occurs, but before \$enter-room-command1 is executed, \$enter-room-plan1 becomes impossible. This is because the protection between \$door-open1 (which refers to an actual fact) and \$open-door-command1 is necessarily false. When a partially-executed plan fails, a new copy of the plan with new subactions but the same conditions is generated to try to solve the goal. In this case a plan to achieve \$inside-room1 would be generated with outside-room1 as its outside-room condition. See section 7.13.

\$enter-room-plan1 would also fail if \$enter-room-command1 did not have the expected effects. For example, it might have no effects. It was inferred under the assumptions of \$enter-room-event1 that \$enter-room-command1 would temporally meet \$door-open1. If there is no effect, these assumptions will be found to be nogood. This will delete \$enter-room-plan1, since its assumptions include those of

~~Enter-room-event1~~. As in the previous case described, another enter-room-plan could be proposed for the goal.

9.2 Nell and Train Example

9.2.1 The Problem

This example is based on an example described in McDermott [1982]. In his example Nell is a maiden tied up on some railway tracks where a train is expected to arrive. This creates the expectation that Nell will be killed. The planner does not want Nell to be killed, so has to generate a plan of action to save her. The important feature of the example is that the planner cannot prevent Nell from getting killed by deciding *not to perform some action*. This example is beyond the ability of all conventional STRIPS or NOAH-type planners, which have no conception of undesirable facts, let alone how to prevent them. PANDORA (Wilensky[1983], Faletti[1982]) can reason about bad facts, but can only handle planning conflicts between tasks. In this case, there is only one task, namely to prevent Nell from getting killed. In addition it does not seem that McDermott[1982] has a program which can solve this problem, though he briefly proposes a method. (See section 2.8 for a brief description of his approach).

Initially Nell is alive and on the tracks. It is predicted (via an event definition) that the train train1 will arrive on the tracks. It is then predicted that Nell will be run over and killed by the train, under the assumptions that Nell is on the tracks and alive when the train arrives. A prevent plan called \$prevent-Nell-dead1 is generated. It contradicts the fact asserting that Nell will be killed. The

contradiction is a planning conflict, so a resolve-conflict goal \$resolve-conflict-goal1 is generated to resolve the planning conflict. The planning conflict is between \$prevent-Nell-dead1, the assumption that Nell will be alive and on the tracks with train1, and the assumption that if Nell is alive and on the tracks with train1 she will be killed. There are four plans proposed to resolve this conflict.

1. Keep Nell from being alive while she is on the tracks with the train. The way to solve this plan would be to kill Nell before the train arrives. This plan will be avoided if it is chosen, because it leads to another conflict with another plan to prevent Nell getting killed. It would also be avoided if it were selected because the planner does not know how to kill a living thing.
2. Keep the train from being on the tracks while Nell is alive and on the tracks. The way to solve this plan would be to delay the train from arriving until after Nell stops being alive on the tracks. The planner does not know how to do this, so if it is chosen it will soon be avoided.
3. Keep Nell from being on the tracks while she is alive and the train is on the tracks. The way to solve this plan is to get Nell off the tracks before the train arrives. This is the plan which the planner uses in our example.
4. Avoid preventing Nell getting killed. In other words, let her be killed. This plan would only be used as a last resort.

The goal of the third plan can be solved via an achieve-between plan to

achieve a fact that is mutually exclusive with Nell being on the tracks before train1 arrives. In this case, there is only one such plan generated.: to achieve Nell being off the tracks before train1 arrives. A regular plan for removing objects from tracks is selected for this achieve goal. The task network is then complete. To save Nell, the planner executes the plan to remove her from the tracks.

We also consider the case in which it is discovered that Nell is no longer on the tracks before the train arrives, and before any plan is executed to save her. This is recognized as a helpful solution which causes the existing plan to save her to be discarded.

9.2.2 The Domain Definition

We only distinguish between two locations : "on-tracks" and "off-tracks". We are referring to a particular stretch of tracks, so asserting that an object is "off-tracks" only means that it is not on that particular stretch of tracks. An object cannot be both "on-tracks" and "off-tracks" at the same time.

```
(define-fact on-tracks
  arguments:
    (physical-object ?object))
```

```
(define-fact off-tracks
  arguments:
    (physical-object ?object))
```

```
(define-exclusive-facts
  facts:
    (on-tracks
      (object ?object))
    (off-tracks
      (object ?object)))
```

We define trains and people as our only types of physical objects.

```
(define-physical-object train)
```

```
(define-physical-object person)
```

A person may be alive or dead, but not both at the same time.

```
(define-fact alive
  (person ?person))
```

```
(define-fact dead
  (person ?person))
```

```
(define-exclusive-facts
  facts:
    (alive
      (person ?person))
    (dead
      (person ?person)))
```

A killed-by-train event asserts that if a person is alive on the tracks with a train, that person will become killed while on the tracks with the train. This event type definition will predict that a person gets killed whether the person or train is on the tracks first.

```
(define-event killed-by-train
  conditions:
    (and ?alive-on-tracks-with-train
      (subfact1
        (alive ?alive
          (person ?person)))
      (subfact2
        (and ?on-tracks-with-train
          (subfact1
            (on-tracks
              (object (train ?train))))
          (subfact2
            (on-tracks
              (object ?person)))))))
  effects:
    (dead ?dead
```

```
(person ?person))  
(TIME-RELATION ?alive-on-tracks-with-train [meets] ?dead))
```

The following event type definition triggers the initial expectation that train1 will arrive. For every time interval that train1 is not on the tracks, this definition will create an event predicting that it will be on the tracks immediately following that interval. If we added the converse rule that train1 will eventually get off the tracks once it is on, our proposed system would get into a loop generating on-track and off-track facts ad infinitum, since all these event type definitions are used in a forward-chaining manner. Since we do not have the converse rule we are safe. We use the particular train instance train1 because we have not addressed the issue of allowing non-choice physical object or value variables to be generated by plans, though it seems that such extensions could be made.

```
(define-event train-arrives  
  conditions:  
    (off-tracks ?train-off-tracks  
      (object train1))  
  effects:  
    (on-tracks ?train-on-tracks  
      (object train1))  
    (TIME-RELATION ?train-off-tracks [meets] ?train-on-tracks))
```

When a remove-from-tracks-command command is executed on an object that is on-tracks, that object will become off-tracks. The remove-from-tracks-command cannot be performed on trains.

```

(define-command remove-from-tracks-command
  arguments:
    (physical-object ?object)
  restrictions:
    (NOT-INSTANCE ?object train))

(define-event remove-from-tracks-event
  conditions:
    (remove-from-tracks-command ?remove-from-tracks-command
      (object ?object))
    (on-tracks ?on-tracks
      (object ?object))
    (TIME-RELATION ?remove-from-tracks-command in-or-equal ?on-tracks)
  effects:
    (off-tracks ?off-tracks
      (object ?object))
    (TIME-RELATION ?remove-from-tracks-command [meets] ?off-tracks))

```

The remove-from-tracks-plan is a plan to achieve an object being off-tracks that is initially on-tracks. It is essentially composed of a remove-from-tracks event.

```

(define-plan remove-from-tracks-plan
  arguments:
    (physical-object ?object)
  conditions:
    (on-tracks ?on-tracks
      (object ?object))
  subactions:
    (remove-from-tracks-command ?remove-from-tracks-command)
  events:
    (remove-from-tracks-event ?remove-from-tracks-event
      (remove-from-tracks-command ?remove-from-tracks-command)
      (on-tracks ?on-tracks)
      (off-tracks ?off-tracks)))

(define-planfor
  goal:
    (achieve
      (goal-fact (off-tracks ?object-off-tracks
        (object ?object))))
  conditions:
    (on-tracks ?on-tracks
      (object ?object))
  plan:
    (remove-from-tracks-plan

```

```
(on-tracks ?on-tracks)
(off-tracks ?off-tracks)))
```

We say that it is undesirable for a person to be dead. This will make the planner try to prevent people from getting killed.

```
(define-bad-fact
 fact:
  (dead
   (object (person))))
```

9.2.3 Generating The Task Network

The physical objects of our example are Nell, train1, and self. self refers to the planner.

```
(Derived
 (train train1)
 { })

(Derived
 (person Nell)
 { })

(Derived
 (person self)
 { })
```

The initial conditions are that Nell is alive and on the tracks, self is alive and off the tracks, and train1 is off the tracks. now1 is the initial now time-interval.

```
(Derived
 (alive Nell-alive1
  (object Nell))
 { })

(Derived
 (TIME-RELATION now1 in-or-equals Nell-alive1)
 { })
```

```
(Derived
  (on-tracks Nell-on-tracks1
    (object Nell))
  { })
```

```
(Derived
  (TIME-RELATION now1 in-or-equal Nell-on-tracks1)
  { })
```

```
(Derived
  (alive self-alive1
    (object self))
  { })
```

```
(Derived
  (TIME-RELATION now1 in-or-equal self-alive1)
  { })
```

```
(Derived
  (off-tracks self-off-tracks1
    (object self))
  { })
```

```
(Derived
  (TIME-RELATION now1 in-or-equal self-off-tracks1)
  { })
```

```
(Derived
  (off-tracks train-off-tracks1
    (object train1))
  { })
```

```
(Derived
  (TIME-RELATION now1 in-or-equal train-off-tracks1)
  { })
```

The expectation that train1 will arrive is generated.

```
(Derived
  (train-arrives $train-arrives1
    (train-off-tracks1 train-off-tracks1)
    (train-on-tracks $train-on-tracks1))
  { $train-arrives1 })
```

```
(Derived
  (on-tracks $train-on-tracks1
    (object train1))
  { $train-arrives1 })
```

```
(Derived
  (TIME-RELATION train-off-tracks1 [meets] $train-on-tracks1)
  ( $train-arrives1 ))
```

The expectation that the train will run over and kill Nell is generated. The conjunction facts \$Nell-on-tracks-with-train1 and \$Nell-alive-on-tracks-with-train1 are constructed to serve as conditions of this event.

```
(Derived
  (and $Nell-on-tracks-with-train1
    (subfact1 Nell-on-tracks1)
    (subfact2 $train-on-tracks1))
  ( $train-arrives1
    (TIME-RELATION $train-on-tracks1 intersects Nell-on-tracks1) ))
```

```
(Derived
  (and $Nell-alive-on-tracks-with-train1
    (subfact1 Nell-alive1)
    (subfact2 $Nell-on-tracks-with-train1))
  ( $train-arrives1
    (TIME-RELATION $train-on-tracks1 intersects Nell-on-tracks1)
    (TIME-RELATION Nell-alive1 intersects $Nell-on-tracks-with-train1) ))
```

```
(Derived
  (killed-by-train $skilled-by-train1
    (alive Nell-alive1)
    (train-on-tracks $train-on-tracks1)
    (person-on-tracks Nell-on-tracks1)
    (on-tracks-with-train $Nell-on-tracks-with-train1)
    (dead $Nell-dead1))
  ( $train-arrives1
    (TIME-RELATION $train-on-tracks1 intersects Nell-on-tracks1)
    (TIME-RELATION Nell-alive1 intersects $Nell-on-tracks-with-train1)
    $skilled-by-train1 ))
```

```
(Derived
  (dead $Nell-dead1
    (object Nell))
  $skilled-by-train1 assumptions )
```

```
(Derived
  (TIME-RELATION Nell-alive1 [meets] $Nell-dead1)
  $skilled-by-train1 assumptions )
```

Since it was defined to be a bad fact for people to be dead, we derive the

following relation :

```
(Derived
  (BAD-FACT $Nell-dead1)
  $skilled-by-train1 assumptions )
```

This causes a prevent-fact plan to be generated. This plan protects the assertion that \$Nell-dead1 does not occur.

```
(Derived
  (prevent-fact $prevent-Nell-dead1
    (bad-fact $Nell-dead1))
  { (NOT $Nell-dead1) })
```

\$prevent-Nell-dead1 contradicts \$Nell-dead1 leading to the following nogood assumption set being generated :

```
(NOGOOD { $train-arrives1
  (TIME-RELATION $train-on-tracks1 intersects Nell-on-tracks1)
  (TIME-RELATION Nell-alive1 intersects $Nell-on-tracks-with-train1)
  $skilled-by-train1
  (NOT $Nell-dead1) })
```

This satisfies the criteria of being a planning conflict because (NOT \$Nell-dead1) is a top-level assumption in the conflict, and is protected by a plan. This leads to a planning conflict. The persistence assumptions underlying the conjunction-fact \$Nell-alive-on-tracks-with-train1 are broken into three persistence assumptions, each of which describes one of the primitive facts in the conjunction intersecting the conjunction of the other facts. These types of persistence assumptions lend themselves to being solved by make-disjoint plans in a uniform way. The way that conjunction facts are built out of subcomponents is fairly arbitrary, so the time-relations which are found in the

nogood set are inconvenient to reason about. A resolve-conflict plan is generated, which contains a resolve-conflict goal to resolve the planning conflict.

```
(Derived
  (PLANNING-CONFLICT
    { (TIME-RELATION Nell-alive1 intersects $Nell-on-tracks-with-train1)
      (TIME-RELATION $train-on-tracks1 intersects $Nell-alive-on-tracks1)
      (TIME-RELATION Nell-on-tracks1 intersects $Nell-alive-and-train-on-tracks)
      $prevent-Nell-dead1 })
    { })

(Derived
  (resolve-conflict-plan $resolve-conflict-plan 1
    (subgoal $resolve-conflict-goal1))
    { $resolve-conflict-goal1 })

(Derived
  (resolve-conflict $resolve-conflict-goal1
    (conflict
      { (TIME-RELATION Nell-alive1 intersects $Nell-on-tracks-with-train1)
        (TIME-RELATION $train-on-tracks1 intersects $Nell-alive-on-tracks1)
        (TIME-RELATION Nell-on-tracks1 intersects $Nell-alive-and-train-on-tracks)
        $prevent-Nell-dead1 )))
    { $resolve-conflict-goal1 })
```

The following plans are proposed for \$resolve-conflict-goal1:

```
(Derived
  (PLANFOR
    $resolve-conflict-goal1
    (make-disjoint-plan $Nell-alive-disjoint-Nell-on-tracks-with-train-plan1
      (subgoal (make-disjoint-goal $make-Nell-alive-disjoint-Nell-on-tracks-with-train-goal1
        (arg1 Nell-alive1 )
        (arg2 $Nell-on-tracks-with-train1 ))))
    { })

(Derived
  (PLANFOR
    $resolve-conflict-goal1
    (make-disjoint-plan $make-train-on-tracks-disjoint-Nell-alive-on-tracks-plan1
      (subgoal (make-disjoint-goal $make-train-on-tracks-disjoint-Nell-alive-on-tracks-goal1
        (arg1 $train-on-tracks1 )
        (arg2 $Nell-alive-on-tracks1 ))))
    { })
```

```

(Derived
  (PLANFOR
    $resolve-conflict-goal1
    (make-disjoint-plan $Nell-on-tracks-disjoint-Nell-alive-and-train-on-tracks-plan1
      (subgoal
        (make-disjoint-goal $Nell-on-tracks-disjoint-Nell-alive-and-train-on-tracks-goal1
          (arg1 Nell-on-tracks1 )
          (arg2 $Nell-alive-and-train-on-tracks ))))
      ))

```

```

(Derived
  (PLANFOR
    $resolve-conflict-goal1.
    $avoid-prevent-Nell-dead)
  { })

```

The third plan (to make Nell off the tracks while she is alive and the train is on the tracks) is selected.

```

(Derived
  (and $Nell-alive-and-train-on-tracks1.
    (subfact1 Nell-alive1)
    (subfact2 $train-on-tracks1))
  { $train-arrives1
    (TIME-RELATION Nell-alive1 intersects $train-on-tracks1) })

```

```

(Derived
  $Nell-on-tracks-disjoint-Nell-alive-and-train-on-tracks-plan1.
  { (NOT (TIME-RELATION Nell-on-tracks1 intersects $Nell-alive-and-train-on-tracks1 )) })

```

```

(Derived
  $Nell-on-tracks-disjoint-Nell-alive-and-train-on-tracks-goal1
  { (NOT (TIME-RELATION Nell-on-tracks1 intersects $Nell-alive-and-train-on-tracks1 )) })

```

We get one plan proposed for the goal and it is selected.

```

(Derived
  (PLANFOR
    $Nell-on-tracks-disjoint-Nell-alive-and-train-on-tracks-goal1
    (achieve-between-plan $achieve-between1
      (before-time Nell-on-tracks1)
      (after-time $Nell-alive-and-train-on-tracks1)
      (achieve-goal $achieve-Nell-off-tracks1)))
    { (NOT (TIME-RELATION Nell-on-tracks1 intersects $Nell-alive-and-train-on-tracks1 )) })

```

```
(Derived
  (achieve-between-plan $achieve-between-plan1
    (before-time Nell-on-tracks1)
    (after-time $Nell-alive-and-train-on-tracks1)
    (achieve-goal $achieve-Nell-off-tracks))
  { $train-arrives1
    $Nell-off-tracks1
    (TIME-RELATION $Nell-off-tracks1 starts-before $Nell-alive-and-train-on-tracks1)) })
```

```
(Derived
  (off-tracks $Nell-off-tracks
    (object $Nell))
  { $Nell-off-tracks })
```

```
(Derived
  (achieve $achieve-Nell-off-tracks
    (goal-fact $Nell-off-tracks))
  { $Nell-off-tracks })
```

A single plan is proposed to solve this goal, and it is selected.

```
(Derived
  (PLANFOR $achieve-Nell-off-tracks $rescue-Nell1)
  { $Nell-off-tracks })
```

```
(Derived
  (remove-from-tracks-plan $rescue-Nell1
    (on-tracks Nell-on-tracks1)
    (command $remove-from-tracks-command1)
    (event $remove-from-tracks-event1)
    (off-tracks $Nell-off-tracks1))
  { $remove-from-tracks-command1
    $remove-from-tracks-event1
    $Nell-off-tracks1 })
```

```
(Derived
  (remove-from-tracks-command $remove-from-tracks-command1
    (object Nell))
  { $remove-from-tracks-command1 })
```

```
(Derived
  (remove-from-tracks-event $remove-from-tracks-event1
    (object Nell)
    (command $remove-from-tracks-command1)
    (on-tracks Nell-on-tracks1)
    (off-tracks $Nell-off-tracks1))
  { $remove-from-tracks-command1
    $remove-from-tracks-event1
```

\$Nell-off-tracks1))

We now have a complete task network to try to prevent Nell from getting killed. The command \$remove-from-tracks-command1 is executable. \$remove-from-tracks-command1 is constrained to be temporally equal to its plan (\$rescue-Nell), and \$rescue-Nell is constrained to be equal to its supergoal (\$achieve-Nell-off-tracks). Since \$achieve-Nell-off-tracks is protected by \$achieve-between-plan1 to start before \$Nell-alive-and-train-on-tracks, \$remove-from-tracks-command1 will be constrained to precede \$Nell-alive-and-train-on-tracks under all the assumptions of the time-relations described above.

Without duration reasoning the planner has no way of knowing whether it has time to execute this command within this time frame or not. It will therefore always assume it can.

9.2.3 Execution

The planner can start executing \$remove-from-tracks-command1. If Nell gets off the tracks before train1 arrives, the make-disjoint plan and the resolve-conflict plans are solved. It is a peculiarity of the achieve-between plan that it is only solved if the train actually comes. Whether or not the train comes will not affect the resolve-conflict goal in this case, however.

If Nell becomes off the tracks spontaneously before the planner can execute

his plan to remove her, the \$skilled-by-train1 event becomes impossible, and the \$resolve-conflict-goal1 is spontaneously solved. The plan to save her is no longer selected, so will not be executed.

If for any reason another plan to rescue Nell becomes applicable before the planner has completed executing the plan produced in the example, the Plan Selector would compare it with the currently selected plan and perhaps decide to use it instead.

9.3 The Raincoat Example

9.3.1 The Problem

This example is based on an example by Wilensky [1983]. On a particular morning a goal for the planner to know what is going on in the world is generated. The usual plan to satisfy this goal is to go outside, fetch the morning newspaper, then come back inside and read it. On this particular morning it is raining. The planner knows that if it is raining and he is outside without wearing a raincoat, he will get wet. The planner always wants to keep from getting wet. The problem is how the planner could satisfy his goal of knowing what is going on in the world, while satisfying his goal of not getting wet. Wilensky[1983] handles this by considering the conflict between a preservation goal of staying dry and the plan of going outside, and proposing a domain-specific meta-plan to solve the conflict. A plan to resolve conflicts between going outside to fetch newspapers and keeping dry is stored, consisting of putting on a raincoat before going out.

We will take a different approach. Instead of storing a domain-specific meta-plan we will solve the problem using only domain-specific plans stored to solve achieve goals, and domain-independent meta-plans to solve meta-goals. The ability of our planner to handle conflicts in this way provides greater flexibility, since it allows conflicts to be handled for which it has no domain-specific rules. Wilensky[1983] describes a Change-Circumstances

meta-plan which might be similar to what we are doing here, though he is somewhat sketchy as to how it works in his system.

9.3.2 The Domain Definition

```
(define-value time-of-day-value
  range: [morning afternoon night])
```

```
(define-fact time-of-day
  arguments:
    (time-of-day-value ?value))
```

```
(define-exclusive-facts
  restrictions:
    (NOT-EQUAL ?value1 ?value2)
  facts:
    (time-of-day (value ?value1))
    (time-of-day (value ?value2)))
```

```
(define-value weather-value
  range: [raining not-raining])
```

```
(define-fact weather
  arguments:
    (weather-value ?value))
```

```
(define-exclusive-facts
  restrictions:
    (NOT-EQUAL ?value1 ?value2)
  facts:
    (weather (value ?value1))
    (weather (value ?value2)))
```

```
(define-value wetness-value
  range: [wet dry])
```

```
(define-fact wetness
  arguments:
    (physical-object ?object)
    (wetness-value ?value))
```

```
(define-exclusive-facts
  restrictions:
    (NOT-EQUAL ?value1 ?value2)
  facts:
    (wetness (object ?object) (value ?value1))
    (wetness (object ?object) (value ?value2)))

(define-fact self-inside)

(define-fact self-outside)

(define-exclusive-facts
  facts:
    (self-inside)
    (self-outside))

(define-fact know-whats-going-on)

(define-fact not-know-whats-going-on)

(define-exclusive-facts
  facts:
    (know-whats-going-on)
    (not-know-whats-going-on))

(define-fact wearing-raincoat)

(define-fact not-wearing-raincoat)

(define-exclusive-facts
  facts:
    (wearing-raincoat)
    (not-wearing-raincoat))
```

The get-wet-in-rain event definition describes the expectation that a person who is dry and outside in the rain without a raincoat will get wet. (See section 3.12 for a more in-depth treatment of this event).


```

(define-event get-wet-in-rain
  conditions:
    (and ?outside-in-rain-without-raincoat
      (subfact1
        (and
          (subfact1
            (self-outside))
          (subfact2
            (weather (value raining))))))
      (subfact2
        (not-wearing-raincoat)))
    (and ?dry-outside-in-rain-without-raincoat
      (subfact1
        ?outside-in-rain-without-raincoat)
      (subfact2
        (wetness
          (object ?person)
          (value dry)))))
  effects:
    (dryness ?person-wet
      (object ?person)
      (value wet))
    (TIME-RELATION ?dry-outside-in-rain-without-raincoat [meets] ?person-wet)
    (TIME-RELATION ?outside-in-rain-without-raincoat [overlaps] ?person-wet))

```

The top-level plan for knowing what is going on in the morning.

```

(define-plan top-level-know-plan
  preconditions:
    (time-of-day ?morning-time
      (value morning))
  subactions:
    (achieve ?achieve-know-whats-going-on
      (goal-fact (know-whats-going-on ?know-whats-going-on)))
  protections:
    (TIME-RELATION ?know-whats-going-on in-or-equal ?morning-time))

(define-planfor
  goal: top-level-know-plan
  conditions:
    (time-of-day ?morning-time (value morning))
  plan:
    (top-level-know-plan
      (morning-time ?morning-time)))

```

Following are the fetch-and-read-newspaper command, event and plan

definitions. Executing the fetch-and-read-newspaper command puts the planner outside (to fetch the paper), inside again (to read the paper), then makes the planner know whats going on (after reading the paper). We could have decomposed this command into subparts, but chose to use only one command for simplicity.

```
(define-command fetch-and-read-newspaper-command)
```

```
(define-event fetch-and-read-newspaper-event
```

```
  conditions:
```

```
    (self-inside ?self-inside1)
```

```
    (not-know-whats-going-on ?not-know-whats-going-on)
```

```
    (fetch-and-read-newspaper-command ?fetch-and-read-newspaper-command)
```

```
    (TIME-RELATION ?fetch-and-read-newspaper-command starts-during ?self-inside1)
```

```
    (TIME-RELATION
```

```
      ?fetch-and-read-newspaper-command in-or-equal ?not-know-whats-going-on)
```

```
  effects:
```

```
    (know-whats-going-on ?know-whats-going-on)
```

```
    (TIME-RELATION
```

```
      ?fetch-and-read-newspaper-command [meets] ?know-whats-going-on)
```

```
    (self-outside ?self-outside)
```

```
    (TIME-RELATION ?self-inside1 [meets] ?self-outside)
```

```
    (TIME-RELATION ?self-outside [during] ?fetch-and-read-newspaper-command)
```

```
    (self-inside ?self-inside2)
```

```
    (TIME-RELATION ?self-outside [meets] ?self-inside2))
```

```
(define-plan fetch-and-read-newspaper-plan
```

```
  preconditions:
```

```
    (self-inside ?self-inside)
```

```
    (not-know-whats-going-on ?not-know-whats-going-on)
```

```
  subactions:
```

```
    (fetch-and-read-newspaper-command ?fetch-and-read-newspaper-command)
```

```
  subevents:
```

```
    (fetch-and-read-newspaper-event ?fetch-and-read-newspaper-event
```

```
      (self-inside1 ?self-inside)
```

```
      (not-know-whats-going-on ?not-know-whats-going-on)
```

```
      (know-whats-going-on (know-whats-going-on ?know-whats-going-on))))
```

```

(define-planfor
  goal: (achieve (goal-fact (know-whats-going-on ?know-whats-going-on)))
  conditions:
    (self-inside ?self-inside)
    (not-know-whats-going-on ?not-know-whats-going-on)
  plan:
    (fetch-and-read-newspaper-plan
      (self-inside ?self-inside)
      (not-know-whats-going-on ?not-know-whats-going-on)
      (know-whats-going-on ?know-whats-going-on)))

```

The command, event and plan for putting on a raincoat :

```

(define-command put-on-raincoat-command)

(define-event put-on-raincoat-event
  conditions:
    (not-wearing-raincoat ?not-wearing-raincoat)
    (put-on-raincoat-command ?put-on-raincoat-command)
    (TIME-RELATION ?put-on-raincoat-command in-or-equal ?not-wearing-raincoat)
  effects:
    (wearing-raincoat ?wearing-raincoat)
    (TIME-RELATION ?put-on-raincoat-command [meets] ?wearing-raincoat))

(define-plan put-on-raincoat-plan
  preconditions:
    (not-wearing-raincoat ?not-wearing-raincoat)
  subactions:
    (put-on-raincoat-command ?command)
  events:
    (put-on-raincoat-event ?put-on-raincoat-event
      (not-wearing-raincoat ?not-wearing-raincoat)
      (put-on-raincoat-command ?command)
      (wearing-raincoat ?wearing-raincoat)))

(define-planfor
  goal: (achieve (goal-fact (wearing-raincoat ?wearing-raincoat)))
  conditions:
    (not-wearing-raincoat ?not-wearing-raincoat)
  plan:
    (put-on-raincoat-plan
      (not-wearing-raincoat ?not-wearing-raincoat)
      (wearing-raincoat ?wearing-raincoat)))

```

We define the planner getting wet as a bad fact.

```
(define-bad-fact
  fact:
    (wetness
      (object self)
      (value wet)))
```

9.3.3 Generating the Task Network

The initial conditions are that the planner does not know what is going on in the world, it is morning, the planner is not wearing a raincoat, the planner is inside and dry and it is raining. The planner is also idle. (i.e. is not executing any command.)

```
(Derived
  (person self)
  { })
```

```
(Derived
  (not-know-whats-going-on not-know1)
  { })
```

```
(Derived
  (TIME-RELATION now1 in-or-equal not-know1)
  { })
```

```
(Derived
  (time-of-day morning1
    (value morning))
  { })
```

```
(Derived
  (TIME-RELATION now1 in-or-equal morning1)
  { })
```

```
(Derived
  (not-wearing-raincoat not-wearing-raincoat1)
  { })
```

```
(Derived
```

```

    (TIME-RELATION now1 in-or-equal not-wearing-raincoat1)
    { })
  (Derived
    (self-inside self-inside1)
    { })

  (Derived
    (TIME-RELATION now1 in-or-equal self-inside1)
    { })

  (Derived
    (wetness self-dry1
      (object self)
      (value dry))
    { })

  (Derived
    (TIME-RELATION now1 in-or-equal self-dry1)
    { })

  (Derived
    (weather raining1
      (value raining))
    { })

  (Derived
    (TIME-RELATION now1 in-or-equal raining1)
    { })

  (Derived
    (idle idle1)
    { })

  (Derived
    (TIME-RELATION now1 in-or-equal idle1)
    { })

```

The top-level plan is generated.

```

  (Derived
    (top-level-know-plan $top-level-know-plan1
      (morning-time morning1)
      (achieve-know-whats-going-on $achieve-know1))
    { $know1
      (TIME-RELATION morning1 [overlaps contains finished-by] $know1 ) })

  (Derived
    (know-whats-going-on $know1)

```

```

    { $know1 })

(Derived
  (achieve $achieve-know1
    (goal-fact $know1))
  { $know1 })

```

The plan \$fetch-plan1 is proposed for \$achieve-know1. It is the only plan, so is selected..

```

(Derived
  (PLANFOR $achieve-know1 $fetch-plan1)
  { })

(Derived
  (fetch-and-read-newspaper-plan $fetch-plan1
    (not-know-whats-going-on not-know1)
    (self-inside1 self-inside1)
    (know-whats-going-on $know1)
    (self-outside $self-outside1)
    (self-inside2 $self-inside2)
    (fetch-and-read-newspaper-event $fetch-event1)
    (fetch-and-read-newspaper-command $fetch-command1))
  { $know1
    $fetch-event1
    $fetch-command1
    (TIME-RELATION $fetch-command1 starts-during self-inside1)
    (TIME-RELATION $fetch-command1 in-or-equal not-know1) })

(Derived
  (fetch-and-read-newspaper-event $fetch-event1
    (not-know-whats-going-on not-know1)
    (self-inside1 self-inside1)
    (know-whats-going-on $know1)
    (self-outside $self-outside1)
    (self-inside2 $self-inside2)
    (fetch-and-read-newspaper-command $fetch-command1))
  { $know1
    $fetch-event1
    $fetch-command1
    (TIME-RELATION $fetch-command1 starts-during self-inside1)
    (TIME-RELATION $fetch-command1 in-or-equal not-know1) })

(Derived
  (fetch-and-read-newspaper-command $fetch-command1)
  { $fetch-command1 })

```

\$fetch-event1 has the following effects:

```
(Derived
  (TIME-RELATION $fetch-command1 [meets] not-know1)
  { assumptions of $fetch-event1 })

(Derived
  (self-outside $self-outside1)
  { assumptions of $fetch-event1 })

(Derived
  (TIME-RELATION self-inside1 [meets] $self-outside1)
  { assumptions of $fetch-event1 })

(Derived
  (TIME-RELATION $self-outside1 [during] $fetch-command1)
  { assumptions of $fetch-event1 })

(Derived
  (self-inside $self-inside2)
  { assumptions of $fetch-event1 })

(Derived
  (TIME-RELATION $self-outside1 [meets] $self-inside2)
  { assumptions of $fetch-event1 })
```

It is predicted that the planner might get wet:

```
(Derived
  (and $outside-in-rain1
    (subfact1 $self-outside1)
    (subfact2 raining1))
  { $know1
    $fetch-event1
    $fetch-command1
    (TIME-RELATION $fetch-command1 starts-during self-inside1)
    (TIME-RELATION $fetch-command1 in-or-equal not-know1)
    (TIME-RELATION $self-outside1 intersects raining1)})

(Derived
  (and $outside-in-rain-without-raincoat1
    (subfact1 $outside-in-rain1)
    (subfact2 not-wearing-raincoat1))
  { $know1
    $fetch-event1 .
```

```

$fetch-command1
(TIME-RELATION $fetch-command1 starts-during self-inside1)
(TIME-RELATION $fetch-command1 in-or-equal not-know1)
(TIME-RELATION $self-outside1 intersects raining1)
(TIME-RELATION $outside-in-rain1 intersects not-wearing-raincoat1))

```

(Derived

```

  (and $dry-outside-in-rain-without-raincoat1
    (subfact1 $outside-in-rain-without-raincoat1)
    (subfact2 self-dry1))
  {
    $know1
    $fetch-event1
    $fetch-command1
    (TIME-RELATION $fetch-command1 starts-during self-inside1)
    (TIME-RELATION $fetch-command1 in-or-equal not-know1)
    (TIME-RELATION $self-outside1 intersects raining1)
    (TIME-RELATION $outside-in-rain1 intersects not-wearing-raincoat1
    (TIME-RELATION $outside-in-rain-without-raincoat1 intersects self-dry1) ))

```

(Derived

```

  (get-wet-in-rain $get-wet-in-rain1
    (outside-in-rain-without-raincoat $outside-in-rain-without-raincoat1)
    (dry-outside-in-rain-without-raincoat $dry-outside-in-rain-without-raincoat1)
    (wet $self-wet1))
  {
    $know1
    $fetch-event1
    $fetch-command1
    (TIME-RELATION $fetch-command1 starts-during self-inside1)
    (TIME-RELATION $fetch-command1 in-or-equal not-know1)
    (TIME-RELATION $self-outside1 intersects raining1)
    (TIME-RELATION $outside-in-rain1 intersects not-wearing-raincoat1
    (TIME-RELATION $outside-in-rain-without-raincoat1 intersects self-dry1)
    $get-wet-in-rain1 ))

```

(Derived

```

  (wetness $self-wet1
    (object self)
    (value wet))
  { assumptions of $get-wet-in-rain1 })

```

(Derived

```

  (TIME-RELATION $dry-outside-in-rain-without-raincoat1 [meets] $self-wet1)
  { assumptions of $get-wet-in-rain1 })

```

(Derived

```

  (TIME-RELATION $outside-in-rain-without-raincoat1 [meets] $self-wet1)
  { assumptions of $get-wet-in-rain1 })

```


It is derived that \$self-wet1 is a bad-fact. This generates a prevent plan. The prevent plan contradicts \$self-wet and leads to a nogood set being generated.

```
(Derived
  (BAD-FACT $self-wet1)
  ( assumptions of $get-wet-in-rain1 ))
(Derived
  (prevent $prevent-self-wet1)
  ( (NOT $self-wet1) ))

(NOGOOD
  ( $know1
    $fetch-event1
    $command1
    (TIME-RELATION $fetch-command1 starts-during self-inside1)
    (TIME-RELATION $fetch-command1 in-or-equal not-know1)
    (TIME-RELATION $self-outside1 intersects raining1)
    (TIME-RELATION $outside-in-rain1 intersects not-wearing-raincoat1)
    (TIME-RELATION $outside-in-rain-without-raincoat1 intersects self-dry1)
    $get-wet-in-rain1
    (NOT $self-wet1) ))
```

The top-level assumptions of this NOGOOD assumption set are:

```
( $get-wet-in-rain1 (NOT $self-wet1) -).
```

This is a planning conflict since (NOT \$self-wet1) is protected by a plan. In analyzing the conflict to produce the planning conflict set, new time-relation constraints are produced to describe the \$dry-outside-in-rain-without-raincoat1 fact conjunction in terms of the intersections of each simple fact part with the conjunction of all the other parts. A resolve conflict plan is generated to resolve the planning conflict.

```
(Derived
  (Planning-conflict
    { (TIME-RELATION $self-outside1 intersects $dry-no-raincoat-raining1)
      (TIME-RELATION not-wearing-raincoat1 intersects $outside-dry-raining1)
```

```

    (TIME-RELATION dry1 intersects $outside-no-raincoat-raining1)
    (TIME-RELATION raining1 intersects $outside-no-raincoat-dry)
    $get-wet-in-rain1
    $prevent-self-wet1 ))
  { })

(Derived
  (resolve-conflict-plan $resolve-conflict-plan1
    (subgoal $resolve-conflict-goal1))
  { $resolve-conflict-goal1 })

(Derived
  (resolve-conflict-goal $resolve-conflict-goal1
    (conflict
      (TIME-RELATION $self-outside1 intersects $dry-no-raincoat-raining1)
      (TIME-RELATION not-wearing-raincoat1 intersects $outside-dry-raining1)
      (TIME-RELATION dry1 intersects $outside-no-raincoat-raining1)
      (TIME-RELATION raining1 intersects $outside-no-raincoat-dry)
      $get-wet-in-rain1
      $prevent-self-wet1 )))
  { $resolve-conflict-goal1 })

```

\$resolve-conflict-goal1 has several plans proposed to solve it. Each is intended to make one of the planning conflict assertions false.

The first plan is to make \$self-outside1 disjoint from the other subfacts of \$dry-outside-in-rain-without-raincoat1. It is intended to make the first element of the planning conflict false. Intuitively it could be solved by delaying \$fetch-command1 until after raining1, but our planner has no such plan. Plans to delay facts only seem useful for planners that can reason about time-durations.

```

(Derived
  (PLANFOR
    $resolve-conflict-goal1
    (make-disjoint-plan $self-outside1-disjoint-dry-no-raincoat-raining-plan1
      (subgoal $self-outside1-disjoint-dry-no-raincoat-raining-goal1)
      (arg1 $self-outside1)
      (arg2 $dry-no-raincoat-raining1)))
  { })

```

The second plan corresponds to the second element of the planning conflict. It is intended to make not-wearing-raincoat1 disjoint from the other subfacts. It is the plan which we will discuss later (and provides the solution Wilensky[1983] suggests).

```
(Derived
  (PLANFOR
    $resolve-conflict-goal1
    (make-disjoint-plan $not-wearing-raincoat1-disjoint-outside-dry-raining-plan1
      (subgoal $not-wearing-raincoat1-disjoint-outside-dry-raining-plan1)
      (arg1 not-wearing-raincoat1)
      (arg2 $outside-dry-raining1)))
  ( ))
```

The third plan requires making self-dry1 not hold while the other facts hold. Its subgoal could be solved by getting wet before going outside, but our planner has no plan to get wet. This will lead to the goal of getting wet being in a planning conflict with the assumption that there are no plans to get wet. (See section 7.10). The only way to resolve this conflict would be to avoid the get wet plan (the plan that the get wet goal is in). This in turn will lead to there being no selectable plan for \$not-wearing-raincoat1-disjoint-outside-dry-raining-goal1. The plan that this goal is part of will therefore conflict with the assumptions that 1) the only plan for this goal is the plan to get wet before going outside; and 2) that there are no plans for the goal to get wet. \$not-wearing-raincoat1-disjoint-outside-dry-raining-plan1 will therefore be avoided. If the planner had a plan to get itself wet, selecting this plan would only lead to a

conflict with another plan to keep self from getting wet. This would presumably be solved by avoiding the plan to get wet.

```
(Derived
  (PLANFOR $resolve-conflict-goal1
    (make-disjoint-plan $dry1-disjoint-outside-no-raincoat-raining-plan1
      (subgoal $dry1-disjoint-outside-no-raincoat-raining-goal1)
      (arg1 self-dry1)
      (arg2 $outside-no-raincoat-raining)))
  { })
```

The fourth plan could be solved by making it stop raining. This is clearly beyond the abilities of the planner. If selected, this plan would soon be avoided because there would be no plans to solve the goal to achieve a *not-raining* goal fact. The *not-raining* goal fact would be part of the *achieve-between* plan proposed for \$dry1-disjoint-outside-no-raincoat-raining-goal1.

```
(Derived
  (PLANFOR
    $resolve-conflict-goal1
    (make-disjoint-plan $raining1-disjoint-outside-no-raincoat-dry-plan1
      (arg1 raining1)
      (arg2 $outside-no-raincoat-dry1)))
  { })
```

The following plan is to avoid \$fetch-plan1. If this were chosen, the planner would have to look for another plan to solve \$achieve-know1. The planner has no other plans for the goal. A conflict would then be generated between: 1) the assumption that \$fetch-plan1 is the only plan for \$achieve-know, 2) \$stop-level-know-plan1, and 3) all the members of the planning conflict of \$resolve-conflict-goal1 except \$fetch-plan1. (See section 7.10). The planner

might choose to solve the resolve-conflict goal for this conflict by choosing an *avoid* plan to avoid \$stop-level-know-plan1 or by choosing one of the *make-disjoint* plans.

```
(Derived
  (PLANFOR
    $resolve-conflict-goal1
    (avoid $avoid-fetch-plan1
      (plan $fetch-plan1)))
  { })
```

The last plan is to avoid preventing \$self-wet1; i.e. to let \$self-wet occur. This would only be used as a last resort.

```
(Derived
  (PLANFOR
    $resolve-conflict-goal1
    (avoid $avoid-prevent-self-wet1
      (plan $prevent-self-wet1)))
  { })
```

The second plan is chosen to solve the resolve-conflict goal:

```
(Derived
  (and $self-dry-and-raining1
    (subfact1 self-dry1)
    (subfact2 raining1))
  { })
```

```
(Derived
  (and $outside-dry-raining1
    (subfact1 $self-outside1)
    (subfact2 $self-dry-and-raining1))
  { (TIME-RELATION $self-outside1 intersects $self-dry-and-raining1) })
```

```
(Derived
  (make-disjoint-plan $not-wearing-raincoat1-disjoint-outside-dry-raining-plan1
    (arg1 not-wearing-raincoat1)
    (arg2 $outside-dry-raining1)
    (subgoal $not-wearing-raincoat1-disjoint-outside-dry-raining-goal1 ))
  { (NOT (TIME-RELATION not-wearing-raincoat1 intersects $outside-dry-raining1)) })
```

```
(Derived
  (make-disjoint-goal $not-wearing-raincoat1-disjoint-outside-dry-raining-goal1
    (arg1 not-wearing-raincoat1)
    (arg2 $outside-dry-raining1))
  { (NOT (TIME-RELATION not-wearing-raincoat1 intersects $outside-dry-raining1)) })
```

The following plan is proposed to solve the make-disjoint goal above. It is the only one, so is selected:

```
(Derived
  (PLANFOR
    $not-wearing-raincoat1-disjoint-outside-dry-raining-goal1 $achieve-between1)
  { })
```

```
(Derived
  (achieve-between $achieve-between1
    (before not-wearing-raincoat1)
    (after $outside-dry-raining1)
    (subgoal $achieve-wearing-raincoat1)
    (goal-fact $wearing-raincoat1))
  { (TIME-RELATION not-wearing-raincoat1 starts-before $wearing-raincoat1)
    (TIME-RELATION $wearing-raincoat1 starts-before $outside-dry-raining)
    (TIME-RELATION $self-outside1 intersects $self-dry-and-raining1) })
```

```
(Derived
  (achieve $achieve-wearing-raincoat1
    (goal-fact $wearing-raincoat1))
  { $wearing-raincoat1 })
```

```
(Derived
  (wearing-raincoat $wearing-raincoat1)
  { $wearing-raincoat1 })
```

A *put-on-raincoat-plan* is proposed for \$achieve-wearing-raincoat1, and selected. The task network is then solved.

The *put-on-raincoat-command* of this plan is constrained to meet the effect of \$wearing-raincoat1 by this plan. Since \$wearing-raincoat1 is protected to precede \$outside-dry-raining1, the *put-on-raincoat-command* is also protected to precede

\$outside-dry-raining1. This will constrain it to precede *\$fetch-command1*, because *\$outside-dry-raining1* holds during *\$fetch-command1*. Both the *put-on-raincoat-command* and *\$fetch-command1* are based only on their own assumptions, so would have been given to the Command Executor as potentially executable commands. Since *\$fetch-command1* is constrained to be after the current now according to protected assumptions, only the *put-on-raincoat-command* would be executable. The raincoat is therefore to be put on before the planner goes outside.

8.3.4 Execution

The following figure illustrates how execution would proceed if all occurred as expected. *wearing-raincoat2*, *\$self-inside3* and *know2* are observed facts:

Figure 9.2 The final ordering of facts and actions if no unexpected events occur.

8.3.5 Replanning

If it stops raining before the planner puts on its raincoat, this would provide

a helpful solution to `$resolve-conflict-goal1`, and the plan to put on the raincoat could be discarded. If it becomes the afternoon before the planner has figured how to solve the problem, this would cause the top-level plan to fail, because the top-level plan specified that the planner must know what is going on before the end of the morning. In such a case, `$fetch-plan1` would be discarded. The planner would therefore no longer expect to get wet, so the plan to put on the raincoat could also be discarded.

Chapter 10

FUTURE WORK AND CONCLUSIONS

10.1 Summary

We have described a knowledge representation formalism in which goals, commands, plans, events and facts can be described. We treat all the above entities uniformly as objects. Our temporal representation uses the time-interval as the primitive temporal unit. Each fact and action is associated with a time-interval. Time-relation constraints can be imposed between time-intervals using a method described by Allen[1983a]. Inequality and instance constraints can be imposed on objects. We describe a method for describing events in a flexible manner. Events can be described which are not the result of actions on the part of the planner. The time over which conjunctions of facts hold can be described using time-intervals defined as the intersection of the sub-intervals.

We integrate this representation with assumption-based reasoning methods based on the ideas of DeKleer[1984a,1986]. This allows reasoning about an uncertain future. Variables are used to refer to hypothesized objects. Each variable is associated with an assumption set, which describes conditions under which it refers to an actual object. This allows the system to reason about the future which might occur. It does not permit further derivations to be made when variables are known not to represent actual objects, unlike the Situational Calculus/Possible Worlds approaches. Our approach does, however, allow the distinction between hypothesized and actual objects to be

made more clearly.

We designed a metaplaner that uses the world model described above. The use of the above world model allows conflicts which are partly due to the persistence of background facts to be reasoned about more effectively. In particular the planner is able to plan to prevent facts which are not the results of its own actions. The metaplaner also has bind metagoals for binding choice variables.

Finally we have shown how temporal constraints can be used for solving problems involving time. Problems involving time are not well known or understood. Our examples provide an important contribution towards their understanding, much like the early solution techniques to parallel problem solving contributed to a theory of parallel programming.

10.2 Implementation

The system described in this thesis has not been implemented. Our focus has been less on computational issues and more on representational ones. We have not ignored implementation considerations altogether, though, and will discuss some of these briefly.

The primary advantage offered by our technique is that a large portion of the work is performed by the assumption-based reasoning and constraint reasoning modules. This simplifies the complexity of the other modules. Our use of metaplaning techniques further simplifies the planner, particularly with respect to reasoning about planning conflicts.

It seems that the combination of a constraint-based approach and assumption-based reasoning should also lead to more predictable behavior on

the part of the system.

It may be, however, that the techniques as described will lead to an implementation which is much too inefficient. The bottlenecks of the implementation are of course the assumption-based reasoning and constraint propagation discussed above.

Allen[1983a]'s time-relation approach requires $O(n^2)$ time-relations between n time-intervals. This leads to a large number of constraints being propagated. In our system things could be even worse. This is because we allow multiple derivations of time-relation constraints. Our policy of having only one assumption set per time-relation is important in reducing this factor, as is our use of *necessary relations*. It is still hard to predict how many derivations of relations between two time-intervals will occur. An implementation would be necessary to test this point. Allen[1983a,1986] proposes methods to reduce the space and time-requirements of his approach using *reference intervals* and *date lines*. These types of approaches clearly seem to be necessary. Such approaches seem promising. The addition of such techniques would most likely not change the organization of the rest of the system, but only increase the efficiency of the time-relation inferencing. The main point, however, is that such techniques could be added without affecting the existing design.

The other bottleneck is the assumption-based reasoner. The operation which is performed most often is the union of assumption sets. It is clearly important to optimize this operation. DeKleer[1984a,1986] suggests either a bit-array or sorted cdr-coded list representation.

3 OF/DE 3

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

10.3 Future Work

Perhaps the greatest advantage of our approach is its potential for extensibility. Many systems are based on inadequate temporal world models which limit their extensibility. We make various world assumptions, but they are not for the most part necessary consequences of the underlying representation.

The most important extension to our model would be to incorporate temporal duration reasoning. As we have noted several times in the thesis, this is a crucial ability to solve any useful problems. It seems that this might be done using techniques described by Allen[1983a,1985].

We have made the assumption that the planner knows the complete current state of the world. This is clearly a drastic assumption, and the task of relaxing it deserves attention. Our approach has an advantage over the branching-future methods here, because there is no bias towards only reasoning about future uncertainty built into the underlying temporal representation. Intuitively, it should also be possible to reason about past and present uncertainty using our framework. Clearly the assumption-based reasoner can be used to make tentative fact assumptions, but it is not entirely clear when such assumptions should be made. For example, in an event rule, how many of the fact conditions need to be derived and how many can be assumed? The Event Detector handles the extension to an uncertain present elegantly. It can easily be expressed that it is no longer known whether a fact holds by removing that fact from the set of currently observed facts without adding a mutually-exclusive fact. Observed facts would no longer be constants, but instead variables since the same fact could be observed twice. The

arguments of observed fact variables could also be variables. The planner would also have to generate knowledge goals to reason with this extension.

Another useful extension would be the ability to handle conflicting event rules. For example, it might be useful to describe events to varying levels of detail. In the case of a conflict between such expectation rules, the more specific one would take precedence while the assumption of the less specific one could be made *unlikely*. Such conflicts could probably be easily detected by analyzing the new nogood assumption sets added, as we noted in section 6.4.

10.4 Conclusions

Wilensky[1983] describes a model of metapanning in informal terms and uses an adhoc temporal representation to support this reasoning. We feel we have distilled some of the essential components of his theory and shown in more concrete terms how metapanning can be performed with a more sophisticated world model.

McDermott[1982] describes a complex temporal logic based on a branching future in order to reason about interaction with an uncertain future. He addresses the problem of prevention in some detail. His logic is unable to express the difference between actual and possible occurrences, though. We have shown that it is possible to prevent facts without the use of a branching future model. In addition, our method allows straightforward reasoning about the actual world.

DeKleer[1984a,1986] describes a method for assumption-based reasoning. His method involves a small set of basic operations including: removing less

specific derivations, removing derivations based on contradictory assumptions, and combining assumption sets. It is a cleaner and simpler method than justification-based approaches. It also enforces ruthlessly the condition that all assumptions be made explicit.

Instead of using an adhoc representation with various patches to handle problematical cases, we approached the problem from the ground up. We selected a temporal representation which did not require strong world assumptions to be imposed, and chose an assumption-based reasoning approach which allowed maximum flexibility. The cost of this approach is that it requires a fairly large amount of effort to get an adequate formalism. We found that once we had our basic temporal and assumption-based model in place, however, many of the solutions to higher-level problems just fell easily out of the formalism. For example, our Command Executor and Event Detector are relative simple. Replanning is also fairly simple.

In conclusion we propose that the combination of Allen[1983a]'s time-relation reasoning with DeKleer[1984a,1986]'s assumption-based reasoning provides a flexible world model. This world model can be used to advantage in a planner, not only because it allows more flexible descriptions of goals and plans, but also because it allows planning conflicts to be easily detected and effectively described.

REFERENCES

We will use the following abbreviations for conference proceedings :

AAAI : national conference on artificial intelligence (of the American Association for Artificial Intelligence).
IJCAI : International Joint Conference on Artificial Intelligence.
CSCSI : conference of the Canadian Society for the Computational Study of Intelligence.

The following annotations indicate the extent to which each reference influenced the thesis :

** = a major influence.

* = a fairly strong influence.

Allen, J. [1979], "A Plan Based Approach to Speech Act Recognition", TR 121/79, Department of Computer Science, University of Toronto, Toronto.

Allen, J. [1983a], "Maintaining Knowledge about Temporal Intervals", *Communications of the ACM*, vol 26, no 11. **

Allen, J., Koomen, J.A. [1983b], "Planning Using a Temporal World Model", IJCAI. **

Allen, J. [1984], "Towards a General Theory of Action and Time", *Artificial Intelligence*, 23 (2). **

Allen, J. [1985a], "A Model of Naive Temporal Reasoning", in Hobbs[1985]. **

Allen, J. and Hayes, P. J. [1985b], "A Common-Sense Theory of Time", IJCAI. *

Charniak, E. [1978], "On the Use of Framed Knowledge in Language Comprehension", *Artificial Intelligence*, 11 (3).

Charniak, E. [1981], "A Common Representation for Problem-Solving and Language-Comprehension Information", *Artificial Intelligence*, 16 (3). *

- Charniak, E., and McDermott, D. [1985], *Introduction to Artificial Intelligence* (Addison-Wesley). *
- Coelho, H., Cotta, J.C., and Pereira, L.M. [1980], *How to Solve it with Prolog*, Laboratorio Nacional de Engenharia Civil, Ministerio da Habitacao e Obras Publicas, Lisboa.
- Cohen, P. [1979], "On Knowing What to Say, Planning Speech Acts", TR 118/79, Department of Computer Science, University of Toronto, Toronto.
- Dean, T. [1984], "Temporal Reasoning with Metric Constraints", CSCI. **
- DeKleer, J. [1983], "Assumptions and ambiguities in mechanistic mental models" in Gentner [1983].
- DeKleer, J. [1984a], "Choices Without Backtracking", AAAI. **
- DeKleer, J. [1984b], "How Circuits Work", *Artificial Intelligence* 24 (1-3).
- DeKleer, J. and Brown, J.S. [1984c], "A Qualitative Physics Based on Confluences", *Artificial Intelligence* 24 (1-3).
- DeKleer, J. [1986], "An Assumption-Based TMS", to appear in *Artificial Intelligence*. **
- Doyle, J. [1979], "A truth maintenance system", *Artificial Intelligence* 12 (3). *
- Doyle, J. [1983], "Methodological Simplicity in Expert System Construction : The Case of Judgments and Reasoned Assumptions", *AI Magazine*, Summer.
- Ermann, L., Hayes-Roth, F., Lesser, V., and Reddy, D. [1980], "The Hearsay-II Speech Understanding System : Integrating Knowledge to Resolve Uncertainty", *Computing Surveys* Vol 12, No. 2.
- Faletti, J. [1982], "Pandora - A Program for Doing Common Sense Planning in Complex Situations", AAAI. **
- Fikes, R.E. and Nilsson, N.J. [1971], "STRIPS : a new approach to the application of theorem proving to problem solving", *Artificial*

- Intelligence*, 2 (3-4).
- Fikes, R, Hart, P. and Nilsson, N. [1972], "Learning and Executing Generalized Robot Plans", *Artificial Intelligence*, 3 (1-4).
- Forbus, K. D., and Gentner, D. [1983], "Learning Physical Domains : Towards a Theoretical Framework", in *Proceedings Second International Machine Learning Workshop*, June 1983.
- Forbus, K.D. [1984], "Qualitative Process Theory", *Artificial Intelligence*, 24 (1-3).
- Forbus, K. D. [1985], "The Role of Qualitative Dynamics in Naive Physics", in Hobbs[1985].
- Gentner D., and Stevens A. (Eds.) [1983], *Mental Models* (Erlbaum, Hillsdale, NJ).
- Green, C.C. [1969], "Application of Theorem Proving to Problem Solving", *IJCAI* (also in Webber [1981]).
- Haas, A. [1985], "Possible events, actual events, and robots", *Computational Intelligence*, Volume 1, Number 2, May 1985. *
- Hayes, P. J. [1973], "The Frame Problem and Related Problems in Artificial Intelligence", from *Artificial and Human Thinking*, A. Elithorn and D. Jones (eds.), (Jossey-Bass, Inc. and Elsevier Scientific Publishing Company, 1973), reprinted in *Readings in Artificial Intelligence*, B. L. Webber and N. J. Nilsson (eds.), (Tioga, Palo Alto, 1981).
- Hayes, P.J. [1979], "The Naive Physics Manifesto", in *Expert Systems in the Micro-Electronic Age*, edited by D. Michie (Edinburgh University press, May 1979).
- Hayes, P.J. [1985], "The Second Naive Physics Manifesto", in Hobbs[1985].
- Hayes-Roth, B., Hayes-Roth, F. [1979], "A Cognitive Model of Planning", *Cognitive Science*, Vol 3.
- Hobbs, J.R. and Moore, R.C. (Eds) [1985], *Formal Theories of the Commonsense World*, Ablex Publishing Corporation, Norwood, New Jersey.

-
- Kautz, H.A. [1982], "A First-Order Dynamic Logic for Planning". Technical Report CSRG-144, University of Toronto.
- McAllester, D. A. [1978], "A Three-Valued Truth Maintenance System". S.B. Thesis, Department of Electrical Engineering, Cambridge, M.I.T..
- McAllester, D. A. [1980], "The Use of Equality in Deduction and Knowledge Representation", M.I.T., January 1980.
- McCarthy, J. and Hayes, P. [1969], "Some Philosophical Problems from the Standpoint of Artificial Intelligence". *Machine Intelligence 4* (Edinburgh University Press, Edinburgh, 1969).
- McCarthy, J. [1977], "Epistemological Problems of Artificial Intelligence", IJCAI.
- McCarthy, J. [1980], "Circumscription - A Form of Non-Monotonic Reasoning", *Artificial Intelligence*, 13(1-2).
- McDermott, D. V. [1976], "Artificial Intelligence Meets Natural Stupidity", SIGART newsletter No. 57, April 1976.
- McDermott, D. V. [1978], "Planning and Acting", *Cognitive Science*, vol 2. *
- McDermott, D. V. and Doyle, J. [1980], "Non-monotonic logic I", *Artificial Intelligence*, 13 (1-2).
- McDermott, D. V. [1981], "Non-monotonic modal theories", *Journal of the ACM*.
- McDermott, D. V. [1982], "A Temporal Logic for Reasoning About Processes and Plans", *Cognitive Science*, vol 6. **
- McDermott, D. V. [1983a], "Contexts and Data Dependencies, a Synthesis", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol PAMI-5, No 3, May 1983. *
- McDermott, D. V. [1983b], "Generalizing Problem Reduction : A Logical Analysis", IJCAI. *
- McDermott, D.V. and Davis E. [1984], "Planning Routes through Uncertain Territory", *Artificial Intelligence* 22 (2).

- McDermott, D.V. [1985], "Reasoning about Plans", in Hobbs[1985]. *
- Moore, R.C. [1977], "Reasoning About Knowledge and Action", IJCAI. *
- Moore, R.C. [1985], "Semantical Considerations on Nonmonotonic Logic", *Artificial Intelligence* 25 (1). *
- Moore, R.C. [1985], "A Formal Theory of Knowledge and Action", in Hobbs[1985]. *
- Newell, A. [1981], "The Knowledge Level", *AI Magazine*, Vol 2, No. 2..
- Nilsson, N. [1980], *Principles of Artificial Intelligence* (Tiago Publishing Co., Palo Alto, CA). *
- Reiter, R. [1978], "On Closed World Data Bases", from Logic and Data Bases, H. Gallaire and J. Minker (eds.) (Plenum Press, 1978), reprinted in *Readings in Artificial Intelligence*, B. L. Webber and N. J. Nilsson (eds.), (Tioga, Palo Alto, 1981).
- Reiter, R. [1980], "A logic for default reasoning", *Artificial Intelligence*, 13 (1-2).
- Rosenschein, S.A. [1981], "Plan Synthesis : A Logical Approach", IJCAI.
- Sacerdoti, E. D. [1974], "Planning in a Hierarchy of Abstraction Spaces", *Artificial Intelligence*, 5 (2).
- Sacerdoti, E. D. [1975], "The non-linear nature of plans". IJCAI.
- Sacerdoti, E. D. [1977], *A Structure for Plans and Behavior* (North-Holland, New York). *
- Schank, R.C., and Abelson, R.P. [1977], *Scripts, Plans, Goals, and Understanding* (Lawrence Erlbaum, Hillsdale, N.J., 1977).
- Schank, R.C. [1982], *Dynamic Memory : A Theory of Reminding and Learning in Computers and People* (Cambridge University Press, Cambridge, New York).
- Schmidt, C.F. [1985], "Partial Provisional Planning : Some Aspects of Commonsense Planning", in Hobbs[1985].

- Sridharan, N.S., Bresina, J.L. [1982], "Plan Formation in Large, Realistic Domains", CSCI.
- Sussman, G.J. [1973], "A Computational Model of Skill Acquisition". Phd. thesis. Massachusetts Institute of Technology, Cambridge, MA.
- Stefik, M. [1981a], "Planning With Constraints (MOLGEN : Part 1)", *Artificial Intelligence*, 16 (2). *
- Stefik, M. [1981b], "Planning and MetaPlanning (MOLGEN : Part 2)", *Artificial Intelligence*, 16 (2). *
- Tate, A. [1974], "INTERPLAN : A plan generation system that can deal with interactions between goals". Memorandum MIP-R-109, Machine Intelligence Research Unit, University of Edinburgh, Edinburgh.
- Tate, A. [1977], "Generating Project Networks", IJCAI. *
- Tate, A. [1984], "Planning in Expert Systems", DAI Research Paper no. 221, University of Edinburgh, Edinburgh, Scotland.
- Turner, R. [1984], *Logics for Artificial Intelligence* (John Wiley & Sons Limited, New York).
- Vere, S.A. [1983], "Planning in Time : Windows and Durations for Activities and Goals", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol PAMI-5, No 3, May 1983. **
- Vilain, M.B. [1982], "A System for Reasoning About Time", AAAI. **
- Waldinger, R.J. [1977], "Achieving Several Goals Simultaneously". *Machine Intelligence 8* (Ellis Horwood, Chichester).
- Warren, D.H.D. [1974], "WARPLAN: A System for Generating Plans, Department of Computational Logic", Memo No. 76, University of Edinburgh, June 1974.
- Webber, B.L., and Nilsson, N.J. (eds.) [1981], *Readings in Artificial Intelligence*, (Tioga Publishing Company, Palo Alto, CA).
- Wilensky, R. [1983], *Planning and Understanding, A Computational Approach to Human Reasoning* (Addison-Wesley). **

Wilkins,D.E. [1984], "Domain-Independent Planning : Representation and Plan Generation", *Artificial Intelligence*, 22 (3). *

Wilkins,D.E. [1985], "Recovering from Execution Errors in SIPE", *Computational Intelligence*, Vol 1, no 1. **

Appendix A:
Index of Goal and Plan Types

Achieve Goal	109
Achieve-Between Plan	121
Avoid Plan	119
Bind Goal	109
Bind Plan	109
Bind Command	109
Make-Disjoint Goal	120
Make-Disjoint Plan	120
No-op Plan	120
Prevent Plan	111
Resolve Conflict Goal	117
Resolve Conflict Plan	117

END

FIN

2 0 4 8 7