# Intelligent Execution Monitoring and Error Analysis In Planning Involving Processes.

Brian Drabble.

Submitted for the degree of Doctor of Philosophy.

THE UNIVERSITY OF ASTON IN BIRMINGHAM

July 1988

Aston University

Summary

Intelligent Execution Monitoring and Error Analysis in Planning involving Processes.
Brian Drabble.
Thesis submitted for Ph.D. Degree, 1988.

An intelligent agent, operating in an external world which cannot be fully described in its internal world model, must be able to monitor the success of a previously generated plan and to respond to any errors which may have occurred.

The process of error analysis requires the ability to reason in an expert fashion about time and about processes occurring in the world. Reasoning about time is needed to deal with causality. Reasoning about processes is needed since the direct effects of a plan action can be completely specified when the plan is generated, but the indirect effects cannot. For example, the action 'open tap' leads with certainty to 'tap open', whereas whether there will be a fluid flow and how long it might last is more difficult to predict. The majority of existing planning systems cannot handle these kinds of reasoning, thus limiting their usefulness.

This thesis argues that both kinds of reasoning require a complex internal representation of the world. The use of Qualitative Process Theory and an interval-based representation of time are proposed as a representation scheme for such a world model.

The planning system which was constructed has been tested on a set of realistic planning scenarios. It is shown that even simple planning problems, such as making a cup of coffee, require extensive reasoning if they are to be carried out successfully. The final Chapter concludes that the planning system described does allow the correct solution of planning problems involving complex side effects, which planners up to now have been unable to solve.

Keywords : Execution Monitoring, Planning, Qualitative Process Theory, Time.

To Mum and Dad

# Acknowledgements

First and foremost I would like to thank Dr. Peter Coxhead, my supervisor. The fact that he guided me through all the important stages in this project, whilst still allowing me to conduct the research in my own way is greatly appreciated. Also I would like to thank Neil Toye and Dave Stopps for attempting to restore sanity to some of the computing facilities and nearly always succeeding. I should like to thank Hugh Dorans for his guidance and advice in coding various parts of this project. Finally, I acknowledge the financial support of the S.E.R.C. of Great Britain.

# List of Contents

# List of Figures

# Chapter 1: Introduction

As planning systems become increasingly sophisticated and their capabilities become ever more advanced, the trend in planning system research has been towards applying these new systems to increasingly more demanding problems. Today, the applications of planning systems cover a multitude of uses (especially since the advent of non-linear hierarchical planners), including process control applications in power stations and chemical plants, military, production and assembly applications. Planners have also been used in space systems for controlling satellites and deep space probes.

All planning systems however, adhere to a basic requirement of vital importance, namely that the plan generated be reliable. In other words, the plan must continue to function at all times. In some cases this reliability may be essential to avoid halts in production and the subsequent loss in profit. In spacecraft control however, a planning system failure could mean the failure of the entire mission or in some cases the loss of human life. This reliability is not easy to achieve as the planner faces problems posed by differing interactions and constraints on goals within a given plan, and with changes which occur in the world as the plan is executed. In artificial intelligence planning systems research until recently, the main emphasis has been placed on the creation of systems capable of producing correct plans free from interactions; dealing with problems caused by changes in the world has taken second place.

As planning systems are developed to cope with more realistic application domains above that of the traditional 'block stacking' type, the problem of monitoring the plan as it executes in the world becomes of greater importance. In the simple block stacking domain used to test many early planners a few lines of simple facts could be used to describe fully the planner's world. However, when we address problems in more complicated domains a correct description of the world

using this sort of representation would be almost impossible to achieve. Thus, the advances in planning techniques need to be followed by similar developments in the representation of the planner's world and the changes which can occur within it.

In many of the early planning systems the plan was never actually executed in the real world, but was merely simulated in a world described by a few simple facts. The actions of the plan could completely specify the changes brought about by its execution and thus the simulation was plausible. However, in more realistic domains an action's effects cannot be completely specified at plan time as there may well be side effects. For example, if the action is to open a valve, then under a certain set of conditions a liquid may begin to flow as a side effect. To avoid this problem many of even today's planners treat the world as a static one, in that nothing changes without the planner either initiating it or knowing about the change. This means that the resulting plans are created using a world model which is inherently a subset of the real world, with some information missing and other information being incorrect or out-of-date. As a result, the plan cannot be simply run in the real world with any degree of certainty that it will accomplish the task for which it was created. The reason for this stems from the obvious fact that the world is not static, so facts about the world change during the course of the execution of the plan. As a result a plan action may no longer be capable of execution, resulting in an execution error. Thus a planning system operating in an external world which cannot be fully described in its internal world model, must be able to monitor the success of a previously generated plan and to respond to any errors which may have occurred.

Execution monitoring and error analysis is an intuitively worthwhile goal, but necessitates having means to reason about why a plan failed and having a mechanism for creating a solution to the problem which can be integrated into the failed plan without causing any further problems. This avoids the problem of having to re-create the entire plan every time there is a failure in one of its actions. A simple approach

could be to place a conditional statement on each action sufficient to handle most errors. This approach, however, is impractical and would create a bottle neck in any planner. An alternative method is to create an internal world model which is a true reflection of the real world. The planner could then check the required effect of an action against the expected consequence of the action. To be a true reflection of the world, the model must change as the real world changes and the planning system must be able to reason why a certain change has come about. Again this approach is intuitively worthwhile, but adds to the complexity of the knowledge representation needed within the planner.

The two most important questions which must be resolved to create such a representation are those of representing time and change. The execution of an action and its effects both persist over a duration of time. These effects may cause changes in the world which again persist over a duration of time. By being able to represent these durations and allow reasoning about how change comes about, the representation should provide an accurate model of the real world. Time plays an important part in planners as many situations a planner may encounter involve dynamic issues.

1. The world changes and thus the planner's view of the world should be modified.
2. Plans evolve and goals shift to meet new situations.
3. Execution deadlines approach and must be met.
4. Assumptions made at one point during planning are later invalidated by new knowledge or by other plan's effects.

The aims of the present research are four-fold :-

1. To create a planner which is capable of
   - planning which brings about and interacts with continuous events in the world;
   - planning which requires the actions to be synchronized with external events which may not be brought about directly by the plan.

2. To create a time representation schema which can
   - maintain the logical consistency between durations in that two durations asserting contradictory facts are not allowed to overlap;
   - represent change occurring over a duration which may be infinite;
   - represent plan actions and their dependency upon events happening outside of the plan;
   - take information from the user and planner and update the model accordingly and report any changes which may affect any of the plan actions.

3. To create a module capable of reasoning about change and updating the temporal model in accordance with its findings as well as being able to reason about why a certain situation has come about.

4. To create a module capable of reasoning about plan dependencies and to be able to patch and repair plans which have errors as well as put forward plans to deal with unforeseen and un-required changes in the real world.

The structure of this thesis is as follows :-

Chapter 2 examines the approaches taken by other researchers to the problem of planning and execution monitoring. The first part gives a brief introduction to planning and its aims. The next three parts deal with previous work in execution monitoring, time and qualitative reasoning.

Chapter 3 discusses the literature reviewed in the previous chapter and points out the problems which planners still face, setting out more sharply the goals of the present work.

Chapter 4 gives an overview of the system which was developed and implemented together with information on how the modules of the system communicate.

Chapter 5 gives details of the module responsible for creating and maintaining the logical consistency of the temporal representation scheme which makes up the internal world model. The module also reports changes in belief about facts within the model to the modules outlined in Chapters 6 and 7.

Chapter 6 gives details of the module responsible for reasoning about changes in the real world caused by plan actions as well as by natural events. The analysis is then used to synchronize the internal world model with what happens in the real world.

Chapter 7 gives details of the module responsible for reasoning about planning in terms of failure analysis and re-planning. It is also responsible for setting up the requirements necessary for the plan generator to generate its plans.

Chapter 8 gives details of the test examples used to verify the correctness of the system. The test examples outline the plan schemas and world knowledge provided

for the system as well as the solutions put forward and the tactics used to generate these solutions.

Finally, Chapter 9, presents the conclusions, made on the basis of the test examples run on the system implemented, as to the relative merits of the approach. This Chapter also suggests extensions and further work which could be carried out to improve on the basic system.

Appendix A gives a detailed description of the communication system and the format of the messages passed between the modules of the system. Appendix B outlines the major data structures used by the modules described in Chapters 5, 6 and 7.

# Chapter 2: Literature Review

## 2.1 Introduction to Planning

Problem solving is the process of achieving a goal; when this involves developing a sequence of actions in advance, planning is needed: a plan is thus a representation of a course of action. Planning is important because if we act without reasoning about the effects of our actions the results may not be reversible; for example, if I chop down a tree without planning where it will fall, the result may be that my house will be crushed. Planning our actions may avoid wasting time and resources: for example, if I know I am short of milk and I pass a supermarket on the way home, I can save myself another journey by buying the milk then. In the examples above there was a condition which either I did not require to come about ('losing my house' in the first example) or did require to come about ('having milk' in the second example). These conditions are the goals which a sequence of actions is required to bring about. The goals of the plan can be unordered, as in a list of errands, or have implicit orderings, as in the instructions to bake a cake.

The parts of a plan can be broken down into two types:

| | |
|---|---|
| Primitives: | which can be executed immediately, e.g. pick up milk from fridge. |
| Subplans: | which represent subgoals of the overall goal(s) which are vague and require further refinement, e.g. obtain money to buy the milk. |

Thus the process of planning is the replacement of a level of subgoal structure by a more detailed level of subplan until the level of primitives is reached. A plan to repair an engine can be subdivided starting from 'remove engine' at the highest level of representation 'to remove 1st bolt of second mounting' at the primitive level of

representation. Hence the plan has an implicit goal hierarchy within it even though the plan contains primitives which are linearly or partially ordered.

## 2.2 Planning System Approaches

Planning systems have been a field of interest within Artificial Intelligence for many years and the majority of research has concerned the creation of planners capable of carrying out efficiently the goals set to the system. Most of these planning systems can be classified as being either hierarchical or non-hierarchical. Nearly all the problems which planners are faced with have a hierarchy of goals and subgoals, but the meaning of hierarchy in the context of planning refers to the representation of the plan within the planner. A hierarchical planner builds representations of the plan at different levels in which the highest level is a vague overview of the plan and the lowest level contains sufficient level of detail to solve the goal. A non-hierarchical planner has only one representation of the plan. Thus both types of planner create plans which have a goal-subgoal hierarchy, but only hierarchical planners use a hierarchy or representations of the plan to make these explicit. A non-hierarchical planner develops a sequence of problem-solving primitives by reducing goals to simpler ones or by reducing the differences between the initial state and the goal state by the use of means-ends analysis. Examples of the non-hierarchical planners are Strips (Fikes & Nilsson 1971), Sussman's (1973) Hacker and Tate's (1975) Interplan.

By using a single level representation, non-hierarchical planners were incapable of reasoning about the importance of goals and sorting out those details which were critical to a plan's success and those which were not. These planners would develop an entire subplan down to the level of the primitives before realising that another high level goal was impossible to achieve. This means the planner gets embroiled in too many levels of detail or in some cases is incapable of creating an optimal plan as it cannot solve the interactions between goals in the plan. The problem was identified

by Sacerdoti (1973) to be the premature commitment to a level of detail before the critical parts of the plan were sketched out, and is overcome by hierarchical planners. These use depth-first decomposition to form a sketch of the plan which is complete but vague and then refine it until a complete sequence of primitives is defined. With a limited amount of detail at the top level of the plan, a complete sketch of the plan can be achieved without the expenditure of too much computational effort and thus important errors can be picked up earlier. Hierarchical planning can be developed in several ways. Abstrips (Sacerdoti 1973) used the idea that certain subgoals were more important than others and expanded those first at the highest level and initially ignored the other subgoals. The name Abstrips gave to these levels were 'abstraction spaces' and the advantage in using them was the reduction in searching in any one abstraction space. This is because ignoring detail reduces the number of subgoals to be expanded in any one abstraction space. The G.P.S. of Newell and Simon (1963) was a problem solving system based on theorem-proving in logic and only had one abstraction space, which was defined by treating one representation of the problem as more general than others. Levels of detail could be ignored thus extracting a more general representation of an aspect of its problem space. The hierarchical planners Noah (Sacerdoti 1977), Nonlin (Tate 1976) and Molgen (Stefik 1981a 1981b) are different again from the two mentioned above. Noah and Nonlin use operators which are defined in abstraction levels. They plan first with generalised operators to create a skeleton and then refine the latter with more primitive operators. An example from the Nonlin system is building a house in which we have operators like 'build house' which is refined to a set of primitives and generalised operators like 'decorate' and 'install services'. These in turn are defined in terms of primitives like 'lay storm drains', 'build walls' etc. Molgen was used to plan in the domain of Molecular Genetics experiments and extended the ideas further. The levels of abstraction in the plan were extended above the highest level of the domain to include levels of meta-planning i.e. planning about planning. With these levels correctly outlined the

solution of the domain planning problem was made easier, and the abstraction level representation was also applied to the objects the system used so that commitment to a particular laboratory experiment did not limit the planner.

A further sub-division of planners exists in the description of how they deal with conjunctive goals. If the preconditions of a primitive are explicit then they can be reasoned about and constraints found on the interactions of subgoals. The earlier planners Hacker, Interplan and Waldinger's (1975) system planned the goals separately, finding a plan to satisfy the first goal and then fitting a plan to achieve the second goal. This meant the planner would often have to return to the plan to fix it due to a failure in the ordering of constraints. This assumption is called the Linear Assumption:

> "subgoals are independent and thus can be sequentially achieved in an arbitrary order" (Sussman 1973)

The linear assumption does stop the system trying all possible orderings of subgoals to find a non-interacting solution but instead states that if you have no knowledge as to how to order the subgoals try any order and then fix it. This however will lead to sub-optimal solutions and in some cases to no solution at all.

The alternative assumption is non-linear planning which uses a least commitment strategy. Rather than committing itself to an ordering early, the planner makes no ordering between goals until forced to either because the effect of one plan action interferes with another plan action's precondition, or to take advantage of a fortuitous situation. Thus the plan is not an ordered set of actions but is partially ordered. Actions can occur in parallel if they do not interact and are only ordered in the cases outlined above. The idea is used successfully in the Nonlin and Noah systems and was extended to the Molgen system which applies the least commitment strategy to the objects used in the experiments.

This literature review aims to put forward the basic ideas and research which relates to the fields of Plan Execution Failure and Error Analysis. The systems reviewed are mainly those defined as non-linear and hierarchical because up until their introduction, the domains in which the planners were tested offered little chance of failure and and thus the replanning capability required was very small if it existed at all.

## 2.3 Plan Execution Failure

As described in the introduction, the bulk of planning research has been directed towards creating systems capable of creating a sequence of actions to solve a given problem. However there exists a second set of problems in actually executing the plan in the real world so that it brings about the goal the plan was designed for. The problem is that the preconditions of the actions cannot contain every possible condition that may arise in the world as this is impossible. Hence there will arise times when the planner is faced with unexpected situations. These situations may invalidate parts of the plan and bring about what is called 'plan execution failure'. Some types of interaction are inconsequential and can be ignored while others may threaten the viability of a particular plan in a given context. To solve this, the system must first reason about the failure and secondly redefine part of the plan to reachieve the part which was invalidated. Thus plans come into conflict if the projected actions and effects of one plan cause assumption violations in the other or if conditions in the world outside the planner's control bring about situations not expected in the plan. These situations may be as simple as the door the robot wishes to pass through being locked or as complex as reasoning about a vat of chemicals while moving some boxes around. In the second example, the plan to move the boxes around is not at fault but the planner must stop what it is doing to see to the chemicals if the vat enters an undesirable state.

Due to the difficulty planners face in executing plans in the real world the domains chosen to test planning systems have been restricted to abstract ones where there is little notion of a metric, let alone spatial relations, temporal problems or errors. Sussman's (1973) Hacker works mainly in the abstract blocks world but it has also been applied to the problem of generating computer programs. Sacerdoti's (1977) Noah is not restricted to the blocks world, but every task description must be accompanied by the procedural semantics of the domain, and in order to determine these, the user virtually has to know the answer in advance. Stefik's (1981a, 1981b) Molgen works in the domain of planning molecular genetics experiments, but does not even have a strong sense of quantity. Hierarchical planners like NOAH or Tate's Nonlin (1976) allow a hierarchical representation of the planning process and the explicit representation of goals and subgoals within a given problem. They have been fairly successful in producing plans for a limited number of problems in a wide variety of domains.

As the domains chosen to test planners are usually abstract or simulated, the amount of detail required to describe the world is limited as frequently a few lines serve to describe completely the semantics of their worlds. The impoverished semantics of the worlds employed in these planners is not their only problem. Many planners treat the world as a static one, in that nothing changes without the planner either initiating it or knowing about the change. This means that the resulting plans are created using a world model which is inherently a subset of the real world, with some information missing and other information being incorrect or out-of-date. As a result, the plan cannot be simply 'run' in the real world with any degree of certainty that it will accomplish the task for which it was created. The reason for this stems from the obvious fact that the real world is not static, so that facts about the world change during the course of the execution of the plan. As a result a plan action may no longer be capable of execution and an execution error occurs.

The following sections review work done in the area of plan execution failure and point out attempts which may look like error analysis and replanning but which are in fact simplified solutions to the problem of unexpected situations in the execution of a plan. The main problems which arise when trying to solve these problems are how to represent time as the plan executes, how to input unexpected situations and how to reuse as much of the plan as possible when failure is detected.

Probably the two most reviewed planning programs of the last decade are Sacerdoti's (1977) NOAH and Sussman's (1973) Hacker. Neither NOAH or Hacker emphasised either an approach to representing time or execution monitoring, but both were concerned with planning a sequence of actions for accomplishing a task. Both programs required a data structure in which to represent their internal models of the world. The representation needed to include the changing states of the world over the period of time in which the planner proposed to perform the actions. The objective was to create a record structure in which the preconditions and effects of an action could be stored in such a way that subsequent changes to that structure which violated those preconditions or made the effects redundant would be noticed and the appropriate action taken. Taken from this perspective, a plan interaction occurs when an assumption is invalidated, where an assumption is an interval asserting a plan precondition required over the duration of the whole or part of the plan. The representation was required to detect interactions between steps in the developing plan, but another major problem which planners face is how to deal with unexpected situations which cause interactions with the plan, and how to replan to overcome the interaction.

There is very little previous work in the area of plan execution failure since most domain independent planners do not address the problem of replanning. The only three planning systems that do are those of Hayes (1975), Sacerdoti (1977) and Wilkens (1983). Tate's (1976) Nonlin and Vere's (1983) Devisor do not concern themselves with plan execution monitoring. Planx10 (Bresina 1981) covers 'plan

review strategies' as an area to be covered in future research and as such cannot be presently said to be a replanning system. Srinivas (1978) categorises errors which can occur in the execution of robot plans and suggests several approaches to correcting them. Hayes-Roth and Hayes-Roth (1979), working on incremental and opportunistic planning, put forward a formalism which suggests that human planning operates on several levels of data and planning abstraction, with execution taking place on partial plan fragments rather than on a complete plan for the given problem. Their formalism requires the integration of the creation and execution of a plan, with its subsequent modification through the monitoring of the actual execution and also through simulation of the effects of the plan actions. Other planning systems specify a need for execution monitoring and error detection, but do not necessarily indicate whether this should be carried out by the planning agent or by the user. Pandora (Faletti 1982) suggests as a solution the use of meta–goals integrated into the overall plan framework.

Elmer (McCalla *et al* 1982) is a planning system to create plans for a taxi driver in an imaginary town by splicing together parts of plans of journeys made previously. The system contains an executor which compares what is seen this time against what was seen in a previous journey, if the route had been transversed before. It works by simulating execution to add detail to the plan e.g. 'if you see the drug store you have gone too far'. These checks are implemented as demons within Elmer. These secondary plans are supposed to help Elmer avoid making mistakes in the execution of the route. However, if Elmer does get 'lost' then it has a set of techniques to try and work its way back to a point it recognises. However such techniques as retracing its steps may prove a little dangerous in a one way system! This is not replanning but a version of planning in which detail is sorted out at execution time and thus Elmer's plans tend to be high level and vague. The executor produces complex conditional plans (and thus possible parallel interactions) for situations which previous journeys have told it may arise. It does not accept arbitrary input during execution as its entire

input is provided by a simulation program in the form of 'windows' onto the world. These are features which can be seen and noted e.g. a set of traffic lights, a pedestrian crossing, a supermarket, etc. Any discrepancies which are noted are then analysed and if necessary a set of plan patches is used to rectify the problem. The system can modify its actions by the use of these secondary plans but cannot replan by changing the original plan.

Sussman's (1973) Hacker does modify plans but only at the planning stage, as do most hierarchical parallel planners, but has not yet got the ability to deal with unexpected situations. The idea employed by Hacker is to use a library of plan actions to produce incomplete plans which are simulated to detect errors. If any errors are detected then Hacker tries to solve them with a few simple actions such as reordering parallel actions and or goals to avoid clashes. This is similar to the Tome and Critic system employed in Noah to sort out plan interactions. The Tome held a record of the interactions at a particular level in the plan and the Critics were programs capable of detecting a certain type of interaction in the developing plan e.g. a precondition of one action made true by a side effect of another action, or a precondition which would be deleted if the actions were left unordered, etc. This means the planning difficulties are expected to be solved at plan generation time and not at execution time, thus the interactions are known. What happens in true execution monitoring is that correct plans are generated to begin with and then modified by a re-startable planner on the basis of unexpected occurrences. The re-startable planner must be capable of taking the current plan context and then weaving in an extra plan fragment to take into account the unexpected situation. The planner must be capable of using edited plan schemas to avoid creating plans which include steps already executed and thus avoiding any consideration of interaction with them.

Fahlman's Build system (1974) was capable of impressive behaviour when given the task of building a block structure from some blocks on a table. A method of simulation was used and this indicated any problems which were in the plan to build

the required structure from the blocks given. This system accomplished this by the use of multi-processes where each process was spawned by a parent process whose control and data environment were saved. If carried to its extreme there would be many worlds around, only a few of which could exist in reality; thus the system did a lot of unnecessary reasoning.

The run time execution monitoring system was split into two sections:

1. Choice Functions

   Every function which made a major choice (where to put a block) had associated with it a choice function. This kept a list of the choices it chose from so if an error was to occur the replanner would have this information at hand.

2. Error Handlers

   The reasoner had a piece of re-entrant code known as a Gripe handler to solve the execution time errors and problems. For example, if the move function found block B2 in the position it wanted to put block B1, it would pass (hit B1 B2) to the Gripe handler of the most recent choice function. With this information, the Gripe handler was called to sort out the problem. Due to being called as part of the choice function the Gripe handler had access to the failure message and the choice's control and data environment. This gave it information on the conditions the choices were made from, and with access to the data environment it could copy this out and experiment to find a solution to the problem. If the problem arose due to a decision made at a higher level, then control was passed to the Gripe handler at that level. The actions available to the Gripe handler were as follows :-

1. Detach the disaster environment and drop the choice from the plan. If all the choices at a level were exhausted the system retreated to the next level. Thus the system did not have any way of judging how damaging the disaster had been. All the choices at a particular level might be invalid but the Gripe handler tried them one after the other before retreating to the next level. In the case where no Gripe handler could handle the problem the system admitted defeat and asked the user.

2. If the disaster could be overcome the the disaster environment was kept and the Gripe handler tried to splice in actions to overcome the problem. For example the Gripe handler in the above example would splice in actions to remove block B2. The same applies to the refinement and redefinition of goals which could be changed by the Gripe handler. This however, leaves the worlds in varying states of completeness which may or may not be used again. In a particularly complex pattern of construction there may be many levels of Gripe handlers stacked up waiting for one of them to signal success. Fahlman (1974) does not make clear how, when goals are redefined and changed, the different worlds are checked for consistency. It appears not be an easy problem to solve, because with the model full of different worlds which may or may not be completed, there will be a fair amount of inconsistency and indeterminacy to reason with, making checking difficult.

Sacerdoti's (1973) is often cited as a planner which could function in the real world. The Abstrips system, as its name suggests, was an extension of the Strips system. The problem with Strips was that the subgoals of its plans were held in a

stack, elements of which were never checked for interactions between them. This meant the system would achieve a goal and then undo it again. A further problem due to the variable binding in rules was that there were many combinations of bindings from the given world, thus creating many more states in the search space. This was recognised by Sacerdoti as a too early commitment to a planning strategy, i.e. dealing with the detail before the skeleton of the plan was complete. His solution was Abstrips: a version of Strips which planned in a hierarchy of abstraction spaces. Each level was decomposed into subplans and actions of greater detail at the next level; and thus major planning problems were detected earlier. Abstrips did partially solve the problem of dealing with levels of detail but in doing so introduced new problems. Before moving on to the next level of detail, a complete level of detail was expanded. A level of detail was defined as a test on the preconditions of the rule. The more general the preconditions of the rule the sooner it was expanded in the hierarchy and *vice versa*. A precondition which could not be expanded caused the corresponding path to be ignored. This meant a great deal of processing was done in filling out the plans at each level. The plan was not filled out at different levels simultaneously so the system would never find a plan at level n until it had filled out every level to n-1. A secondary problem was mapping between levels of description in the hierarchy, e.g. a more detailed level node would have to be filled out before a node at the higher level would make any sense to the planner.

Abstrips was very reliant on the user giving the initial levels of detail for each precondition in the rule. The user would provide the initial values and the system would then adjust them by checking how easy it was to form a plan to meet them, i.e. the easier it was to fulfil the sooner it was expanded. In a domain with changing actions the preprocessing of the preconditions is an unfeasible solution.

However, even though Abstrips was used to plan and drive an actual robot, it was placed in an environment which was simple and well designed. The problems it faced were moving boxes from one room to another connected by doors. The only

real problem it faced was if the door was locked. The plans which it produced were severely sub-optimal due to the lack of inter-goal communication, which caused problems similar to those in Strips.

Errors which the world introduced were handled by a second planning system Planex (Fikes, Hart and Nilsson 1972) available at execution time. This may seem like a good idea, where at execution time, processing power makes it viable to handle deviations encountered in the modeled world. It does however have a problem in that it turns the execution strategy into a somewhat haphazard affair. This comes about due to the fact the model is not completely adequate, so a cycle of "action, sensing, new action to achieve the same goal" can often develop. This problem is caused by the planner encountering an execution time error and in response restoring the search tree back to the previous level before the node was executed. The node is eliminated from the choice and a search for an alternative started. The backtracking method causes a problem in that the process and the context in which the error was found are no longer around as the search tree has been rebuilt. Thus Abstrips relies heavily on being able to produce good resilient plans at the highest levels.

Planex (Fikes and Nilsson 1971) was used as a monitor to check on the execution of STRIPS plans that were given to Planex in the form of triangle tables. This fixed representation of the plan was always adhered to as Planex was incapable of changing the order of the sequence of actions and thus the system could not replan. Planex uses the triangle tables to determine the latest point in the plan where the execution of a patch plan could begin in the current situation, including both the executed and unexecuted portions of the plan in its calculation. If unexpected situations create a restart point from some point other than the current execution point, then Planex will allow this. However, as Planex does not change the order of execution of actions it may skip actions (which is an advantage) but it may also re-execute actions again. This ability to restart from a different point is not replanning, as usually an unexpected situation does not give rise to a condition from which an

existing plan action may be executed, although with the very high level examples as used in Planex this may well be true. For example an action such as 'pick up block' can be simply repeated when the block is accidentally dropped. If however the plan includes a step to pick up the block from a particular point then the original plan will only contain a restart point if the block was dropped at the same point as it was originally picked up from.

The work of Airenti (1985) is concerned with a robot which is placed in a world where there is limited chance of error, e.g. the robot flicks a light switch and the light does not come on. The robot can handle simple deductions like this, but cannot cope with complex problems due to the poor semantics of the world description.

Tate (1984) reports on the outline of a system for condition monitoring as an extension to NONLIN. He argues that the plan should capture the intent of the goal and thus aid in re–planning. His system uses Task Formalism (TF) to represent precisely the outcome of an plan action which should be monitored. If a lower level failure can be detected and corrected while preserving a stated higher level goal structure, the fault need never be reported back to the higher levels. The implications of any propagated failure are computable and appropriate re–planning can take place.

The systems of Hayes (1975), Sacerdoti (1977) and Wilkens (1985) have addressed the replanning problem. However the approaches used in the first two are considerably simpler and less powerful than the system of Wilkens. NOAH does not allow the arbitrary input of predicates so any form of general replanning is never allowed. It does permit the user the indicate when whole sections have been executed and allows a node that has been previously executed to be planned for again if it fails. This essentially provides NOAH's only replanning capability which makes it weak and only useful in very limited situations.

Hayes's (1975) planning system uses two interconnected data structures to represent the plan actions and the decision nodes at each point during the planning process. This is then used to perform limited replanning; limited in that the system's

only action is to delete the part of the plan at fault. The tree is pruned back to a point in the decision tree at which the faulty assumption takes no part. This then allows the planner to reachieve higher level goals, but they are constrained by the decision graph to be the same higher level goals which were already present in the plan. This is a highly inefficient form of replanning as the pruning will result in parts of the plan being thrown away which could have been re-used. For example if only one of the many effects of a plan action has failed, subplans depending on the unfailing effects do not need to be deleted.

The research of Wilkens (1985), describes the execution monitoring and replanning abilities that have recently been incorporated into his Sipe domain independent planning system (Wilkens 1983). The environment of a mobile robot was used as the domain for the development of the techniques. The major limitations of the research stem from the assumption of correct information about unexpected events. This avoids may difficult problems especially generating the high level predicates which Sipe used in its planning. As the system was originally designed as an interactive planner, what information the system is given and how the analysis of unexpected situations is carried out is not clear. In his 1983 paper Wilkens reported that in his opinion it was best served by asking the user for help, so how much actual replanning ability the system has is unclear. Other areas which are not covered by this system are modelling uncertain or unreliable sensors; how much effort is to be expanded checking facts that may be suspect; and how to decide which situations require closer examination.

The work of Brooks (1981) reports on a system to control a robot manipulator for industrial assembly operations which took into account possible parts placement and tolerances. The program uses a geometric, computer aided design type of data base to infer the effects of actions and the propagation of errors. The calculations are done using a symbolic notation which relies on all the information being present, and the amount of symbolic reasoning is limited. The actual calculations of arm trajectory

and part positioning use a standard mathematical notation with symbolic information added in. Thus the comparisons which are carried out by the system require actual numeric values. The changes which occurred were carried out in a simulated world in which no changes occur with out the intervention of the planner. To make the analysis of possible problems easier the system employs a preprocessor phase to check how actions and their effects may deviate and still give the correct outcome. The preprocessor phase grades the outcome of an action on a scale of one to six. It can accept a plan as workable, perhaps adding further constraints, or reject the plan if it is deemed geometrically unfeasible, independently of the degree to which uncertainties in the physical system can be reduced by sensing operations. The work reported outlines how a plan checker may be integrated into a planning system. At one end of the range, the checker may be implemented in the robot control language to aid in the modelling of uncertainty effects which the language would propagate from one command to another. This would form part of the compiler of the language in that when the uncertainties which the commands propagate become greater than an acceptable limit the programmer will be informed. At the opposite end of the range the system would be implemented within an automatic planning system. It would be of use in the expansion of skeletal plans into more detailed levels as well as a test and generate procedure required to provide sensor information. Although these ideas were reported as being useful by Brooks, some were never implemented and the chances of finding a domain in which the mathematical constraints can all be found appears slim. Hence the domains over which the system could be applied are extremely limited, and the underlying representation scheme cannot be expanded to allow use in domains where the robot is mobile or the world changed without the robot's knowledge.

## 2.4 Review of Time Representation Research

The representation of time is a central issue to any system which attempts to model any real world situation. The problem of representing temporal information and temporal reasoning arises in a wide variety of disciplines: computer science, psychology, philosophy and linguistic studies. In computer science research, the problem of representing time is at the heart of information systems, program verification, artificial intelligence and other areas involving process modelling. In the field of planning systems, sophisticated world models are required to represent the effects of change and persistence. A plan has preconditions which must persist over a time interval for the action to be executable and the effect of executing the action is to bring about a change in the world which must also be represented. For this model to be a true representation of the world, it must be synchronised with information from the real world. Time is therefore important in terms of detecting the causal relationships between facts and for reasoning about changes which will come about e.g. famine will follow drought. Research into time can be broken down into four areas:

1. State space techniques
2. Time and date systems
3. Before and after chains
4. Temporal Logics

All of these have involved the creation of representations capable of capturing the notion of the time point 'now' as it moves into the future, and the temporal relationships which exist between events. The representations themselves are based on two distinct types: point-based systems, and interval-based systems where the interval is delimited by a begin and end point. The interval-based systems seem to have the advantage as most events have a distinct begin and end points. However,

there are problems in using any of the representations which have been put forward so far.

The problems which need to be addressed when creating a time representation scheme are as follows :-

1. The representation must allow the recording of both the occurrence in time and the relative ordering of events.

2. The period over which the events in the domain occur may require a wide variety of values, e.g. in electronics a program analysing gate switching needs a representation which can reason in nanoseconds, whereas an office planning system needs to reason in hours and days.

3. The representation should allow facts to be believed and remain so until contradictory evidence can be found, i.e. the idea of the persistence of belief.

4. The representation should allow the time point 'now' to be easily updated as it moves into the future and must allow the relationship between any time interval and 'now' to be easily calculated.

The research which has been undertaken in the area of time representation systems can be roughly divided into two areas: those which deal with the creation of logics capable of representing time in a domain independent sense, and those representations which were created to solve time problems in a particular planning domain.

The earliest approaches to the representation of time are the state space representations of Fikes and Nilsson (1971) in Strips, Sacerdoti (1977) and Green

(1969). They were concerned with problem solving and as such needed means of representing change in the world as the plan to solve the problem was executed. This meant they needed to represent the 'state' of the world beyond the current one. These systems had only a crude sense of time in that the states are in chronological order, and represent the world as instantaneous points as the states are unable to represent continuous events and processes. The world is thus a series of states where changes between the states due to the execution of a problem-solving action are represented by the addition and/or deletion of facts in the subsequent state. This however, has major faults in that the size of the knowledge base becomes excessive as state n is copied to state n+1, so a massive amount of duplication occurs. To avoid this problem, the representation could be altered so that it only copies forward the facts which have been changed and thus cuts down the amount of information carried forward. The problem that then arises is that the current world is a summation of all previous states and a great deal of back-tracking is required to find a fact which goes through the plan unchanged. These models are useful in limited domains, but fail to fulfil many of the criteria put forward in Section 3.4, although they do provide a strong sense of persistence as a fact has to be explicitly changed to be no longer believed.

The problem with the above systems was their inability to index facts and events correctly and it was this inability which was corrected in time base systems. The work of Bruce (1972), Hendrix (1973) and Kahn and Gorry (1977) attempted to create an indexing system based on time dates. The facts have associated with them a time date which can be used to sort them into time order, if the indexing system is based on the line of integer numbers. The system can be extended to use calendar dates, providing every event has an associated date with which to index it. This is the problem of using such a representation, as many events have unknown start times and durations. The representation can use ranges to capture the begin points of an interval but this does not solve all the problems. This type of representation cannot

handle disjunction between two facts as the date system puts an implicit ordering on them, either one is before the other, or it is during or after. This problem becomes more severe when we deal with intervals as primitives. To solve the problem we require ranges for both ends of the interval plus a range for the maximum and minimum duration of the interval.

The main problem pointed out with the time base systems was their inability to deal with imprecision and it was this which lead to the development of before/after chains. This representation uses a qualitative vocabulary to capture the relationships between intervals, e.g. 'before', 'after', 'during', 'meets', etc. This was adopted in the systems of Bruce (1972) and Kahn and Gorry (1977) which were successful in the domains chosen. Bruce suggests a formal model of temporal references in natural language which was extended to be a common framework for the analysis of tense, time relations and other references to time in language, and to represent the structure underlying the temporal references. This model begins with a partially ordered set called "time" and successively builds concepts such as "time segment", "duration", "time segment relation", "tense", "reference time" and "tense marker". In his framework, he defined seven relations between time intervals and segments: 'before', 'after', 'during', 'overlaps', 'overlapped', 'meets' and 'same time'. His work could have been extended to planning, if the temporal logic in the representation had been sophisticated enough. It was this work which formed the basis of the time representation schemes of Allen and Vilain.

However, these systems suffer from two major problems: namely as the number of events grows the search time becomes too great; and if, as in some of the systems, the transitive closure of interval relationships is calculated and stored, the processing required becomes too great. The problem can be overcome by using reference intervals as grouping points for intervals. For example, we could break the day up into three reference intervals: breakfast, lunch and dinner. By asserting all events relative to one of these reference events, then we can group these events together.

This cuts down the amount of information processed by the transitive closure operation as knowing the relationship between two reference events means that the relationship between the intervals grouped with them is also known. This type of representation has been fairly successful in domains which have a well-defined layered structure with easily definable reference events. Such a situation is found in medical information systems and the work of Mittal and Chandrasakeran (1984) uses this representation. The reference intervals used in their work were e.g. date of admission, date of operation, date of release, etc. Below these were further reference intervals and grouped at the lowest levels the 'primitive' events.

Plans themselves are a hierarchical structure, but the preconditions and effects are not assigned to a particular reference interval as no form of parallel action, continuous processes or persistence can be shown by these systems. The fact that two events are disjoint cannot be shown explicitly by this representation unless disjunction is represented. The basic ideas of reference intervals was improved by Allen (1981, 1983, 1984), Allen and Kooman (1983), Vilain (1980) and Cheeseman (1984).

Cheeseman (1984) describes a representation scheme for time and its associated reference rules. The timing information is again stored with the predicate, but instead of a situation value, the begin and end points which define the interval are inserted. The representation uses two schemas: one which indicates that a predicate holds over an interval, and one which indicates that a predicate does not hold e.g.

1. holds (valve_1 open t1 t2)
2. notholds (valve_1 open t1 t2)

This representation is clear and simple but it is difficult to use in reasoning about plans and actions since time relationships represented in terms of the end points of intervals are difficult to handle.

Allen (1984) suggested a formalism for reasoning about actions that is based on temporal logic. This formalism is used to characterise different types of events, processes, actions and properties that can be described in English sentences. In his formalism, the world description contains both static and dynamic aspects of the real world. The static aspects are recorded as the properties which hold over a time interval. The dynamic aspects are recorded as occurrences which are further subdivided into two classes: processes and events. Processes refer to occurrences not involving culmination and anticipated results, and events describe occurrences that involve products or outcomes. Actions are then defined in terms of occurrences that are caused by agents. Actions themselves can be further divided into performance-causing events and activities-causing events which obviously forces the user of the system to put a great deal of thought into the definition of the actions. However, using a representation which only has two aspects sometimes causes problems of definition. For example 'I sat at my desk for two hours' can be viewed both as static and dynamic. Another example is 'the robot hand holds the block' which may be viewed as an occurrence or not since after the robot picks up the block and before it drops the block, it does not take any plan action. To solve the problem a status predicate was introduced into the world description to define the status of an object at a point in time. In the first example, I am in 'sitting' status and in the second example the block is in 'holding' status and the hand is in 'busy' status. This seems to be an artificial solution to the problem which again requires the user to provide the statuses which form a set of high level axioms of world description.

The interval-based representations may seem to reason like humans, but the system does have its draw-backs. The representation requires the user the input the reference intervals as none of the systems was capable of organising itself around a key event. However the main problem arises when the system attempts to add new intervals beyond the current future point, in that the system has to add a new level at the highest level in the system and thus recalculate every relationship in the

knowledge base again. The movement of the time point 'now' in such a reference based representation is by having a pointer variable pointing at a particular interval in the knowledge base. This means the relationship between 'now' and any interval can be calculated fairly easily but the granularity of the change will depend on the facts asserted in the knowledge base and what relationships are known. This will work in domains when the amount of change is minimal because Allen (1983) and Allen and Kooman (1983) makes the key assumption that, while the present is continually changing, most of the world description remains the same. Allen admits in his paper that the system is capable of carrying out simple problems and not complex temporal ones. The reference interval before/after chain systems reported in the literature gave performance figures which seem quite impressive; however, none of the systems employed truth maintenance in that once a fact was asserted it was never removed. If intervals are removed from reference interval systems, and especially higher level references, the amount of recalculation becomes prohibitive.

Truth maintenance involves keeping track of the assumptions under which certain facts are believed, these assumptions forming the justification of the fact. This type of reasoning is called non-monotonic reasoning and differs widely from monotonic reasoning in which the addition of new axioms to a set of axioms can never decrease the set of theorems. It can cause the new axioms to give rise to new theorems, so that the set of theorems grows monotonically with the set of axioms. In non-monotonic reasoning, the set of theorems may lose members as well as gain members when new axioms are added, i.e. beliefs may change from true to false and vice versa.

Allen (1981) suggested an interval-based temporal knowledge representation scheme in which nine primitive relations between intervals were defined, and gave a network representation for relations and a 4 x 4 transitivity table for deducing new knowledge from given relations. Both Allen (1981) and Vilain (1980) went on to define a thirteen primitive interval relation logic. Vilain used a relational vector to represent all the possible relations existing between two intervals. Allen gave a 12 x

12 transitivity table and suggested an approach to maintaining consistency of temporal knowledge in the database of temporal relations when relations were added to it. Allen's transitivity table is not simplified but keeps all the possibilities.

Allen (1983) later suggested a model based on the Strips formalism and temporal logic with temporally defined preconditions, add and delete lists. He tried to apply his temporal logic to ordering actions by using a simple blocks stacking problem. This was the only work done but it was a good attempt. His work did not give a systematic approach to ordering actions automatically by time reasoning and did not address planning tasks with highly interacting goals. Since his paper discussed planning with time reasoning as part of understanding natural language discourse the impact of his representation in the general planning context is unclear.

Allen and Hayes (1986) put forward a representation scheme in which the thirteen primitives defined by Allen could be rewritten in the form of 'meets' relations between sub-intervals defined over the interval of one of the thirteen primitive relations. This does solve the point problem, but how a system can keep track of the number of sub-intervals in a complex planning domain is unknown. The following example illustrates how using this system you can define the relation 'meets' between two intervals. If a light is switched on, then interval **I** where 'light-on' is false, meets interval **J** where 'light-on' is true. This may be regarded as the definition of the relationship 'meets'. From this simple relationship, it is possible to define other relationships in terms of the 'meets' relation. For example '**I** overlaps **J**' can be defined as in Figure 2.1. Sub-intervals **K,L,M,N,O** are constructed based on the end points of **I** and **J**. We can now write the relationship '**K** meets **I**' as '**K** m **I**' and define the whole relationship as follows:

**KmI & KmL & LmJ & LmM & ImN & MmN & JmO & NmO**

Allen and Hayes then define a 'moment' as a non-decomposable interval in which something can happen e.g. a flash of light. A point in this representation is then a meeting place of intervals. Thus every interval has exactly two end points and there are no points between the beginning and end of a moment.

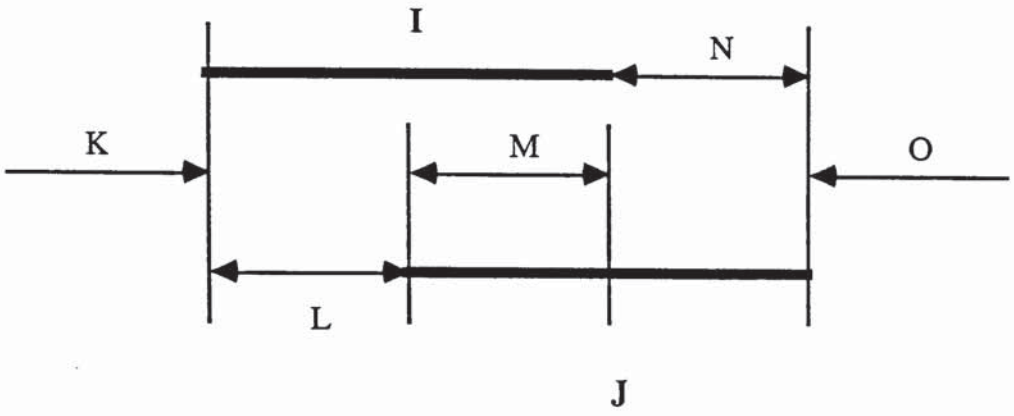



Figure 2.1

The final section on the types of time representation schemes deals with formal logics and models. As described earlier, time representation is required in many areas of research and the book by Rescher and Urqhart (1971) covers the work in psychology and philosophy. One of the more notable models of temporal representation in artificial intelligence is situation calculus (Hayes 1973). In the GPS and Strips family and many other planning systems, time is not especially taken into account. In GPS whether a logic formula holds or not does not depend upon time and the applications of rules are not related to time. In the Strips systems: Strips (Fikes and Nilsson 1971) and Abstrips (Sacerdoti 1973), a world state description is just a snap shot of the real world, and actions and state translations caused by actions are instantaneous. Taking time out of the representation scheme causes problems in consistency, if a fact (e.g. 'valve_1 open') is first true over a time period, then becomes untrue. Obviously 'open valve_1' is not an appropriate description for valve_1 as it holds in the first time period but not in the second. It was to solve this

problem that Hayes (1973) put forward his situational calculus. In this representation every predicate has an extra field to indicate the 'situation' in which the fact is being asserted. In the example above the inconsistency can be removed by asserting the valve is open in situation S1 and not open in situation S2. In situation calculus the world is described as a series of situations each describing the world at an instant and an action is a function to change from one situation to another. It uses the precondition, add and delete list representation of the Strips family of systems to effect change in the world. Although situation calculus removes some logical inconsistencies, it still treats the world as instantaneous and thus a continuous action or event cannot be represented. This theory is only valid in domains where only one event can occur at a time as the transforms from one state to another are instantaneous thus they cannot be reasoned about or decomposed. Situation calculus has the reverse notion of persistence in that a fact true at one instant needs to be explicitly reproven to be true in succeeding events. Thus using situation calculus to represent the changing states of the world requires writing formal and explicit axioms that state what things change and what things remain the same. The number of these so-called frame axioms rises as a product of the number of predicates and actions, which means that adding a new action can require adding a large number of new frame axioms to deal with its effects on the world. There have been several attempts to rectify the problem e.g. by Fikes and Nilsson (1971) and Minsky (1974), but none of them have seemed particularly fruitful. This is because the faults lie with the assumptions in the representation that facts are instantaneous and do not occur together, hence no matter how much research effort is expanded the problem can only be overcome by fiddling with domain dependent solutions.

The work of McDermott (1982) put forward a representation and associated inference mechanism to deal with reasoning about processes and plans. The representation introduced a branching time line which branched at points of uncertainty as to how a situation would turn out, or at a situation which contained

unknown information. The facts in a particular state were called a Chronstate and a Chronset was defined as the number of states which fit a given description. This logic does have problems in that facts have to be constantly reproved to give an idea of persistence; it was unable to reason backwards in time about a given situation; and the number of futures which the system keeps track of is large, as some of the states created bear no relation to reality. The main flaw as far as planning was concerned was its inability to represent 'now' which would greatly handicap it when reasoning about the outcomes of plan actions. The paper presented a sketch of an implementation which has not yet been reported in the literature, so its impact on planning is unclear.

During the discussion there have been references to point-based representations schemes and interval-based representations. In every day usage we rarely make the distinction between points and intervals, but in time representation the differences must be distinguished and maintained. An interval can be defined as a time period delimited by a begin and an end point. This means that by turning up the granularity we can further decompose an interval. If we associate a fact with a time point, then we can treat the fact as being instantaneous whereas in reality it will not be e.g. a knock on the door may appear to be at a time point, but by looking at it further we can break it down into the actions of knocking on the door, and possibly still further into the inferences needed to carry out the actions. Thus the formal notion of a time point which would not be decomposable is not useful. There are examples which point out the problems with consistency in point-based systems, e.g. the problem with the status of a light bulb. If we allow zero-width time points and we consider a light bulb being turned on we require an interval where the light is on and an interval where the light is off. The problem in the representation is whether the intervals are open or closed. If they are open, then there is a time point at which the bulb is on *and* off, and if they are closed, time is not dense and a hole exists in time where neither is true. The second case provides greater semantic problems than the first and could be

overcome by using a representation in which an interval is closed at its lower end and open at its upper end. This seems an artificial solution which reinforces the idea that points on a real line is not the way we see time.

However, we need to use time points in interval-based systems, even though it is accepted that points are not atomic, and they must be properly handled, e.g. in the sentence "I left my flat at 9.00 a.m", "9.00 a.m." is a time point. Although time points are absolute moments in the standard calender they do also have relative meaning, e.g. 9.00 a.m. was the beginning of the interval representing my journey to the office. Therefore time points naturally define the boundaries of intervals at which change takes place and in some systems a time point is merely a zero-width interval. This allows a logic to be defined which defines relations between intervals and time points.

## 2.5 Planning Systems involving time reasoning

As the techniques for representing time have become more advanced, they have from time to time been implemented in successively more complex planning systems. The following section describes some of the planners which have been developed and the uses they have been put to.

The earliest temporal planner was Wesson's (1977) Autopilot which operated in the world of air traffic control. It is hard to see how his techniques could be applied in the real world as the planner relied on simulated aircraft which moved along fixed routes, always obeyed instructions and never interfered in each other's plans. Any interactions which were detected, e.g. an aircraft too close, meant the plan was 'repaired', by inserting simple patches e.g. turn left, turn right, climb, etc.

An extension to this system was put forward by Masiu *et al.* (1983) which attempted to deal with the air traffic control problem in real time, but the work reported was still in its early stages. The project was later dropped in December of

1984 with the reason given that the environment was not knowledge-intensive enough.

Vere's (1983) Devisor system was a planning/scheduling system specifically designed for the Voyager space program, but its main control structures were domain-independent. It was a hierarchical planner and the plans it produced resembled PERT charts. It was designed to achieve a set of goals, where each goal had an associated time window, i.e. the duration of time over which the goal must hold. Devisor generates plans using backward chaining techniques, but has several problems. It is unable to represent actions which occur simultaneously, and a great deal of work is done in continually updating the goals, as new constraints move them around in time and constrict the size of the associated window. Plans can be built around external facts e.g. the time the bank opens, but these facts have to have known time durations as the planner is incapable of handling goals as well as facts which have an infinite window (i.e. when the finish time is unknown).

Of the temporal planning systems put forward the most advanced one reported so far is the Forbin planner of Dean (1985). This is a job shop scheduling system which attempts to create plans to achieve production and assembly schedules. Forbin takes schemas of objects and plan actions which have temporal constraints and relations between their fields. From this the planner creates plans by using a temporal knowledge base which Dean refers to as a Time Network Management System (TNMS) (Dean 1983, 1984). As the planner makes projections of future actions and events they are checked by the TNMS and any interactions which are found are reported to the planner. As such his representation tries to deal with the problems which may arise at planning time when for example the planner decides to use Lathe17 when another plan requires Lathe17 to be idle. The Forbin planner has two major limitations: the first is that it does not backtrack and thus may occasionally fail. A solution put forward was to use a plan debugger to solve what Dean described as impossible situations and to keep the number of fixes small in order not to degrade

the performance of the system. Thus these fixes become a way of adding power to the shortcomings of the representation used in the system. The second problem Forbin suffers from is that it does not have the ability to re-use plans and thus re-plans from scratch each time it is confronted with the same problem.

The Forbin system was put forward as a general purpose planning system for use in domains in which hierarchical planning techniques could be employed. However Dean made various comments about the type of domain in which his system is appropriate, namely, the effects of the planning tasks must be highly predictable, and the duration and position information required in the action schemas must also be estimated fairly precisely. These criteria place fairly strong limitations on the types of domain in which the system can be used.

The representation employed by the TMS (Dean and McDermott 1987) uses an update mechanism to position a point in time by the constraints placed on other tokens in the time map. For example the TMS finds all the paths which exist between two tokens A and B, by finding all the tokens whose constraints place them between A and B. By exploring the various paths the TMS can find an upper and lower bound on the temporal distance between and A and B. This differs from the earlier approach (Dean 1983) in which the TMS held a list which contained the relationship between any two points as a derived_before relation. This list created the transitive closure of all the points so that a derived_before relation existed for every pair of points in the knowledge base. The new idea allows other tokens to define the position of a point and to give metric information on the time difference between any two points. The problem with such a system is the amount of processing required to create the path values within the knowledge base. If the system is employed between every pair of points in the knowledge base, then as the size of the knowledge base expands the update mechanism will soon bog down in processing these path values. To calculate the number of paths between the earliest and the latest point in the knowledge base will require a great deal of processing as the number of paths will be

enormous. The paper of Dean (1985) reports on the latest developments to this representation and suggest methods for partitioning large time maps and guiding search. The obvious way to avoid this recalculation is to cache the results of the previous calculations. This however requires the system to maintain checks as the tokens which move in time will cause a recalculation of the temporal distance. In a volatile domain the advantages of caching would become limited. The application which Dean and McDermott (1987) report as an example domain is that of a publishing house. They suggest breaking down the partitioning into a calender hierarchy, but this requires that there is a date to index every token in the knowledge base which, as it is defined as a calender, forms a natural hierarchy. However the system organises such an index, the problem of representing 'A is disjoint from B' is still a problem. The second problem which this representation suffers from is the creation of the data dependency contradictions as creating them between every contradictory pair of tokens is prohibitive. Dean again refers to the publishing domain to make the assertion that information is unlikely to change very much and as such we can make selective data dependencies according to the application. How this is to be carried out is not made clear, but in any complicated domains these 'dependency rules' would become quite excessive. If the user were to supply them, then the domain description required would require the user to almost know the answer in advance.

In their current implementation of the TMS there are several problems e.g. if two tokens of the same type overlap. The example given by Dean and McDermott (1987) is of a house where the upstairs and downstairs lights are on for parts of the day and actually overlap at one point. From these facts the TMS should have been able to deduce that the lights were on in the house all day but was unable to do so. Dean and McDermott report that a solution is known for this problem but has yet to be implemented. The TMS does have a method of default assumption e.g. if during **PT1 - PT2** there is no evidence to believe **~P** then create a default assumption that

**P** is true over the period **PT1 -PT2.** However this form of reasoning may cause problems within the TMS. As Dean and McDermott describe, there needs to be a way of keeping track of all the tokens of a particular type and there must be a method for concentrating on a restricted subset of all token types. The assumption defaults are referred to by Dean and McDermott (1987) as anti-projections and they claim they can be handled with the same ideas as for setting up data dependency checks, but again this would require a great deal of domain dependent knowledge. The problem of detecting contradictions is further complicated by binding variables in partial plans at run time. This may occur in planning systems if the state of the world is not known exactly at plan time, e.g. the robot may require somewhere to place an article but leaves the exact place to put the article blank. Later the planner may find a table on which it may place the article. This is easy to handle as the article moves from the robot to the table therefore the contradiction in facts asserting the position of the article is easy to detect. If however the robot has a choice of working on machine A or machine B in room 1 or room 2 respectively and the planner has a plan in which the robot is in room 3, then a contradiction occurs which is more difficult to detect as it must realise that the choice of machine may also cause a problem in the robot's position.

The Nonlin planner (Tate 1976) does have the ability to reason with actions which take place over a duration of time. The duration of the action is fixed, but the planner associates with each action a window of time during which it should be executed. However, the system does not have the ability to protect a goal over a certain time interval or to make a goal true by a certain time point. As such the system cannot be considered to be a true temporal planner as in the case of Devisor or Forbin.

## 2.6 Qualitative Reasoning

The problems a planning system faces in the execution of a plan are how to interpret the information arriving back from the real world against the effects a plan

action should have brought about and how to recognise if the action which was executed has failed. The plan generator is responsible for generating bug-free plans, but the planned effects of the actions are not the only effects they will have on the world when the plan is executed. For example, the action 'open valve' will have the action effect 'valve open', but when executed in the real world may also have the effect of allowing a fluid flow along a pipe. It is obvious that we cannot put every effect into an action schema definition, as it is also obvious that we cannot reason for failure with a large conditional statement. The problem could be partially solved if the execution monitor could create a model of the world which it expects and then synchronise this model with the information received from the real world. The model must be able to simulate the world in sufficient detail to pick out discrepancies but not be so detailed as to make the number of possible futures too great to handle. Representing how things change is a central problem as the changes brought about by the planner need to be reasoned about as well as the changes which occur naturally in the domain. For any particular domain we need to identify the effects that act between classes of objects in the domain and the events which result from these effects. These causes of change can be associated with processes which act over time. Using these processes to describe what is happening in a situation enables the problem solver to predict how the situation will change and evolve over time. Qualitative dynamics as put forward by Forbus (1981, 1983, 1986), offers one way of providing the necessary representation. It has the ability to reason with incomplete information and the ability to represent change symbolically. Some conclusions can be drawn with very little information. For example, if we have a liquid in a container and we place a flame beneath the container we can deduce that the liquid may boil. Also if we seal the container it may explode if the liquid boils. This analysis does not say that the container will explode but the possibility is there among the alternatives put forward as possible outcomes. The alternatives can be used to drive the collection of information (DeKleer and Brown 1985) and decide on the activity of the planner,

i.e. if the container may explode, carrying out a plan (which takes half an hour) out of sight of the container is not a good idea. The classic problem which arises when analysing change is the frame problem identified by Hayes (1973), and it was to try and solve this problem that Hayes devised the representation known as 'histories' (Hayes 1979). These solve the frame problem by only allowing objects to interact when their histories interact. For example, if I am wiring a circuit and I am interested in how the parts interact, connect, etc., then I should not include any consideration of the gas heater in the room. This is because the circuit is isolated spatially from it and if it is summer temporally isolated. Thus in this type of representation, temporal and spatial reasoning carries out nearly all the interaction detection. This is not simple, as Forbus admits, but creating a domain-independent representation using qualitative reasoning seems more fruitful than writing clever frame axioms for each new domain, where the frame axioms define constancies and changes in objects from one state of a problem to the next. To accompany the representation, a reasoning mechanism is required first to create the histories and then to maintain a check on their independence. These divisions are only semi-permanent as e.g. I will have a few problems with my circuit if the heater explodes and thus the histories of the circuit and the gas heater interact. This is inherently a non-monotonic problem (Mcdermott and Doyle 1979) which requires the histories to be collapsed and re-analysed. The second problem is that if two objects come together spatially their histories may or may not interact. For example if two crates are pushed together, they will block each other, whereas if a steel ball is dropped through a flame there will be no effect on the movement of steel ball even though the histories of the ball and the flame interact spatially. Solving this problem, as Forbus states, is difficult:

> "in general it requires knowing what kind of things can happen and how they can effect each other - in other words a theory of processes".

An example of this is in Figure 2.2, which describes the effect on the temperature of the steel ball as it passes through the flame. The scenario can be described in four histories, two events and eight episodes. The rise in temperature of the ball causes a heatflow to occur, which in turn causes the temperature of the ball to increase. The ball interacts spatially with the flame hence the three episodes associated with the position of the ball. The process 'movement' describes the ball's path through the flame. The events 'event1' and 'event2' which delimit the episodes are the ball entering and leaving the flame and are said to be instantaneous.

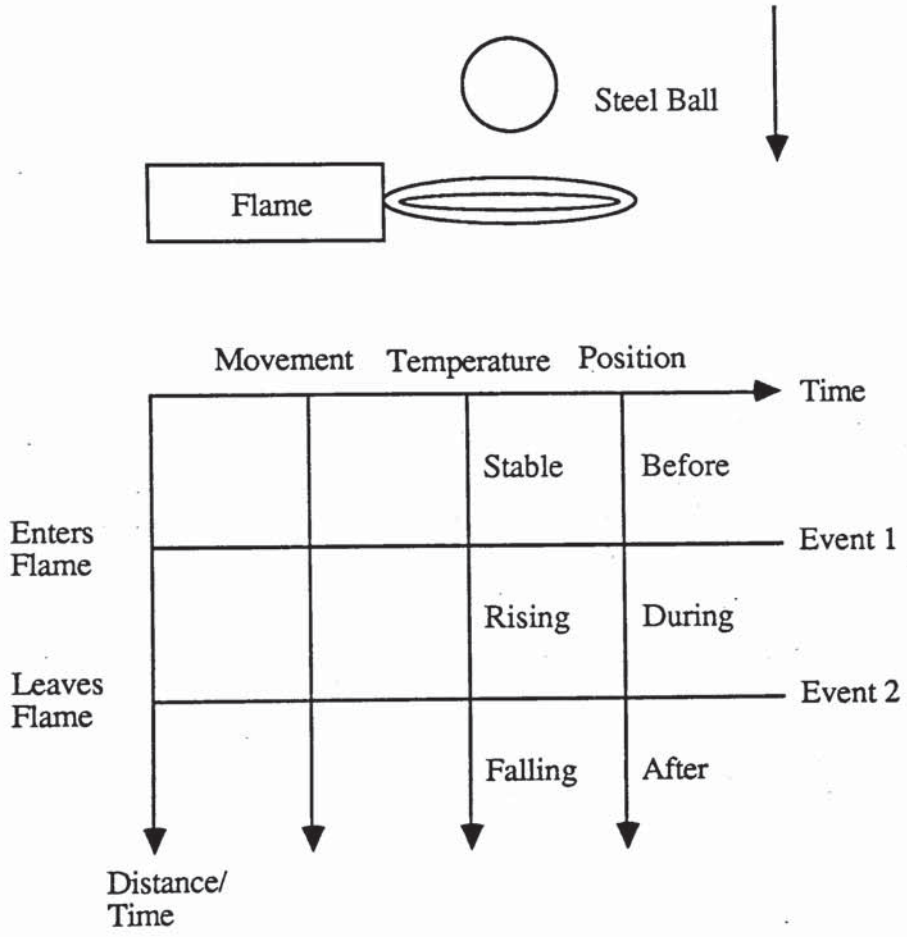



Figure 2.2.

# Chapter 3: Restatement of Problem

The review of the literature in Chapter 2 outlined the research carried out in Execution Monitoring, Time and Qualitative Reasoning. A great deal of effort has been expanded in creating planners capable of generating correct and efficient plans. These have been successful in producing plans for a limited number of problems in a wide variety of domains. However although progress has been made, none of the current programs appears capable of producing plans which are able to work in the real world. The major problems identified in Chapter 2 relate to handling time and reasoning about the consequences of actions executed in the world.

To create plans with a greater chance of executing in the real world, we need to make changes to existing planners. We could try to make the world used by the plan generator more detailed. However, this will cause the plan generation phase to become slower, and it will still not deal with the dynamic aspects of the world. Most of the reasoning which we do, and especially common-sense reasoning, involves time in one way and another. Even facts which are true in every day situations change over time, and from these facts we construct inferences which may be true at a point, or over a bounded or unbounded interval. The static world descriptions used by many planners rule out the expression of relationships such as before, after, during, etc., and cannot express the concept of actions happening in parallel and the idea of two facts being exclusive. For a planner to reason about change, it must be able to reason about time. If time is not taken into consideration then every change or occurrence must be viewed as instantaneous, e.g. actions have no duration and effects like 'door open' and 'door closed' change instantly. Time is thus very important in understanding change.

Of the systems which are discussed in Chapter 2 (those of Allen, McDermott, Vilian, Cheeseman, etc.), the one chosen for this research is that of Dean, in his planning system Forbin. Forbin has a temporal data base of tokens which allows the

representation of facts and events occurring over time. These tokens can be added to, amended or deleted relatively easily, which would be required by a planning system which was trying to model a dynamic and changing world.

Adding time to a problem means that actions can occur over time as opposed to being instantaneous. This again introduces problems in describing both the preconditions of an event and its effects. A precondition can be required to start an action or it may be required for the whole duration over which the action is executing. For example to pick up a block we have to be holding it and it has to be on the table. The precondition, that we are holding the block, is required during the whole of the action, whereas 'the block is on the table' is required only at the start of the action. The fact that the block is no longer on the table at the moment we begin to pick it up is un-important to the execution of the action. This requires the planning system to maintain a check on the temporal relationships of the preconditions to an action i.e. the preconditions begin before the action begins and the preconditions end after the action has finished executing. This can then be used to monitor for failure before the action is executed, i.e. the begin or end point falls within the interval of the action; or for a failure while the action is being executed. In the case of a precondition which is required only to start an action, the effects of the action will be still seen even though the precondition failed. In order for this to be carried out successfully the monitoring must be done in a non-monotonic fashion. This will allow the planner to reason about all the changes a failure will cause at a given point in time.

The knowledge representation scheme for reasoning about change which involves actions occurring over time will have to re-align its reasoning as time passes. The reasoner will only be able to analyse the effects of its actions by simulation, in an internal world model, if the internal world model is a true reflection of the real world. This means part of the planner's function should be to synchronise its internal world model with the real world. The synchronisation is required because the future is not an ordered time line of facts as is the past, since the future branches at points where

information is missing or an outcome is uncertain. The problems of a branching future time line were discussed in Chapter 2. The main problem was that in any realistic problem the number of futures tended to be too excessive. One way of rectifying some of the problems involved in creating an internal world model which accurately reflects the external world would be to use 'qualitative reasoning'. Current research in qualitative reasoning allows the behaviour of various physical systems to be predicted by using qualitative descriptions of the world. However, the physical systems must be considered in isolation as current systems do not allow intervention by an outside agent. For example, a state-of-the-art qualitative reasoning system can predict the possible behaviours of a boiler, but cannot intervene if the boiler is about to explode. Just as qualitative reasoning systems need to understand planning, so planning systems need to understand processes, since plans very often involve them. As a simple example, planning to make a cup of coffee requires some understanding of the process 'boiling' if only to know when to begin pouring the water into the cup. By using qualitative representation and reasoning techniques the number of futures can be cut down because the amount of numerical information is reduced and replaced by qualitative statements, e.g. 'A > B', 'A is increasing', etc., rather than having specific numerical values, e.g. 'A is increasing at 3 units per second'. The work on Process Theory put forward by Forbus (1983) will be used as the basis of the qualitative reasoning system presented here. It will be altered to allow it to be used in real time, by using a frame based process schema approach and reasoning; and the time representation within it improved, by allowing intervals associated with a given fact to have an unbounded end point and by having ranges for the begin and end points. The resulting representation will be able to deal with planning occurring over time and with continuous events and processes. Using qualitative reasoning does cut down the number of futures but the scheme still has to maintain its own consistency. This involves two basic processes, firstly synchronising the real outcome of an initial situation with the possible futures reasoned as plausible, and secondly, maintaining the knowledge base as a consistent set of facts. The idea of

having a synchronisation process was put forward by Dean (1984), and used later in his Forbin planner, but his original idea has been heavily modified. The scheme used here allows the representation to be split into multiple contexts, depending on which objects the system is reasoning about, and enables it to deal easily with the integration of qualitative and quantitative data. The qualitative data allows the representation of continuous change as one object either goes through a change (e.g. heats, cools, boils, etc.) or exerts change on another object (e.g. moves, slides, etc.). These continuous self maintaining events are referred to as processes. Adding qualitative reasoning to the planner's representation scheme means that the truth maintenance required needs to be improved above that put forward in existing planners. The favoured system in these planners is the truth maintenance/dependency directed programming approach of Doyle (1979). This will be taken as the basis of the truth maintenance system but will be augmented by the requirements of both process reasoning and qualitative reasoning. The truth maintenance system has to reason about intervals which define the time of occurrence of predicates as well as intervals which define a stated relationship between two quantities required for a process to begin. For example, a process may require quantity A to be greater than quantity B for it to be active. The truth maintenance system must be able to monitor this relationship between the varying quantities A and B as well as ensuring that the relationship between the two quantities continues to hold. If an outcome from an action of process is found which is contradictory to any of the ones expected then an investigation is made to find the probable cause in order that the knowledge base can be re-aligned and synchronised with this new outcome.

Processes can be started by the plan as side effects or they may be required as part of the plan. The action to open a valve may have as a side effect a water flow. A problem arises if the planner wishes to interact with the world to bring a process about as a planned effect. No current planner is capable of doing this and it is a significant short fall in their ability to deal with real world situations. Even in simple examples the ability to interact with processes is vital. For example to make a cup of

tea, you fill the kettle, turn on the gas, wait until it boils and then pour out the boiling water. This requires the planner to have the ability to wait for the water to boil, and to do so only if the available data shows the water is capable of boiling. If there should be a failure in a precondition to the boiling process then the planner should be notified so that the boiling process can be restored. When executing a plan in a world which is constantly changing, there will be times when a plan will fail. This may be because an action cannot execute due to the failure of a precondition, or because a process or event which the planner required to come about will now not occur. This means the planner should have the options to move actions in time so as to avoid any problems, re-execute a single action or set of actions to re-instantiate a predicate which is lost, to create a patch plan and to integrate it into an overall plan which is at fault. If none of these can solve the problem then the plan can be re-generated from the point of failure. In all of these cases the replanner must take into account which actions have already been executed and the effects in place in the real world.

The planner chosen as the basis of the system is Tate's NONLIN (1976). This is the only commercially available planner which is widely known. The complete planning system will be written in Pop11, which is the same language as Nonlin. Nonlin will be modified so that it accepts schemas with definitions capable of handling external processes. The operator schemas which Nonlin uses will be edited by the planning system so as to avoid Nonlin creating patch plans which have redundant steps already met by the main plan. In order to modify the operator schemas at run time there needs to be explicit information present on the preconditions required by a schema and how the actions of operator schema fit together. Nonlin has operator schemas which have explicit types on their preconditions and explicit sequencing information between actions within the schema. This means the schemas can easily be modified for use by the replanner.

In summary, the aim of the research is to build a planning system, using Nonlin as the planner, which has:

1. The ability to deal with unexpected situations which arise due to an incomplete world model by monitoring the effects of a plan as it executes.

2. The ability to interact with the world and to have parts of the plan dependant on outcomes in the real world.

3. The ability to synchronise the internal world model with the real world such that the facts in the knowledge base are consistent and the model is a true reflection of the real world.

4. The ability to diagnose errors in a plan's execution and by means of a replanner put these right while keeping as much of the original plan as possible.

5. The ability to handle temporal information and inferences including changes in temporal information as time passes.

# Chapter 4: System Overview

## 4.1 Introduction to the Plan Management System (PMS)

In order to remedy the deficiencies of previous planning systems noted above, a complete 'Plan Management System' (PMS), is needed. A possible architecture for the conceptual structure of a PMS is presented in Figure 4.1. A Plan Management System differs from a mere Planner in that, as well as planning, it carries out execution error monitoring and analysis and can manage plan patching and re-planning. It is proposed that the PMS should be composed of three major modules: a 'traditional' planner; a system for plan co-ordination, monitoring and error analysis; and, for the present, a simulation of effectors, sensors and the real world.

In the current work the Planner is represented by Tate's NONLIN, which can be regarded as having two major parts: a set of schemas which form a World Model, albeit a very limited and incomplete one; and a Plan Generator. The Plan Generator is responsible for proposing an efficient, self-consistent plan to achieve a given goal, using knowledge from the associated World Model. The plan will consist of partially ordered actions and preconditions, but with no absolute time dependency. Ultimately, the plan must be passed to some 'Executor' which has the responsibility of carrying out the plan in the real world; this part of the PMS can only be dealt with via simulation at present. The Executor requires Effectors, to make the required changes in the real world; and Sensors, to determine and report to the Monitor what is happening in the real world, whether as a direct or indirect consequence of a plan action, or independently of the PMS.

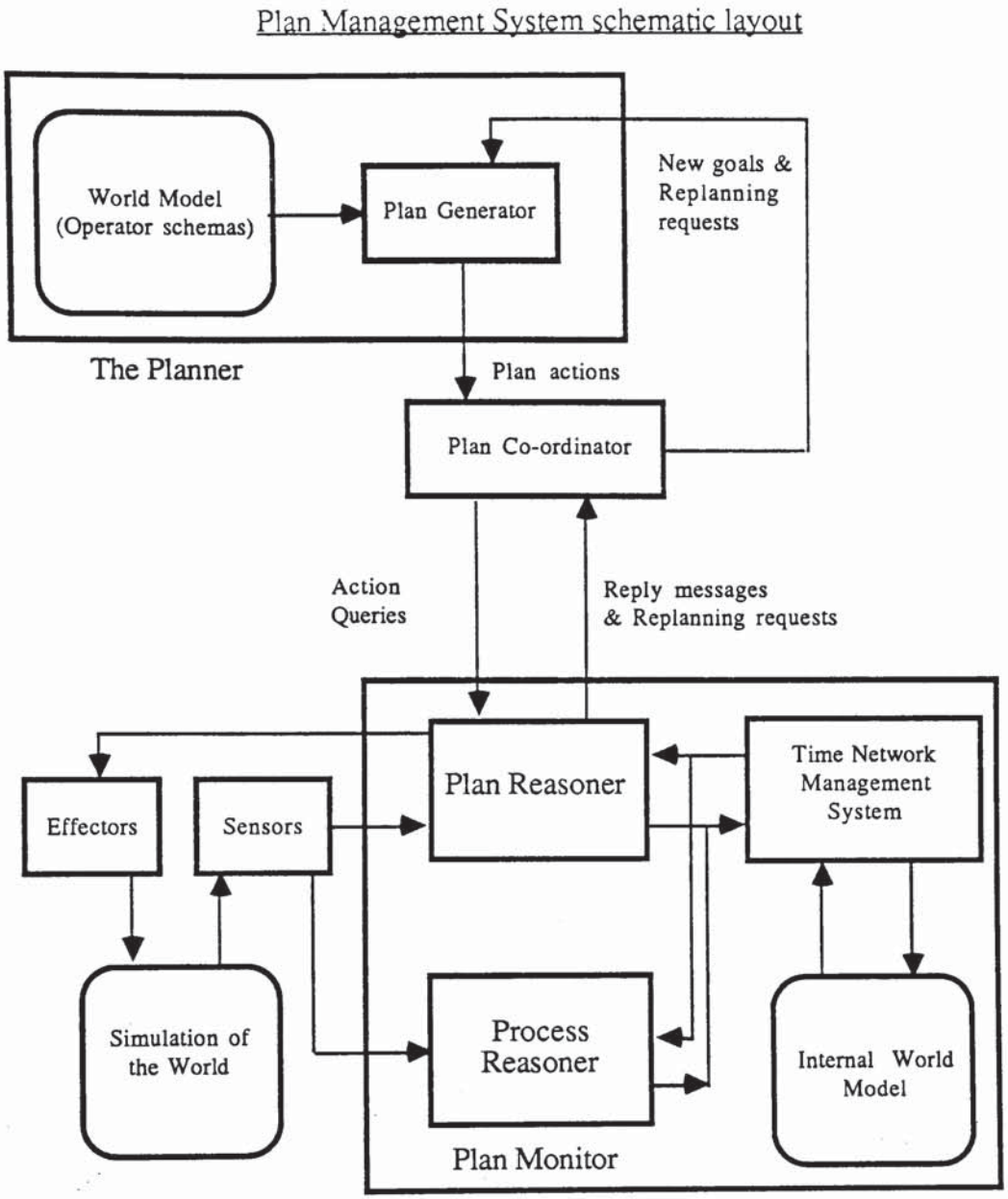



Figure 4.1

## 4.2 Overview of the Structure of the PMS

At the core of the PMS, is a system called Excalibur (Execution Analysis Linkage By Uncertainty Reasoning). At the conceptual level, Excalibur has two major units: a Plan Co-ordinator and a Plan Monitor. The Co-ordinator's function is basically to interface the Planner and the Monitor: it accepts a complete plan from the Planner and stores it. Plan actions are then passed to the Monitor for execution, one at a time. The

Monitor tries to build up a string of plan actions to an optimal depth, which in the current system is set to 10, and this string is used to provide a context for re-planning. In the event of a problem occurring during execution, the Monitor returns to the Co-ordinator with a request for an appropriate 'patch plan' to achieve a particular goal. It is the Co-ordinator's responsibility to transmit this request to the Plan Generator and pass back the new plan fragment to the Monitor.

The motivation behind the detailed design of the Monitor and its associated World Model is an attempt to overcome some of the problems discussed earlier, in particular the complexities which arise in planning and execution occurring over time. It is obvious that the world does not remain static during the execution of a plan and that change is central to reasoning involving time. Thus changes not only cause the retraction of certain facts and the subsequent re–assessment of the knowledge base, but also new information may undermine assumptions about critical hypotheses.

The Monitor in turn can be divided, conceptually at least, into three sections: a Plan Reasoner, a Process Reasoner and a Time Network Management System (TNMS). These are discussed in more detail below, but briefly, the Process Reasoner is needed to reason about individuals and changes in processes occurring in the world which are brought about via plan actions and other processes. The Excalibur World Model is partitioned by the use of 'contexts' - sets of facts and actions which are inter-related by causal influences, where no such influences operate across contexts. Changes in the world may cause contexts to require collapsing (or re-dividing). Maintenance of these contexts requires the extensive use of qualitative reasoning. The TNMS must maintain the integrity of the temporal network contained in Excalibur's World Model, and also be able to respond to queries concerning the temporal relationships among facts and actions. The Plan Reasoner is required to deal with the problem of errors occurring during plan execution. The basic strategy is to analyse the problem in the hope of establishing that there exists a goal which, if it could be achieved via a new plan fragment, would have the effect of patching the existing plan. Management plans are also created in

response to changes in the Planner's world, so as to avoid any undesirable situations, e.g. an explosion in the plant in which the system is placed.

## 4.3 Communication

### 4.3.1 The User Interface

The information from the real world which the system requires to synchronise its internal model is input via the user interface. This allows the user to input information in a psuedo-natural language format which is converted to internal message descriptors which are then sent to the TNMS. The user interface allows the user to set up the initial knowledge base of facts as well as the individuals and their attributes in the real world. The goals which the Planner is required to solve are also input via the user interface. The user can set the Planner a new goal at any time and the goals will be solved in the order in which they are input. The system is not currently capable of planning to have the world set up in an advantageous position for subsequent plans. For example if a plan uses a ladder and the next plan requires the use of the same ladder the the first plan should put the ladder back where it belongs so as to help the second plan. In the current system the world is left in the state it reaches after the execution of the final action. The system does have a primitive explanation facility which allows the explanation of a process tree in terms of what conditions bring a certain state about, the processes and influences which are active in a certain state, and the changes in the state which will bring about transitions to future states.

### 4.3.2 Communications Module

The modules of the system communicate by passing fixed format messages to one another. The Plan and Process Reasoners are separate modules and do not communicate directly with each other. However, both of the reasoning modules communicate with the TNMS. To deal with this message passing there exists a

communication module which resides within the plan monitor. The messages which the module wishes to pass to the TNMS are placed in a 'send table' within the communication module along with the reasoner's identifier and an indicator to tell the communication module if a reply is expected. The communication module scans this table for new messages and sends the message to the TNMS. If a reply is expected then the message is held within the communication module until the reply is sent. Any messages from the TNMS are compared against the outstanding replies first and if a match is found the message is sent to the reasoner concerned. Some messages from the TNMS are not replies such as a process occurring, a execution failure, etc. These are analysed by a parser which on finding a match sends the message to the reasoner concerned.

## 4.4 The structure of the PMS

Each of the modules within Excalibur will now be described. A more detailed description of the Process Reasoner, Plan Reasoner and Time Network Management System can be found in Chapters 5, 6 and 7 respectively.

### 4.4.1 Plan Co-ordinator

The Plan Co-ordinator is responsible for co-ordinating the planning of user goals, the execution of plan actions and any re-planning which a plan may require.

The Co-ordinator receives goal requests from the user and enters them in a plan table. Each of the goals is then sent in turn to the planning system for the required plan to be generated. When the plan has been received it is copied to a table and the next goal sent to the Planner. The Co-ordinator assigns to each plan an index number so that if an action fails it knows which plan it belongs to, as it may have several plans active at once.

The Co-ordinator then sends a request message to the TNMS to find a time period over which the preconditions of the plan's first action are met. The Co-ordinator

carries on querying the TNMS about the actions of the plan until either a failure is noted in an action's preconditions due to interference from the world, or the TNMS has found intervals for the first ten actions of a plan. The actions in a plan have their preconditions met within the plan and not from the outside world hence the entire plan could be set up in the TNMS. However if the plan fails, the amount of processing required to remove all of the actions of an entire plan may be large. Ten actions was thought to provide enough information to create a planning context should re-planning be necessary without leading to excessive overheads if the actions should need removing in the event of a failure.

If a failure occurs in finding a time period over which the preconditions of the action are true or if one of its preconditions is invalidated before it is executed, then a failure is reported by the TNMS. The message indicates the reason for the failure and with this the Plan Reasoner attempts to find a solution. If the solution requires a new plan fragment then a request is sent to the Planner for the required patch plan and the Plan Reasoner goes back to sleep. The request is entered in a plan table which the Planner scans for things to do so when the plan returns from the Planner, the Co-ordinator knows it is not a plan requested by the user but a patch plan which should be sent to the Plan Reasoner for it to work on. The patch plan is copied out in the same way and the Plan Reasoner informed that the patch plan is now available.

### 4.4.2 The Plan Reasoner

A plan fails in execution (and thus generates an execution error) when an action is attempted for which a precondition is not met. Given a full PMS, there are two main ways in which this can happen:

1. The initial plan did not anticipate an interaction, i.e. some precondition of an action was neither checked nor met. Such interactions differ in whether they should reasonably have been foreseen or not. For example, if I set out to produce a plan which explains how to log-on to a computer, it would be negligent not to include a check on whether the system is currently active or is down for maintenance. However, if the building housing the computer has been destroyed in an earthquake, my plan for logging-on is not faulty because it did not include the precondition "check for earthquake damage".

2. The Monitor initiates re-planning which then causes an execution error. This is essentially a fault in the synchronisation process - the internal World Model may contain faulty sense data or faulty inferences may have been made. This will come to light when the Planner tries to validate its assumptions. However, such assumptions may be costly to verify both in terms of resources expended and in the time taken to carry out the checking required.

In the discussion above, the Monitor is referred to as synchronising itself with the real world and it is this process which a PMS finds most difficult. The future in which the PMS creates its plans is not a simple ordered line of facts, but is instead a tree. The time line branches at points of uncertainty and these different paths through the branching future are referred to by McDermott (1982) as 'chronsets'. If the PMS does not commit itself to the correct chronset then an error will occur. A successful Planner must be able to synchronise itself with a reasonable chronset as time passes and be able to notice any discrepancies between the world and its internal model. It is this function which Excalibur carries out for the PMS. The synchronisation process needs to keep track of changes in the world both in terms of the changes brought about by the interference of the PMS (i.e. the execution of a plan action) and changes

brought about by the rules of causation. In very simple domains, it may be possible to use, e.g. the laws of physics, to make precise predictions of causal changes. In more realistic situations, this is either computationally unfeasible or, more usually, the knowledge we have about the world is inexact and our understanding of it is largely based on qualitative rather than quantitative understanding. It is these qualitative descriptions which Excalibur uses in an attempt to model and reason about the world and the changes which happen within it and so create a synchronisation process.

### 4.4.3 The Process Reasoner

One of the major problems in dealing with temporal information is to keep the amount of information stored in the database at an acceptable level. The information in the Excalibur database will often be vague and indeterminate as it contains hypothetical as well as observed facts and events. The problem is to keep down the number of hypothetical futures, while still maintaining an adequate representation of the future to avoid execution error. Uncertainty is present in the form of predictive limitations as well as unwillingness or inability to assign a specific run time schedule to the actions of a plan. To aid in this objective, the system employs a context mechanism which is derived from the qualitative descriptions of the world. The world of the PMS is divided into a discrete set of contexts which are both spatially and temporally bounded. Within a context, facts and actions are known to influence one another, based on a qualitative understanding of the relevant causal laws. No such influences occur between contexts. Contexts are dynamic in that the synchronisation process may discover from input data or by reasoning that in fact an influence does exists between two given contexts, which can then be collapsed into one. Alternatively if separate chains of influence can be found within a context, it can be broken into smaller ones. Reasoning is not committed to one context at a time and facilities are available for reasoning across the World Model if required by the Process Reasoner.

The temporal data base has the ability to represent processes of the form described by Forbus (1983), which allow the system to represent the continuous aspects of the world. Forbus's representation scheme has been modified to allow not only processes to cause processes (the sole assumption), but also to allow processes to be modified by plan actions, and to allow the creation and modification of histories in real time together with the associated truth maintenance involved. As most of today's Planners are unable to deal with external events in time (Vere's (1983) Devisor being the only exception), Excalibur must deal with the intervention of the Planner and analyse any changes which are caused in the world. As these descriptions in their purest form contain no numbers, the number of chronsets through which the world may pass is reduced, thus making the effect of the future more computable and the propagation of effects easier. As more information about the world is included in the descriptions, then more forms of interaction can be detected and resolved, until in the final case a mix of quantitative data and qualitative descriptions can be used.

The representation of changes within the world brings about problems, as we have seen, in representing temporal information; yet change is central to the problem of error analysis and recovery. Change comes about because of two main reasons: by direct action e.g. after executing the action 'open the tap', there is a change in the state of the tap; and by indirect causal consequences e.g. opening the tap may lead to a change in the amount of liquid in the container. In either case there are qualitatively understandable influences between objects which are adjacent in some sense (Davis 1984): physical force applied to the tap causes a movement in its parts; a liquid which is in contact with an opening in a container will flow out of it. In the system outlined here, such influences are represented in description schemas and as effects of the plan actions. Forbus (1983) has outlined a method for representing these qualitative influences, involving 'quantity spaces', with changes taking place at 'boundaries'. For example, if the Planner generates a plan which involves moving water via a pipe from container 1 to container 2, then the process which occurs can easily be described in qualitative terms, i.e. the level of the water in container 1 should

decrease as the level of the water in container 2 increases. The system can reason about what changes will take place in this process, e.g. while level of container 1 > level of container 2 then the process will be active, but at the boundary condition where the levels are the same there will be a change in the process, i.e. it will stop. It is these changes in influences and the reaching of boundaries which the system monitors for.

### 4.4.4 The Time Network Management System

The problem of handling temporal queries about the world is a complicated one in that the world contains several sources of uncertainty. Knowledge of the time of occurrence of future events or of the order in which these events will happen is generally inexact. Beliefs change over time due to the negative and positive justifications a hypothesis receives as the Planner gains new information about the world. Excalibur is capable of detecting and evaluating interactions between plan steps or a plan step and the world. In order to do this it must keep an explicit check on the implicit assumptions underlying the ability of a plan action to be executed. Excalibur assists the PMS in making choices, based on information in the temporal database. Suppose for example that the PMS wishes to execute a plan action whose preconditions are known. It is clear that the PMS would like to be informed if a precondition fails, either before or during the execution of the plan action. A PMS can only successfully create and execute plans if it has the ability to make assumptions about events and their interactions and about the persistence of facts and events, and if it can interpret observations of the effects of plan actions in relation to these assumptions. The ability to predict and anticipate actual events relies upon having the right information in the knowledge base at the right time, which in turn relies upon the ability to synchronise an internal world model with the actual world. For example, we may know from the appropriate rules of causation that "A causes B in the context of C". If the system knows that A has occurred in an interval over which C is true, then it can predict that B will occur. It may then either await data to

confirm this, or if the event is important enough, the system may need to interrupt its current activities to search for the data.In the course of handling a query from the Planner, the system may require to make further commitments in order for a particular deduction to go through. Such commitments generally consist of adding temporal constraints to the network which serve to further define the occurrence of a fact or event, e.g. the Planner may have a plan to move a box through a door but has made no decision as to when the plan should be executed. It might then have another plan which requires the box to be outside the room so by defining when the 'move box' plan is executed, the second action can also be defined. A similar problem occurs when the Monitor is unable to find an interval in which to execute an action due to interference from the world. It should then be able to move the interfering facts away from the required interval by re-defining their begin and/or end points, if possible. When an interval is found, the Monitor creates a list of the assumptions or preconditions which must hold for this interval to hold. This is achieved by using data dependency techniques of the type used by Doyle (1979). This scheme makes sure that the correct relationships hold between the begin points of the preconditions and the begin points of the interval and that there is a similar relationship between the end point of the interval and the end points of the preconditions. As long as these relationships hold then the system will do nothing, but as soon as one of these relationships fails to hold the system will notice this and attempt to sort out the problems which this failure has caused. In the case of a plan action failure, the re-planning process may add extra actions to solve the problem of a future action or reason about the consequences of failure in a current action. In the case of a process failure, the reasoning is more complex as the effects of the process must also be taken into account as they will only persist for the duration of the process being active. One reason why a precondition might be violated is that a fact becomes constrained by facts about the world or another action in the plan, but the main reason for violations is the addition of constraints on the begin or end points of a fact asserting contradictory information.

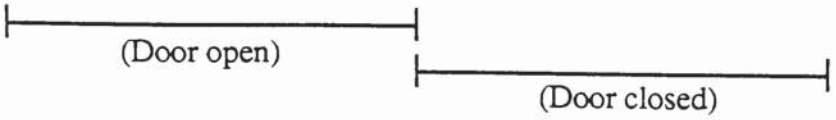



Figure 4.2

The structure in Figure 4.2 is valid as executing an action to close the door would change the status of the door, but if the action was to take longer than the TNMS expected then the structure in Figure 4.3 would initially be present.

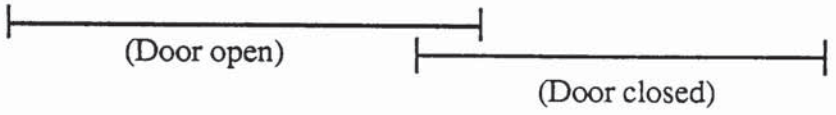



Figure 4.3

This is not a valid structure as the fact 'Door Open' cannot overlap the fact 'Door Closed' as they assert contradictory facts. The rule for persistency clipping is that for any two tokens asserting P and ~P respectively then as long as **before((begin P), (end ~P))** is derivable then there is no problem. The system maintains the interval for a plan action by checking that the begin points of the preconditions are before the begin point of the interval to be maintained, and that the interval's end point is before the end points of the preconditions. However, maintaining an interval involving a process requires more work by the Reasoning System than for a plan action interval. One of the conditions of a process description defines the quantity relationship which must hold for the process to begin and to be maintained, e.g. for a fluid flow between two containers via a pipe, then the pressure in container 1 must be greater than the pressure in container 2. Hence the protection set up for a process interval also requires the system to detect if the inequality no longer holds and then to truncate the end point of the process and any of its effects. The checking of the preconditions and inequalities can be implemented in a straightforward manner using data dependency (Doyle 1979).

# Chapter 5: Time Network Management System

## 5.1 Introduction to the Time Map

The temporal reasoning required to support the Process Reasoner and Plan Reasoner outlined in the previous chapters can be viewed as a special form of a data base management system. The routines and functions required to retrieve and maintain strings of temporally dependent facts (known as a projection), and handle inferences can be described as a Time Network Management System (TNMS). The Process Reasoner and Plan Reasoner can only carry out their function properly if the TNMS provides information which is not only consistent but based on an internal world model as close as possible to the real world. The function of the TNMS is thus to provide a consistent temporal knowledge base of facts, events and processes to aid the Plan and Process Reasoners. The general theory behind time maps is the idea of data dependency. A time map is not a totally linear representation of time but designates a partial order on the points contained within it. Intervals are delimited by begin and end points which are constrained such that the begin point of the interval is before the end point.

The problem of maintaining a consistent set of facts requires the TNMS to synchronise its internal world model with information from the real world. As new information is received from the real world the consistency of projections will be brought into question and new projections put forward. In order to achieve these functions, the TNMS must carry out several important tasks :-

1. As time passes it must accept arbitrary tokens which assert information occurring in the real world. The token can be asserted relative to the time point 'now' as well as relative to an already asserted base token.

2. It must accept query messages from the Co-ordinator, concerning the occurrence of intervals which have certain characteristics in relation to other already asserted intervals in order for an action to be executed.

3. It must accept information about the change to the begin and/or end points of an interval associated with an already asserted token as new information provides constraints about its occurrence in time.

4. It must send annotated messages to the reasoning systems if a problem should arise with a token projection and must then tidy up the projection in accordance with the reasoner's guidance.

5. It must be able to increase the knowledge in the database by searching the contexts to find any processes, views or rules which may be active due to the satisfaction of their preconditions and then inform the Process Reasoner of their activation and occurrence in time.

6. It must accept quantitative information from the sensors about individuals and quantities in the real world.

The following sections define what is meant by a TNMS as well as the events, facts and actions which it contains.

### 5.1.1 Definition of terms in the time map

The problem of handling temporal queries about the world is a complicated one in that the world contains several sources of uncertainty. Knowledge of the time of occurrence of future events or of the order in which these events will happen is generally inexact.

Beliefs about events change over time due to the negative and positive justifications an event receives as the PMS gains information about the real world. A plan to have lunch at 12.30 pm will no longer be justified if I am required to lecture at 12.00 pm. Events occur in time and the events which this research is interested in, events causing actions, can be described as occurring over time. Based on the discussion in Section 2.4 on point-based and interval-based techniques it is convenient to associate with a particular occurrence of an event a time interval delimiting its begin and end points. An 'event type' refers to an event without reference to when it will take place; e.g. 'a general election takes place' is an event type. We can refer to an interval as the time period over which the event type is said to occur and this combination is known as an 'event token'. Thus 'a general election took place on the 11th June 1987' is an event token describing one instantiation of the event type described above. The assertion of an event can also be time dependent as well as time independent i.e. 'it will snow on Christmas day' is independent of the time of assertion where as 'it will rain tomorrow' is not .

Event tokens can be compared temporally by means of the relative positions of their begin and end points, assuming they are mapped to a number line or there is a known relationship between the intervals.

In addition to events causing other events, events can cause facts and processes to become true or believed. For example the event 'open door' will have the effect of making the fact 'door open' believable. As with the assertion of events we have fact types and fact tokens. The begin point of a fact token defines when the fact first became true and the end point defines the point at which the fact is no longer believed. The fact is only believed for as long as the event which causes it is believed. After the event or fact becomes true in the data base it requires no more justification. Facts may also truncate other facts already believed. For example if I close the door in the example above then the fact 'the door is open' would be truncated and the interval designating the 'door is closed' would begin. This is termed 'token clipping' as the earlier token is truncated because of token interference

and will be different depending on which point in time the second fact is asserted. If facts were added to the time map and never deleted or moved then enforcing this clipping rule at assertion time would be fairly simple. However the time map will be used by the reasoning modules which require the flexibility to quickly add, move and delete large event projections. The Plan Reasoner and Process Reasoner need to be able to hypothesise about a given fact occurring at any point in time, but in order to do this the effects of the projection need to be analysed. Suppose we are considering an event token 'open door' occurring at a particular point in time. The event causes facts which may affect the persistence of facts caused by other events which preceded it. In turn, facts the event causes may be affected by events which follow. The Plan Reasoner and Process Reasoner must be able to infer how an event will change the knowledge base at a particular point in time and from these inferences form a projection by examining the current fact and event tokens and the rules of causation. This forms the context of the reasoning i.e. those intervals in time before and after the point at which the token is to be asserted. However, a projection is not fixed because :

1. Facts may disappear which under-pin the projection causing it to be removed from the knowledge base.
2. Facts may be deleted which cause a premature halt in the projection.
3. Facts may be added which cause the projection to be extended which may further interfere with other token projections.

As stated in the previous paragraph, events may cause facts to become true but they may also combine to cause processes to become true. A process is defined as an event which is self sustaining as opposed to an action which is caused by an agent (here the PMS). The process occurs over a period of time delimited by two limit facts. The end fact of a process may not be known and thus it may persist for an indefinite interval of time. Unlike a plan action whose effect can normally be

considered as starting after the completion of the action, a process causes other facts to become true at the point in time at which the process is asserted. Processes are used to represent events such as motion, swing, fluid flow, heating, boiling etc.
For example, a fluid flow has the starting limit fact that the level between two tanks of liquid is not the same and the end limit fact either that the levels become equal or that one tank drains completely. The preconditions which made the process active need to be maintained during the entire time the process is active and if any of these facts are truncated then the process will stop and its end point will be truncated to the point where the interference took place.

Thus the classification of tokens is as follows :

1. Events represent change and events can be actions or processes. An action event is an event caused by an agent and has a known duration. A process event is self sustaining and may be of finite duration. For example, 'open the valve' is an action event; 'water flows' is a process event.

2. A fact describes the results or precondition of an event. For example 'the door is open' is a fact whereas 'open the door' is an event.

In order to check the validity of actions and processes as well as their effects, the TNMS has to keep track of the end points of the action, process and fact tokens which cause them. This is done to maintain the consistency of the time map, by which is meant :

1. No token is allowed to persist beyond the earliest end point in time of the precondition tokens it relies upon for existence.

2. No two tokens asserting contradictory information are allowed to overlap in time. If there exists a token **P** with delimiting points **B1** and **E1**, then for all pairs of points **B2** and **E2** referring to the token **~P** then it is not the case that **B2 < B1 < E2** or **B1 < B2 < E1.** If an overlap does occur then the earlier token is truncated to restore consistency.

This means an entire token projection may be removed by withdrawing a precondition of the action or process which began the projection. The reasoning systems are warned about such failures by a message sent by the TNMS. The projection may be restored by the reasoner by restoring the justification on the failing event providing the Plan Reasoner or Process Reasoner requires the effects of the projection and the cost in time to the reasoning system is not too high. The TNMS must be capable of displacing a projection in time so that the string of causally connected events, processes and facts can be hypothesised to occur at a different point in time. However shifting an event from one reasoning context in which it was formulated to another may threaten the projection. For example I may decide to go jogging at 10.00 a.m, which is a valid assumption according to my current diary. If I am called away to a meeting and I have to postpone it then there is no guarantee I can find a later slot to go jogging.

### 5.1.2 Representation scheme of the time map

The time map is an internal representation of the real world and in the real world there are objects which are referred to in this representation as individuals. In a domain such as process control, the individuals involved are valves, pipes, containers, contained liquids, etc., whereas in a financial domain they are the assets, cash, resources, etc. of a company. The individuals in a domain can be divided up into groups depending on whether or not they are expected to exert an influence on each other. For example two tanks which are not connected together will not normally exert an influence on each other. The example at the end of Chapter 2 about

repairing a circuit in a room with a heater serves as a good example. During what could be described as their normal operation, they occur in separate reasoning contexts. Determining a reasonable context to carry out reasoning about planning and problem solving is difficult. The context required to reason about where I left my pen will not include facts such as 'what time I got up this morning' or 'what I had for breakfast', but will include my visit to the VAX Cluster terminal room and where I had my lunch. This differs from the temporal context described earlier which is defined by the tokens which are adjacent to a given event. The context to which an individual is assigned is only semi-permanent. This is because an individual may be mobile (e.g. a robot) and there may be unforeseen influences between individuals (e.g. the two tanks which were thought to be independent actually have a heatflow between them when one contains a hot liquid). Under these circumstances the contexts which were previously independent need to be collapsed and re-analysed.

The representation scheme used here collects together individuals into influence sets which form an easy indexing method to one of the twenty reasoning contexts in the TNMS. Twenty was chosen as an arbitrary value when the system was created. The primary objects of interest to the system are the intervals associated with facts and events, and the time constraints on the temporal distance between the begin and end points of such intervals. An event can be of any duration including zero (i.e. it is instantaneous), e.g. 'ball enters flame', 'ball leaves flame'. The method of time representation uses the notation of Dean (1983), but the time tokens have been altered so that the begin and end 'points' have ranges. For example, suppose that it is known that lunch lasts 1 hour, but the only information known about dinner is that "it will be between 4 and 8 hours from lunch to dinner, and dinner will last between 1 and 2 hours". If the time at which lunch begins is arbitrarily designated as $T_0$, then the begin point of dinner must lie in the range $T_0 + 5$ to $T_0 + 9$, and the end point in the range $T_0 + 6$ to $T_0 + 11$. Figure 5.1 attempts to reflect this description. Note that strictly speaking lunch is described by an interval whose begin point is in the range $T_0$ to $T_0$ and whose end point is in the range $T_0 + 1$ to $T_0 + 1$.

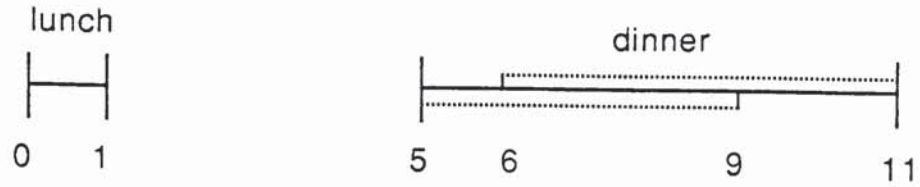



Figure 5.1

The data about a time token's begin and end points will be refined as time passes and the synchronisation process gains more information about a token. The representation allows both numbers and symbols to be integrated in the same scheme. For example, if a fact has an unknown finish time then it may persist into the future for an infinite amount of time. The representation allows the TNMS to assign the time point pos_inf as the finish time of such an interval. For example 'it may rain for between 2 and 3 hours' has a definite finish time where as 'the sun will shine' has an unknown finish time and it would be foolish to assign a finish time of a few million years if we are reasoning about what to do over the next few days. By assigning it a finish time of pos_inf the system merely accepts it as being beyond the end of the numeric values and thus does not commit the system to assigning it a numeric value.

As described above an individual can be referred to as a container or a piece of something. This tag refers to how we see an object, for example if we know nothing about a substance in a container, we can refer to it as a contained something, whereas if we have more information about the substance (e.g. that it is a liquid) then we would refer to it as a contained liquid. These descriptions refer to different views of an object and as we gain more information about an object its view changes through time. For example in the previous case we had a view 'contained something', but by adding the fact that it was a liquid we can define it as a 'liquid' and then by knowing it is inside a container further define it as a 'contained liquid'. These views are only semi-permanent in that the view is maintained by prerequisites which if invalidated will result in the view collapsing. For example we know the temperature of a liquid

must be above its freezing point and below its boiling point, so if we invalidate one of these the individual will change into a solid or a gas respectively. The boil_point and the melt_point described above are quantities which the individual possesses. The quantities asserted for a particular individual vary depending on the individual's function. For example an individual defined as being a container (e.g. a tank, a cup, a dish etc.) will have quantities such as amount_of_in and height whereas an individual described as a liquid will have quantities such as boil_point but not amount_of_in or height. However they both share quantities such as temperature, pressure, location, etc. Each of the quantities of an individual is asserted as a fact token (e.g. has_quantity stuff_1 volume) in the context table, with an infinite duration. The quantities associated with an individual are held in the quantity table and there is a separate quantity table for each of the contexts. The quantity table holds quantitative data about the size and magnitude of quantities and influences. The values held in this table are first indexed by individual and within each individual by quantity. When quantitative data is entered for an individual it is given a token type 'data packet' and assigned to the context table with an infinite duration. This duration will be truncated when new quantitative data is added for the quantity. Thus a quantitative history of change is maintained for each individual, separate from the qualitative assertions. By examining the context table for the data packets for an individual the TNMS can ascertain if any value is known for a quantity at any point in time and can thus reason with both qualitative and quantitative data at the same time. The most basic description of a substance is as a piece of stuff. There are other individuals defined such as tanks, containers, flames, valves, etc. Each of the quantity fields for an individual defines a quantity and contains any quantitative data known about the individual over various intervals of time. The data packet contains: the magnitude of the value; its sign (0 for positive, 1 for negative); the change value (+1 for increasing, 0 for stable and -1 for decreasing); and finally the magnitude of the change. The last entry in the record is the temporal link field which allows further quantities in other more refined views of the same individual to be accessed during

time intervals when the refined view is active. Examples of the quantity fields and link field are detailed in Figure 5.2.

| Volume | [[0 0 0 0 pt1 pt2] [10 0 1 0 pt4 pt5]] |
|---|---|

| Location | [[inside container 1 pt56 pt57]] |
|---|---|

| Link | [2 pt31 pt34] [4 pt56 pt59]] |
|---|---|

Figure 5.2.

As described above the quantity table reflects changes in the view of an individual by creating temporally defined links between different view entries of an individual in the quantity table. These links only allow access to quantities in existence during the time the view is active. For example, if a liquid is viewed as a contained liquid then a new quantity, 'level' is created and only exists while the contained liquid view remains active. The quantities such as temperature, pressure, volume, etc. remain unaltered by the contained liquid view and are not copied to the new view entry. Thus the quantities relating to an individual at any point in time can be found by traversing the links in the quantity table, if the temporal constraints on them permit it. Figure 5.3 describes a hierarchy of views for the individual stuff_1.

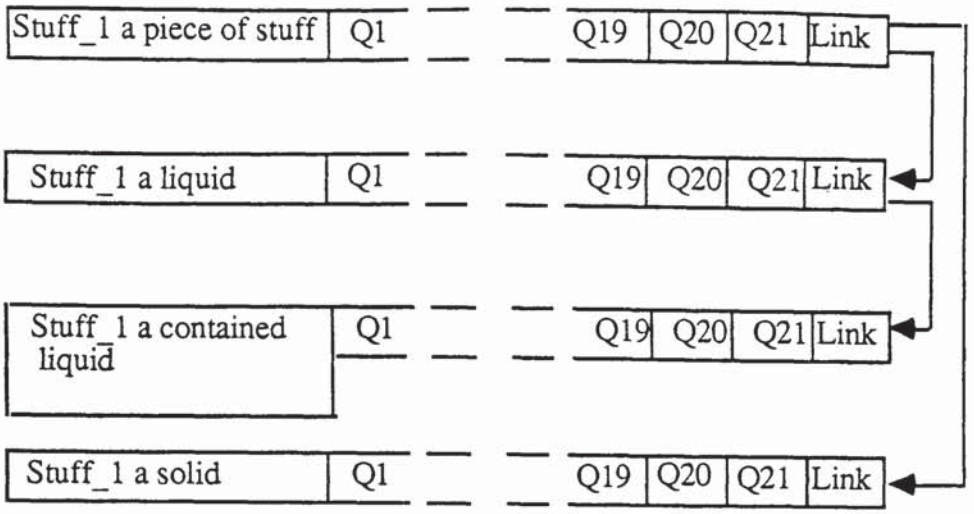



Figure 5.3.

In Forbus's original system a view change created a new individual from an old one. The advantages of using this system of re-defining the individual temporally are :-

1. The number of individuals to reason about is kept down.
2. The system can reason with an object at any level of description e.g. a substance can be viewed as a liquid by one process and as a contained liquid by another.
3. The changes made to one individual are reflected through the levels of views whereas in Forbus's system the reasoner must ascertain which individuals are aliases of each other.

The qualitative data which is present in a domain is held in the quantity space table. As described in Chapter 7 the relationships which hold between quantities can be easily represented in a quantity space. A quantity space reflects the relationships between quantities by creating a partial ordering of the quantities within it. The representation within the quantity space table is the same as for the quantity table except the Q fields are replaced by a double-linked list of the partial ordering of the

quantities. Figure 5.4 outlines the quantity space for the levels and heights of two containers between which a fluid flow is active.

The fields of each quantity space entry define the quantity, its successors, its predecessors and the individual the quantity belongs to. At a later date the process may stop running and the quantity space will be broken up into its constituent parts. For example in Figure 5.4 the levels of WD and WC will no longer be related so there will be separate quantity spaces created for each level quantity in the individuals WD and WC.

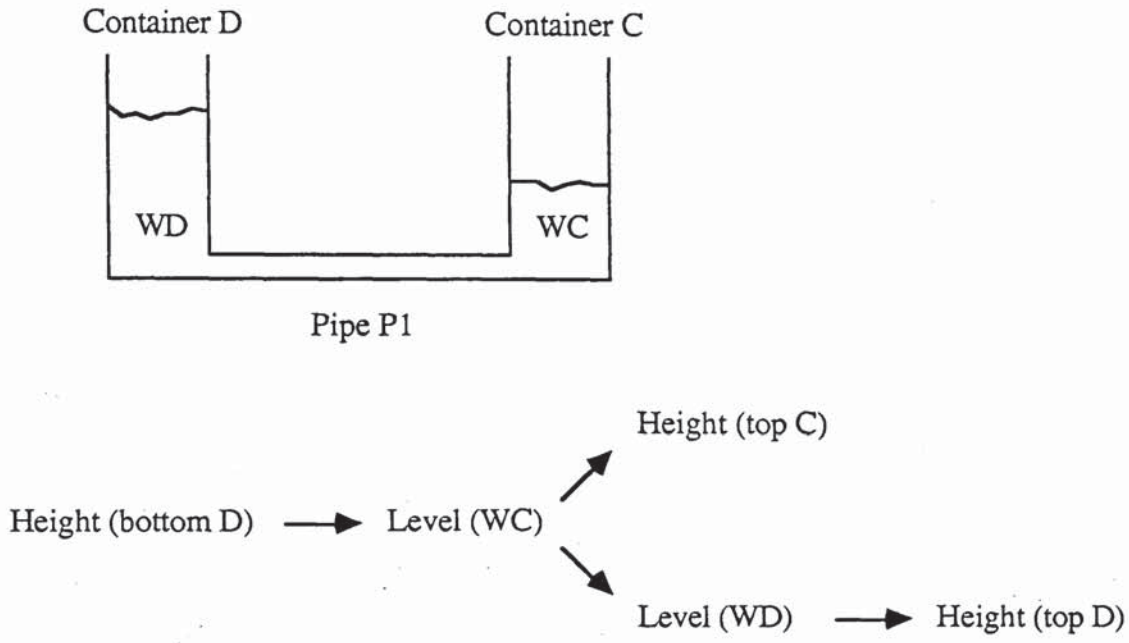



Figure 5.4

If the exact ordering of two values is unknown then the quantity space branches as in the example in Figure 5.4. Without knowing the orientation of the containers the relationship between the height of container C and the level of WD is unknown. The example in Figure 5.5 describes the quantity space of the situation described above. Each of the quantity spaces has a tag to identify the quantity and the fields of the quantity space. Each field is a quadruple which consists of a quantity, its predecessors, its successors and the individual it belongs to. A [0] predecessor or successor means there is no quantity in that direction. By examining which values

are next to each other in the quantity space we can tell what changes will occur. If a quantity decreases then it moves to its left and if it increases it moves to its right. For example if we decrease the level of WD it will move towards the level of WC and if a fluid flow process is active, this will cause it to stop, as a fluid will only flow between a higher and a lower level.

| Level WD | Quantity Space |
|---|---|

[bottom [0] [2] D]
[level [1] [3 4] WC]
[height [2] [0] C]
[level [2] [5] WD]
[height [4] [0] D]

Figure 5.5.

A projection in the TNMS is believed for as long as there exists a justification for the tokens in the projection. To maintain these justifications a truth maintenance system of the kind put forward by Doyle (1979) is used. The justification for a process or action is achieved by justifying each of its preconditions and then justifying each of the effects in terms of the justification for the action or process. In the case of failure of a justification then the action or process may be removed or have its persistence truncated and with it the persistence of any effects it may have. These justifications are held in a table and the TNMS scans it as a background task ensuring each justification is in place. There is a separate dependency table for each context and each entry is one of three different types. Type One is used to justify the begin and end points of two intervals. It is used to justify the relationship between an action or process and its preconditions. Type Two is used to justify the clipping relation between two intervals asserting contradictory tokens. Type Three is used to hold the nodes used in the justification of a process or action. The entries of the dependency table are outlined in Figure 5.6.

Type 1

| Node Number | Relations | Points | |
|---|---|---|---|

Type 2.

| Node Number | Clipping Proc | Justification | Action Proc |
|---|---|---|---|

Type 3

| Node Number | Process or action | Justification | Action Proc |
|---|---|---|---|

Figure 5.6

### 5.1.3 Communication Language of the time map

As described in the system overview the reasoning modules make requests to the TNMS for information about tokens and in turn receive reply messages from the TNMS to these requests. The TNMS also provides annotated diagnostic information about projection failures as well as indicating the occurrence in time of processes and views which are expected by the Process Reasoner and those which are not. In order to allow information to be passed between the modules 'message descriptors' are used. These message are of fixed format and allow the reasoning modules to inform the TNMS of the occurrence of intervals, and their duration and expected start time, and also allows the TNMS to send information back to the reasoner. The message descriptors of the TNMS are as follows:

1. Statement descriptor

   This message requires the TNMS to assert a token either relative to an already asserted token or relative to the time point 'now'

2. Query descriptor

   The query descriptor is used by the Plan Reasoner to request the TNMS to search for an interval which has a certain set of attributes concerning its duration, start time and relation to other tokens already asserted.

3. Time Out descriptor

   The time out descriptor is used by the Process Reasoner to determine if changes in the real world are occurring within the time scale expected. The time out refers to a window of time during which an occurrence should take place.

4. Diagnostic descriptor

   The diagnostic descriptor is used to inform either the Process Reasoner or Plan Reasoner of a failure in the justification of a process or action which subsequently threatens a token projection.This descriptor is unlike the previous descriptors as it is not sent in response to a request from either reasoner. The types of failure can be broken down into those occurring with currently executing actions or processes or those concerning future actions or processes.

5. Process descriptor

   The process descriptor is used by the TNMS to inform the Process Reasoner that it has detected a new process or view in the knowledge base. The process descriptor informs the Process Reasoner of the type of process, the individuals involved and the context in which it was found.

More information can be found on the message formats and examples of their use in Appendix A.

## 5.2 Assertion of External Facts in the Time Map

Information can be asserted into the TNMS in one of two ways. It can be asserted relative to the time point 'now' or relative to another already asserted token In the first case the TNMS scans the contexts to find the one which contains the individual involved. As described in Section 5.1 the system maintains a list of individuals in each context and it is against this list that the scan is made. Once the context is identified the token is asserted relative to the time point which the system holds as 'now'.

In the second case, the processing required is more difficult, as the TNMS must first search for the base interval as described above. When the context is identified then the base token is scanned for, and if it is present it is added to the list of candidate base tokens. If the base token is not present then the TNMS will respond with a message to indicate the problem and abandon the assertion. If the base token used is not unique, i.e. there is more than one token matching the base then the message is ambiguous. This may happen if during a plan a fact switches from positive to negative at different times. For example, a door may be opened and closed several times in a plan, changing its status from open to closed. If we the assert a fact to happen after 'the door is open' then there is an ambiguity over which interval is being referred to. When in doubt the TNMS prompts the user to specify the base from a list of candidate tokens. The candidates are not described by means of their internal representation, but in terms of their duration and their start times from the point 'now'. This ambiguity never occurs with the Process Reasoner as it does not assert facts about the real world in this form. The relationship between the base token and the token to be asserted can be Before, After or During. The relations together with the information on the start time means the relations for overlaps, meets, etc. can be easily defined. Once the base token has been identified, the start time of the new fact can easily be calculated. There are certain rules which the system enforces, such as a token cannot be asserted after a token whose earliest start time is

infinite. This is to avoid the token having infinite start and end times which really means nothing is known about the token.

### 5.3 The Retrieval of Facts from the Time Map

During the synchronisation and plan execution process, the TNMS is required by either of the reasoners to return certain items of information upon request. The items returned will either require a simple search or a more intelligent search. The simple search would involving searching for tokens which match a given pattern. An intelligent search however would include returning not only tokens asserted in the TNMS, but by inference other tokens not explicitly asserted. For example the TNMS may be required to find all those individuals which have a particular quantity or it may be required to find all those tokens involved in a chain of influence; e.g. 'A influences B and A is changing so B changes, now find all the values which are influenced by B, etc.'. The TNMS has a set of routines to carry out the searching and retrieval required by the reasoners. The reasoners pass a parameter and any arguments required by the routines, which in turn send back any tokens which may have been found or can be proved to exist. This set of routines maintains the clear distinction between the function of the reasoners and the function of the TNMS. The types of retrieval which are supported are:

1. Given a quantity, the TNMS will find all the qualitative relations depending on it.
2. Given a plan action, the TNMS will find if any of the side effects of the action have been seen even if the action failed to execute properly.
3. Given a list of tokens of any length, the TNMS will return a list of tokens which match each token in the argument list.
4. Given a quantity, the system will return all individuals and processes which have an effect on the quantity.

5. Given the points assigned to a token, the system will return the actual values from the time line.

## 5.4 The Protection Mechanism of the Time Map

The protection mechanism of the TNMS is responsible for maintaining consistency amongst the tokens asserted. The consistency checks which are carried out are as follows :

1. No fact asserting P is allowed to overlap a token asserting ~P.
2. No fact other than a premise is believed unless a well founded justification can be proved.

The types of consistency will now be dealt with in detail.

### 5.4.1 Token Clipping

When a token is asserted in the TNMS, it is stated as having a persistence equal to the width of the associated temporal interval. The width of this persistence interval will change as new tokens are asserted and moved within the TNMS. If the width of the interval becomes smaller then this is referred to a token clipping. This happens if the negative of a token is asserted and to maintain consistency, the earlier token is clipped to the begin point of the later token so they do not overlap. For example I may assert 'the window is open' to persist over an indefinite time. But if I now assert 'the window is closed', then the token asserting 'the window is open' will be clipped back to the point at which the token 'the window closed' is asserted to begin. This clipping is accomplished by setting up demon tests when the token is asserted which lie dormant in the TNMS until the consistency relation is broken. The clipping justification is created at assertion time such that any token already asserted which is the negative of the token to be asserted is found. Finding the negative of the token

creates a problem as the negative of a token is not that easy to find. For example we could use facts like 'open' and 'not open' which will work quite happily in some domains. However if we wish to assert qualitative descriptions such as 'decreasing', 'increasing' and conditions such as 'stuff_1 is a gas', 'a liquid' or 'a solid' then the problem becomes more difficult. We cannot now say the negative of a token is the token with the word 'not' at the front, as 'not gas' and 'not decreasing' carry ambiguous meaning. In the present system the negatives are held in a list which is matched against the token given and the negative entries returned. Each of the negatives is searched for in the context to which the token is to be asserted. If any are found which would cause consistency problems then they are clipped and their points moved within the time line to their new values. The clipping entry is then asserted in the dependency table for each of the tokens clipped during the assertion. This creates an entry called a clipping node which has associated with it an action procedure which is called if the points move and an inconsistency is detected. This procedure is required to realign the tokens without intervention from the user and then to return to its dormant state. This re-alignment may cause further failures with actions or procedures, and if they occur then they are dealt with by their own special procedures for tidying up process and action failures. An example of this clipping is outlined in Figure 5.7.

Token Asserting P

If we now assert the token ~P then the persistence intervals are re-aligned.

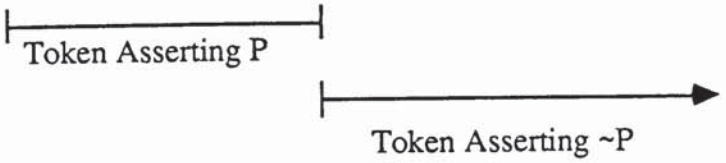
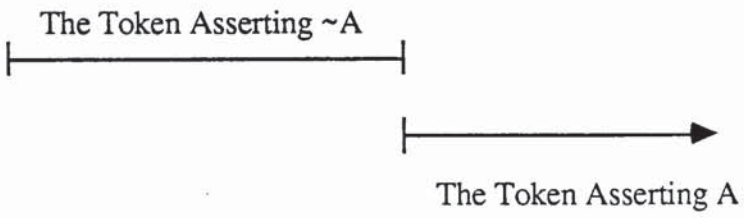



Figure 5.7

Any fact which is involved in a clipping justification by the assertion of a new token is automatically entered into the movement table. This table is required to avoid the appearance of intervals about an individual or event in which nothing is known. For example if token A clips token ~A and we later move the begin point of token A a 'gap' would appear over which nothing is known about A. The representation should avoid this by moving token ~A up to fill in the gap left by token A as in Figure 5.8.




If we now move the begin point of token A a 'gap' will appear.

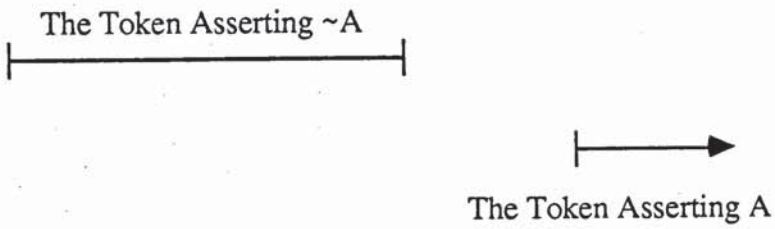



Figure 5.8

This is avoided by having the TNMS automatically re-align the points when the movement occurs. The same is also true if the end of a clipped token was also moved, for example see Figure 5.9.

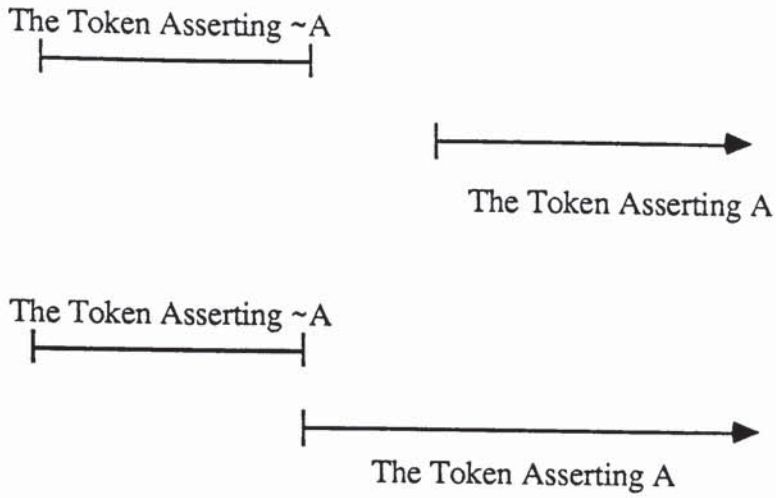



Figure 5.9

However special care needs to be taken when dealing with the effects of actions or processes as they cannot be moved in time to fill a gap if this would cause them to come into being before the action or process which gave rise to them.

### 5.4.2 Process and Action Justification

The protection mechanism maintains a check on the justification of an action or process. The justification needs to be repeatedly checked because an action or process may be capable of execution at the time it is asserted, but as time passes changes occur in the real world which invalidate the justification. The TNMS is capable of detecting and evaluating interactions between plan steps or a plan step and the world. In order to do this it must keep an explicit check on the implicit assumptions underlying the ability of a plan action to be executed. Suppose for example that the PMS wishes to execute a plan action whose preconditions are known. It is clear that the PMS would like to be informed if a precondition fails, either before or during the execution of the plan action. For example, the system decides to execute an action to push a box through a door. The Co-ordinator queries the existence of an interval over which the preconditions of the action are true. It is implicit that the interval returned from the search will be threatened if the temporal relationship between the interval and a precondition change. Consider the example

shown in Figure 5.10. The preconditions to the action 'push box through door' include 'door open', 'next to box' (i.e. the robot or other effector of the planning system can access the box), and 'box moveable'. The action 'push box through door' is possible only if these three preconditions remain true both before and during the action.

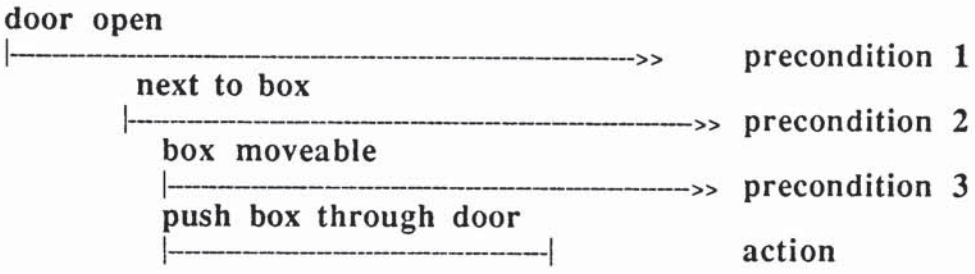



Figure 5.10

The 'door open' fact is represented in Figure 5.10 as an interval whose begin point is fixed but whose end point lies indefinitely into the future. If the interval were to be truncated so that its end–point fell within the duration of the 'push box' action then a problem would occur. Analysed in this way, the difficulty may seem easy to solve, but in fact the solution is not straightforward. If the action is in the future, i.e. its begin point is after the point the system holds as 'now', then the system can attempt to re–plan so as to have the door open by the required time. But if the action is currently being executed and the precondition is suddenly truncated, then the effect of the action can be unknown, since it may not be known whether the precondition which failed was only required as an initial condition, or whether it must be maintained throughout the duration of the action. The picking up of a block serves as a good example here. Two preconditions are 'block on table' and 'holding block'. Almost as soon as the action 'pick up block' begins executing, the precondition 'block on table' will be truncated, but as we know, the action will carry on (see Figure 5.11).


Figure 5.11

However, if we were to truncate the fact 'holding block' then the action will fail, so that the actual effect of the action will be different from the one expected. Knowing the actual effect of an action requires sensors, e.g. equipping a robot with a camera; alternatively, sensors can be simulated by asking the user.

Observation alone is not sufficient to ensure correct inferences. Dean (1983) has a good example of how faulty inferences can be drawn purely from input data. Suppose I want to check whether anyone has dived into a swimming pool. I might:

a. check if anyone is currently in the pool - but this will not prove conclusive because the person may have climbed in rather than dived in;

b. check for water on the side of the pool - but this will not prove conclusive because the person may have jumped in and may have caused some water to splash onto the side of the pool or the person may have dived in and the water may have evaporated.

A PMS can only successfully create and execute plans if it has the ability to make assumptions about events and their interactions and about the persistence of facts and events, and if it can interpret observations of the effects of plan actions in relation to these assumptions. The ability to predict and anticipate actual events relies upon having the right information in the knowledge base at the right time, which in turn relies upon the ability to synchronise an internal world model with the actual world.

For example, we may know from the appropriate rules of causation that "A causes B in the context of C". If the system knows that A has occurred in an interval over which C is true, then it can predict that B will occur. It may then either await data to confirm this, or if the event is important enough, the system may need to interrupt its current activities to search for the data.

As described above an action or process can become active if and only if its preconditions are met. The continued existence of these preconditions with the right temporal constraints means a justification can be found for the process to remain active. The relationships upon which the action or process are dependent are:

1. The latest begin point of all of the preconditions is earlier in time than the earliest begin point of the action or process
2 The earliest finish point of all of the preconditions is later in time than the latest begin point of the action or process

The dependencies between the process or action and its preconditions can be placed in a network which can be thought of as a graph structure whose nodes correspond to belief and whose arcs define support or justification relationships. A node is defined as being IN, which means the associated relation is believed to be true, if there exists a well founded or non-circular justification whose nodes are themselves IN or OUT depending on the type of support relation involved. This is because a justification can depend upon some nodes being OUT as well as IN. Otherwise a node is defined as being OUT. A node can simply be made IN by asserting it i.e. "the ball is green", in which case its justification is the fact we take it as a premise. Nodes which are not premises need justification, which change from OUT to IN and *vice versa* depending upon the status of other nodes in the network which they rely upon for justification. The importance of this is that it supports non-monotonic inference. This means that by changing the status of an existing node we can cause a change in status of other nodes. This will be done by changing the status

of a node from IN to OUT. If I believe my house is safe from burglars and later I find out that six houses in my road have been broken into, then I might wish to re-assess a few of my beliefs. Since the system retains justifications for believing and not believing a node then it can be toggled between IN and OUT depending on changes to the nodes upon which it relies. For example a node justifying an action may be IN until a precondition is invalidated at which point it becomes OUT. The replanner may then replan to have the precondition re-established at which point the justification becomes IN again.

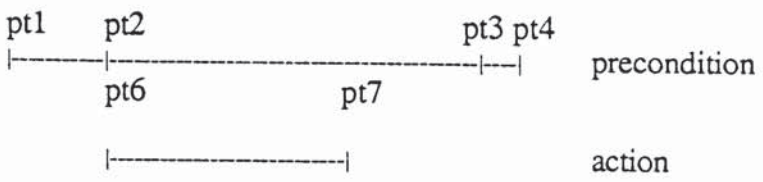
The nodes are associated with two types of relation Defined_before and Defined_during. The two relations will now be described.

Defined_before

The defined_before relation is used to test the relation between the points of an interval asserting a precondition and the points of the interval asserting an action or process. Each precondition defines two defined_before relations.

1. The first is between the latest begin point of the precondition and the earliest begin point of the action or process.
2. The second is between the latest finish point of the action or process and the earliest finish point of the precondition.

For example Figure 5.12 shows the relations defined between one precondition and an action.




| | | | |
|---|---|---|---|
| Node_1 | Derived_before Pt2 Pt6 | ---- | ----- |
| Node_2 | Derived_before pt7 pt3 | ---- | ------ |
| Node_3 | Pick_up_block | Sl([node_1 node_2][]) | |
| | action_proc | | |

Figure 5.12

The action above has only one precondition and hence only two derived_before relations to ensure the correct relationship between the begin and end points of the action and its precondition. The two nodes (e.g. Node_1 and Node_2) are then placed in the SL justification list of the action and this provides the justification of the action. Both of these nodes have to remain IN for the action to be IN and if either Node_1 or Node_2 should change to OUT then the SL justification for the action will fail. The action_proc is the procedure to be executed should the justification fail. This tells the TNMS how to tidy up the knowledge base and the message to be sent to the Plan Reasoner. There is a separate action_proc for the failure of an action, the failure of a process and for the failure of a clipping justification.

## Defined_during

The defined_before relation works well when the precondition is an interval which has points which can be compared temporally. But in the case of a process, one of its preconditions, the quantity relation, is a relation between two quantities. As outlined in the Chapter 5 describing the Process Reasoner, the quantity precondition describes a relationship between two quantities which must be true for the process to begin and be maintained. For example a fluid flow will result between two tanks if the valves

between them are open and the pressure in one tank is greater than the other. This precondition can be satisfied by either finding an interval asserting the relation or by examining any quantitative data for the relation required. The truth maintenance required to maintain consistency between asserted relations (e.g. A > B) and actual quantitative data is carried out automatically by the TNMS. This does not mean the TNMS asserts the relation between every quantity it knows about as a token, but instead it maintains a check on any token asserting a relation input by the user against any quantitative data and clips it as and when necessary.

During the discussion on the quantitative data, the term data packet was introduced, which is an interval over which a quantity has a specific value. This means that over the interval asserting a process a quantity may be updated several times resulting in several data packets. So that checking of defined_during requires more processing than defined_before. The points of the data packets do not normally align to make the test easy, see for example Figure 5.13.

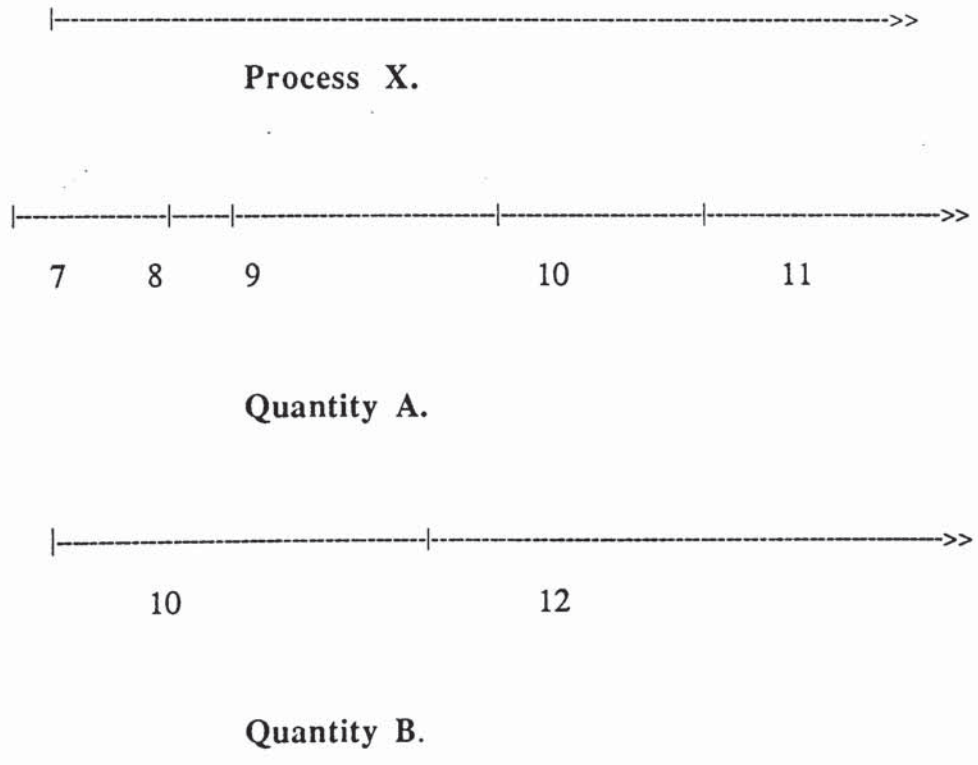



Figure 5.13

If the quantity condition for process X is: 'quantity B greater than quantity A', we can see that the process will be maintained even though the values of A and B are varying.

During the assertion of a plan or process each of the derived_before and derived_during relations are assigned to a node. Each of these nodes is then collected together to form the justification list of the action or process. The justification list is then assigned with the action or process name to a node in the network known as a justification node, which is used as a check for a failure in justification. Should any of the nodes in this list fail and become OUT then the justification will fail and the action or process will become OUT. This justification node will form a part of the justification list of the action or process's effects which themselves will now become OUT. If this effect is now used as a precondition in further nodes it will result in them changing from IN to OUT causing a cascade of tokens to fail. The justification node has associated with it an action procedure which is designed to tidy up failures in a projection such as this. There are separate action procedures for actions and processes. The function of the plan action procedure is to send a failure diagnostic to the Plan Reasoner to inform it of the action which has failed, the precondition which invalidated it and the time of occurrence. This information is required by the Plan Reasoner to create the context to replan if it should wish to do so. The action procedure does not carry out any token alignment by itself as this is done by the replanner in accordance with the replan strategies it has available. The action procedure associated with a process however has a lot more work to do if the failure involves a process. The action procedure must clip the persistence of any relations or influences asserted by the process as they will not exist beyond the end point of the process.

The discussion so far has described process failures, but the same precondition and quantity conditions rules apply to views as well. In the case of a view there is an added complication in that if the preconditions of the view are so constrained by

clipping that the view will never come about, then the temporal link which is inserted into the quantity table also needs to be removed.

When the TNMS has finished tidying up the tables and clipping the persistence of influences and relations, it sends a failure diagnostic to the Process Reasoner. As influences have been removed from a particular process tree the evolution of the tree may change in that processes and views which were thought to come about will now never come about and new processes will be brought into being.

The persistence clipping system used here means a token need only be checked for interaction at assertion time against already asserted tokens. Any negative tokens which are asserted later will pick up this token in its list of tokens to be checked and any clipping checks required will be defined then.

The justification system for processes and plans provides the reasoners with good diagnostic information in order to replan and re-analyse the situation. The non-monotonic reasoning capability of the TNMS allows the Plan Reasoner to try out plan projections at any point in time and to judge correctly any interactions which may occur. This ability also allows the Process Reasoner to synchronise its internal model correctly as the effects of any single change will be propagated through the justifications to find all the possible side effects.

## 5.5 Process and Interval Search and Detection

For a process to come into existence or an action to execute, an interval must be found which satisfies certain criteria. In the case of a plan action the following criteria must be true for the action to execute :

1. All the preconditions of the action are deemed to hold simultaneously over some interval of time.
2. The duration of this interval must be greater than or equal to the duration the action takes to execute.

3. The start time of the interval must fall within a stated start time either from the end of a previous action or from the point 'now'.

In the case of a process the interval must satisfy only the first criterion for it to come into being. Finding such intervals in any one of the contexts within the TNMS is dealt with by special interval search routines. The routines first try to identify a common context which contains all of the individuals mentioned in the preconditions. If such a context can be identified then a scan is made of the context_table_line to try and decide whether the tokens asserting the preconditions are present in the context. Only if these two steps are successful will the routines scan the context_table of the candidate context for the precondition tokens' begin and end points. The final stage is to take each combination of tokens for the given preconditions and check if an interval exists. If such an interval exists and is requested by the Plan Reasoner, then a check is made for the required duration and start time. The plans which are generated by the Nonlin planner have the preconditions for each action met within the plan and never from the external world. The only way in which a precondition can fail is if it is clipped by external events before the assertion of the action which requires it. If this occurs then one or more of the preconditions for the action will fail and the interval search routines indicate, via a diagnostic descriptor, which preconditions are at fault and the time point at which the preconditions need to be made true for the action to execute. It is the function of the replanner to create a plan fragment to overcome the problem with the original plan. If an interval can be found, then in the case of a process the interval is returned to the process instantiation routine which requested the interval search. The process instantiation routines then carry out checks on the quantity conditions before handing the information to the process assert routines. The instantiation routines are in the TNMS but they carry out work for the Process Reasoner and are discussed in Chapter 4. In the case of an plan action the interval search routines pass the information to the plan assert routines. These routines set up the the action and effect

tokens in the context tables and add new justification and relation nodes to the dependency graph. When this is complete the search routines send back a message to the co-ordinator indicating the action has been set up successfully. If any problem occurs after this assertion then it will be picked up by the protection mechanism as detailed in Section 5.4.2.

The interval returned will have a range for the begin points and end points which is calculated from the begin and end points of the precondition intervals. The earliest start time is when all of the preconditions are true and the latest start time is the latest start time of any precondition. When dealing with a plan action the pessimistic view is taken in that an action interval is not asserted to begin until the latest start time of the interval returned. As new information constrains the range of the begin values of the interval, it gives the replanner the option to avoid plan conflicts by moving the action within this interval. The plan actions defined by the Nonlin schemas are defined as having a fixed duration. The planner gives a window in time to each action during which the action needs to be executed for its effects to be in place by the required time. Therefore the plan action duration maximum and minimum is always the same and the start time of the action is the window assigned to the action.

In the case of a process, the interval it occupies will have a varying begin and end range. The process instantiation routines only allow a process to be considered if the latest begin is before the earliest finish. If the earliest finish was to fall in the begin values range then the process may or may not actually ever exist. The TNMS choses to ignore these processes in the synchronisation process for two reasons.

1. The process will come about because of the influences exerted upon it and any individuals which it relies upon. Therefore if the process is expected to occur the influences will have to remain as the Process Reasoner expects. The constraints on the preconditions will define the process later.

2. If the process is not one the system is expecting then there must be a change in influence for the process to come about which will give advance warning to the Process Reasoner that a problem is occurring. As a result of the change in the influences the process tree will be updated to reflect the new expectations which will include the unexpected process.

If the user asserts that an unknown process is occurring in the real world then if there is no information in the TNMS to indicate its happening then this is not a failure of the representation scheme. The processes which arise in the TNMS are sent to the Process Reasoner and checked against the process tree for the particular group of individuals involved. The identification process is carried out by routines within the TNMS which attempt to fill slots in each of the process skeletons which are defined by the user by information from a particular context. If this search is successful then the process is sent to the Process Reasoner for checking. This may add new branches to the process tree e.g. if we close a valve and create a closed tank view then we may expect a new set of outcomes, one of which may be that the tank will explode if it contains a boiling liquid. For a detailed discussion on process instantiation refer to Section 6.4.1.

The time representation system presented here goes beyond that put forward by Dean in his work on Time Maps (Dean 1983, 1984, 1987). In particular, the scheme used here has the following extensions:

1. Processes and continuous events can be represented and reasoned about.

2. The token justification system has ben extended to allow processes to be justified using quantity relationships rather than just temporal relationships, i.e. to allow integration with qualitative reasoning.

3. The representation scheme has the ability to synchronise its internal world model with the real world as time passes and to analyse any

deviation from the expected path, thus re-aligning the knowledge base with the correct future.

4. There is explicit linkage between the effects of actions and the side effects and processes which they bring about in the real world.

## 5.6 Examples of the Use of Time Maps

The examples which are detailed in Figure 5.14 describe the types of reasoning the TNMS is capable of handling. The first example describes a planning problem which involves building a house. Each of the actions has a single effect which is assigned an infinite duration designated by an arrow on its end point. Simultaneous actions such as 'pour footers' and 'lay storm drains' can be easily represented in the time map.

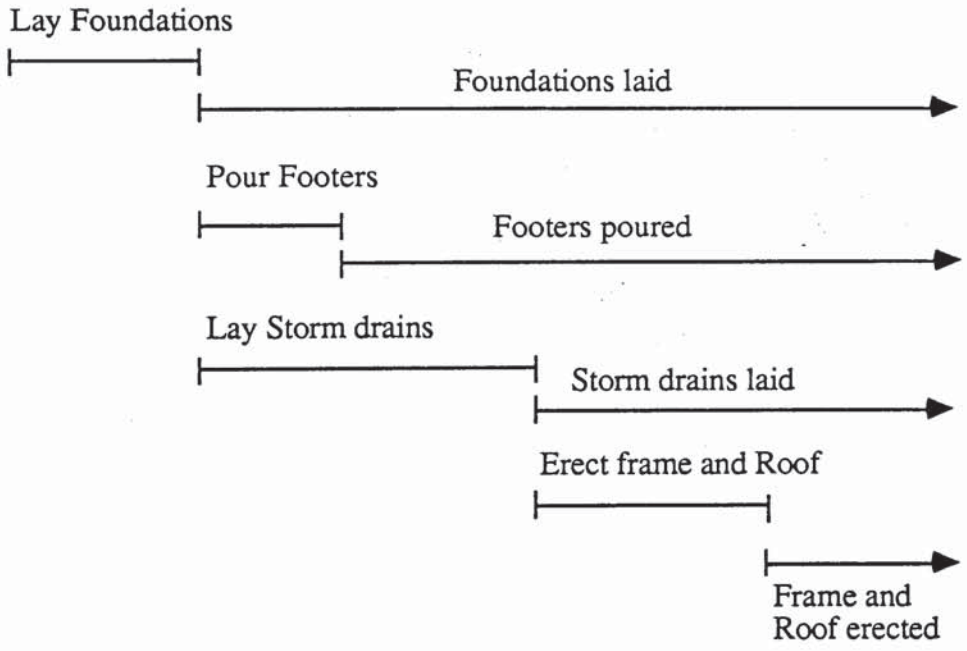



Figure 5.14

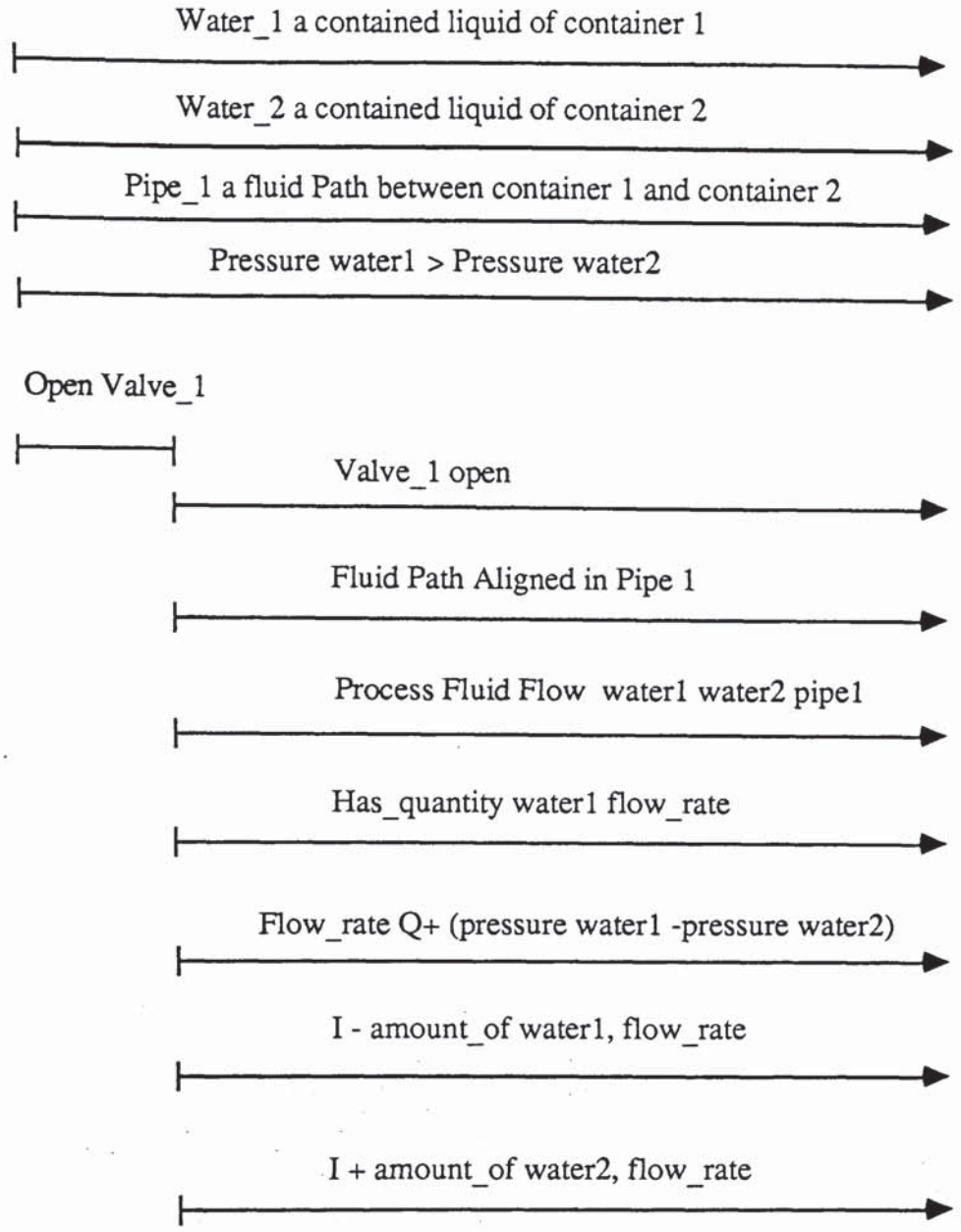



Figure 5.15.

In the second example in Figure 5.15 the facts in the data base give rise to a fluid flow. The example being used is two containers connected together by a pipe which has a single valve. By opening Valve_1 an aligned path is detected by a rule in the TNMS. This differs from a fluid path in that a fluid path is merely capable of carrying fluid (e.g. a pipe), whereas an aligned path means liquid can flow through as all the valves are open. The justification for a path being aligned is stored in the

TNMS so if at a later date Valve_1 is closed the token will be truncated. This together with other fact tokens allows the fluid flow to become active and the process is asserted. The quantity condition, (pressure water1 > pressure water2) is met in this case by a token asserting the relation but it could also be met by entries in the quantity table. The process itself creates a new quantity 'flow rate' which exists between the two tanks, and three new influence conditions. Two of these are direct, namely those affecting the amount of the quantity and one indirect, the influence of the pressure difference on the flow rate. If the process fails it will be these facts which will be truncated from their present infinite finish time to the point at which the process failed.

The final example in Figure 5.16 shows how facts can interact to cause an action failure. The token asserting the door is open is required as a precondition to the action must box through door. When this token is truncated its end point will fall within the interval of the push box action and would be noticed by the non-monotonic reasoner.

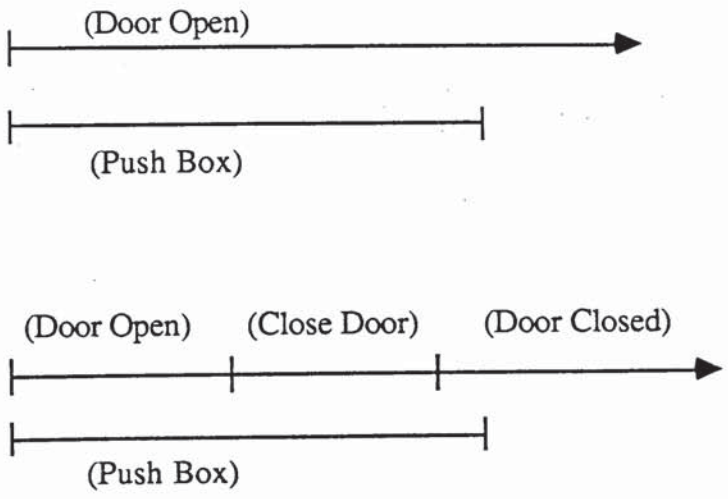



Figure 5.16

# Chapter 6: The Plan Reasoner

## 6.1 Introduction to the Plan Reasoner

The world in which a plan may be executed is constantly changing so that a plan may be executable at one point in time and not at another. For this reason a planner must have the ability to modify its plans to take action if an actual outcome interferes with what was intended. It would also be desirable to respond to advantageous outcomes.

The problem of dealing with initially unknown contingencies can be dealt with by a range of techniques. At one end of this range are the well known goal directed planning techniques which have a complete world model, but do not have the ability to react to unforeseen situations. Moving away from this approach, we have planning techniques such as replanning, conditional planning, sensor planning, deferred planning, and finally reactive planning. At this end of the range, the planner merely creates small plans to solve a given problem, e.g. to move through a door. There is no real overall control, and commitment to a higher directed goal is difficult. The small plans are continually updated as the planner meets unforeseen circumstances. The example most often quoted is journey planning, in that we set off in the direction we want to go, and as we encounter subgoals, such as catching a bus or crossing the road, we create plans to solve them. Moving along this range of techniques, there is a trade-off between the knowledge-intensive approach required to create optimal plans from the beginning, and the reactive capability required in most domains. The middle ground in this range is held by planners which interleave execution monitoring and replanning. This seems to be the most logical way to approach the problem as even the most optimal plan cannot deal with all the dynamic aspects of the world, and a purely reactive approach does not have the overall control required to deal with problems which contain data dependencies or back-tracking. The Plan Reasoner outlined here adopts the approach of interleaving execution

monitoring and replanning within the overall framework of a higher order plan. This allows the PMS to take into account changes in the world which will interfere with the plan as well as to check that the effects of the plan do not bring about a situation in the world which is undesirable.

A planning system would find it very inconvenient if every time it wanted to change part of its plan it was required to plan again from the original goal. In reasoning about everyday situations we continually change our plans by adding new parts, deleting parts no longer required, or modifying existing parts. This ability to replan within the context of an overall plan means that a plan can be modified while still maintaining its overall goal and subplans. For example, one of my plans this weekend may be to cut the lawn, but at the moment it is raining. This does not mean that my entire plan for the weekend needs to be re-examined; merely the part involved in cutting the lawn. I may decide to wait until the rain stops or to do some other task instead. However, this assumes that I have the time to wait or can rearrange my time table, and that I have the same resources available at a later date. For example, I may have borrowed the lawn mower from my neighbour who requires it back by 5 pm. Thus the time requirements and constraints can change according to the point in time when we consider making a change to a subplan. A planning system needs to react according to the plan context which is active at the point in the plan it is considering. It should also have a variety of strategies available to it to solve a replanning problem. Once the planning context for the replanning has been identified, a solution should be generated which satisfies two criteria: firstly, it solves the plan interaction detected; secondly, it does not itself cause any further interactions with the plan.

The Plan Reasoner also has to be able to deal with plan effects which interact with each other as well as changes in the real world which interfere with the plan. The ability of planners to interact with the world is important for the solution of even simple common sense reasoning. For example to make a cup of coffee, you fill the kettle, light the gas and wait for the water to boil, before you pour the water into

your cup. If the actions within a plan are dependent upon external events then problems may arise unless these dependencies are made explicit. For example, suppose a plan to have boiling water in a kettle contained only the actions of placing the kettle on the gas hob, turning on the gas, and lighting it. Once these have been executed, the gas can be turned off, apparently without causing any failures in the plan. However, the goal of acquiring some boiling water will have failed. The problem was that the plan did not contain the fact that this was the ultimate goal.

The types of external events which the PMS can interact with are the creation and destruction of processes and views. A process is a continuous event such as boiling, melting, heating, etc., and a view is a definition of an object as e.g. a gas, solid, liquid, contained liquid, etc. If the effects of the processes required in the plan are not integrated into the overall plan network, then the plan generated may also be faulty. For example, the plan to make a cup of coffee has an explicit 'wait until the water has boiled'. If there were merely sequencing constraints on the actions of a plan to make a cup of coffee, then the action of pouring out the water from the kettle would begin as soon as the gas was lit, and we would end up with cold coffee. If there were merely timing constraints, we might not wait long enough for the water to boil - if the gas pressure were unusually low for example. Neither can the plan simply contain the instruction 'wait for the water to boil' without any understanding of the boiling process, since the planning system may wait for ever if any of the conditions required for boiling are invalidated. The Process Reasoner outlined in Chapter 7 is designed to deal with these sorts of problems by warning the Plan Reasoner that a process which was required as part of a plan has failed. The Plan Reasoner can then plan to alter the world so that the required process comes about after all. In the example discussed above, such reasoning would ensure that if the gas were turned off, a process failure would result (in the boiling process), which would be noted by the Process Reasoner. The failure would then cause the Process Reasoner to alert the Plan Reasoner, which would take steps to rectify the problem.

The next two sections outline the schema definitions and reasoning modules required to carry out analysis and replanning where required.

## 6.2 Schema Definitions

The Nonlin planner used in Excalibur is a non-linear hierarchical planner which uses Task Formalism schemas to define the operators available in the real world. The schemas provide 'chunks' of knowledge of how to perform a particular task. Each schema contains the primitive actions and sub-schemas necessary to carry out its task, as well as information on the sequencing of the actions and sub-schemas and information about the preconditions required by any action within the schema. The preconditions an action may have are split into two main types: 'supervised' and 'unsupervised'. (Nonlin also has a 'holds' type precondition which is not used in the present work). Supervised preconditions are met by the effects of executing other actions within the *same* schema; unsupervised preconditions are met by executing an action in *another* schema. In the original Nonlin system, all preconditions were satisfied from within the plan and not from any external events. However, to solve some problems in the real world, a planning system needs to be able to interact with the world. This requires the plan to contain preconditions over which the planner has no direct control, but which are nevertheless required in the plan. For example, in making a cup of coffee the precondition for turning off the gas is that the water in the kettle is boiling. However, there is no action in the plan which can be executed with the effect 'water is boiling'. Providing the ability to interact with the real world while at the same time leaving Nonlin's task formalism unchanged (as it provides extremely useful information when replanning) means that the changes to be looked for in the real world have to be described in a form which the plan schemas can use as preconditions. As described earlier, the scheme chosen to represent changes in the world is 'Process Theory', put forward by Forbus (1983). Process Theory easily allows the representation of the types of goals and objectives a plan may wish to

achieve. For example, you may wish to fill a container with water, which can be accomplished by opening a valve to let water flow into the container. The schemas of the plan should therefore be able to represent the fact that part of the problem involves the process of a fluid flowing into the container, otherwise errors which may arise due to the failure of the process may not be detected. For example, the valve must not be turned off until the level of water reaches a suitable point, otherwise the container will not be full.

To accomplish the required synchronisation between the world and the planning system, the plan schema formalism has been extended to allow Nonlin to take account of processes and conditions which should arise in the world, and to make explicit the effects of the plan which are required for a process to come about. Synchronisation can be achieved by one of three directives to the loader (which is the part of the Co-ordinator which is responsible for sending actions to be queried to the TNMS):

| | |
|---|---|
| Waitfor | This indicates to the loader that it should wait until a condition is true. For example:<br>Waitfor {level Con_1 = level Con_2} |
| Waitbegin | This indicates to the loader that it should wait until a certain process or view has begun. For example:<br>Waitbegin {fluid_flow con_1 con_2 pipe_1} |
| Waitend | This indicates to the loader that it should wait until a certain process or view has finished. For example:<br>Waitend {boiling stuff_1} |

These directives are declared as primitives, with the effects {achieve condition}, {achieve start process} and {achieve end process} respectively, and then used as supervised preconditions by the actions which follow them in the schema and which

require the directives' outcome as a precondition. The actual effects of the process are not posted in the plan, as they are not directly required.

The only slight problem which occurs is when the individuals involved in a process or condition cannot be specified at plan time because they do not exist. For example, when we make a cup of coffee, we boil 'the water in the kettle'. However, if pouring the water into the kettle is part of the plan, the name the TNMS will give to this new individual (i.e. the water in the kettle) will be unknown until execution time. To solve this problem, the individuals can be expressed in terms of their relationships with other individuals. For example, Waitfor {boiling contents kettle} can be used in the plan and the loader will substitute the actual name of the individual when it is known. This can be extended to any number of individuals, for example Waitbegin {heat_flow contents con_1 contents con_2 pipe_1 next_to box_4}.

When a Wait directive is detected by the TNMS in a query message from the Co-ordinator, it scans the time network for the required process or condition. If the process or condition is present, then any actions which are dependent upon it can be queried by the TNMS (as normal) for an interval over which they can execute. If the process or condition is not active, then the Co-ordinator will ignore the query and mark the plan as suspended. The Co-ordinator then periodically sends a request to the TNMS to query the existence of the process or condition. By default, the Wait directive has a duration of 50 time units. If the Wait directive becomes true before the end of this duration, the plan is moved back in time to the appropriate point, thus shortening the duration of the plan. If the Wait does not become true in this 50 unit time interval, then it is given another 25 units and the plan moved forward in time to reflect this change, thus increasing the duration of the plan.

The suspension of a plan will not cause a 'lock up', because the PMS will only wait for as long as the Process Reasoner believes the process is likely to occur. If the TNMS indicates that the process has failed, or the Process Reasoner indicates the desired state will not come about, then the Wait fails and the replanner is called to deal with the situation. The Wait is thus controlled and will only wait for a defined or

controlled period. When the process or condition does eventually become true, the Plan Reasoner will be notified and will unsuspend the plan, so that the Co-ordinator can query the actions which are dependent on the Wait directive. In this way the Plan Reasoner behaves in what seems to be a human-like fashion: it accepts that a process is continuing and continues to believe in the validity of the plan until it finds information to the contrary. Note that contrary information includes having waited 'too long'. If the outcome is opposite to what was expected, then the replanner can deal with this by means of a set of replanning techniques. Excalibur can thus integrate planning and execution in the real world so that it can react to unexpected situations it may be faced with.

Figure 6.1 shows a possible schema to achieve the goal of making a cup of coffee using this approach. The items @*con and @*cup are variables which Nonlin instantiates at run time, thus allowing the schema to be used in many different situations. The extra primitives which define the requirements of the Wait directives are outlined in Figure 6.2. The primitive to wait for a process or condition has preconditions which make explicit the link between the plan and the process. In this way the Plan Reasoner can infer the problems caused by failures both of a process and of any action whose effect is required to bring the process about. If a process stops because of a legitimate reason, then providing it does not bring about any undesirable side effects it can be ignored. For example, filling the bath can be represented as a fluid flow between the tap and the bath. When the water reaches the required level we turn off the tap, causing the process to stop. However, this is of no consequence as we have reached our objective of filling the bath. However, if the effect of a process is required in a plan and will not now come about then the process's preconditions (as supplied by the plan schema) can be analysed.

As described earlier, the schemas which Nonlin uses to generate a plan are divided up into sections. This division provides valuable information on how the schema accomplishes its task and allows the replanner to modify a schema at run time to deal with a problem in the original plan. For example, if we are building a

house and the plastering of the walls needs to be redone, then we do not need to plan all the actions which preceded the plastering of the walls again; instead the original schema, which contains the actions needed to plaster the walls of the house, should be modified so that it only accomplishes this task. An example of this type of modification is outlined in Figure 6.8.

```
Actschema make_coffee
   pattern        {make a cup of coffee @*con @*cup}
   expansion      1    action {go to the kitchen}
                  2    action {grasp @*con}
                  3    action {pick up @*con}
                  4    action {take @*con to tap_1}
                  5    action  {put @*con under tap_1}
                  6    action  {turn on gas_1}
                  7    action  {waitbegin {boiling contents @*con}}
                  8    action  {turn off gas_1}
                  9    action  {place coffee in @*cup}
                  10   action  {take @*cup to @*con}
                  11   action  {fill @*con}
                  12   action  {fill @*cup from @*con}
   orderings      sequence 1 to 10
   conditions     supervised  {in the kitchen} at 2 from 1
                  supervised  {holding @*con} at 3 from 2
                  supervised  {@*con held} at 4 from 3
                  supervised  {@*con at tap_1} at 5 from 4
                  supervised  {under tap_1 @*con} at 6 from 5
                  supervised  {gas_1 alight} at 7 from 6
                  supervised  {achieve start process} at 8 from 7
                  supervised  not {gas alight} at 9 from 8
                  supervised  {coffee in @*cup} at 10 from 9
                  unsupervised        {@*con on gas_1} at 6
   vars           con undef
                  cup undef;
end;
```

```
Actschema fill kettle
   pattern        {fill @*con}
   expansion      1    action {grasp tap_1}
                  2    action {turn on tap_1}
                  3    action {waitfor {level contents @*con = fill level @*con}}
                  4    action {turn off tap_1}
                  5    action  {ungrasp tap_1}
                  6    action  {take @*con to gas_1}
                  7    action  {put down @*con on gas_1}
   orderings      sequence 1 to 7
   conditions     unsupervised{under tap_1 @*con} at 1
                  supervised  {holding tap_1} at 2 from 1
                  supervised  {tap_1 on} at 3 from 2
                  supervised  {achieve condition} at 4 from 3
                  supervised  not {tap_1 on} at 5 from 4
                  supervised  not {holding tap_1} at 6 from 5
                  supervised  {@*con on gas_1} at 7 from 6
   vars           con <: non tap_1 :>
end;

Actschema fill cup
   pattern        {fill @*con}
   expansion      1    action {grasp @*con}
                  2    action {pickup @*con}
                  3    action {pour contents of @*con onto @*cup}
                  4    action {waitfor {level contents @*cup = fill level @*cup}}
                  5    action  {put back @*con on gas_1}
   orderings      sequence 1 to 5
   conditions     unsupervised{under @*con @*cup} at 1
                  supervised  {holding @*con} at 2 from 1
                  supervised  {@*con held} at 3 from 2
                  supervised  {pouring contents of @*con into @*cup} at 4 from 3
                  supervised  {achieve condition} at 5 from 4
   vars           con <: non tap_1 :>
                  cup undef;
end;
```

Figure 6.1

primitive

{waitfor === } with effect + {achieve condition} : 50

(waitbegin ===} with effect + {achieve start process} : 50

{waitend ===} with effect + {achieve end process} : 50

Figure 6.2

## 6.3 Replan Strategies

The replanning strategies which are used by the Plan Reasoner are detailed in the following sections. The strategies use different methods which take into account the differing types of errors which can arise.

### 6.3.1 Re-scheduling of an action

An action in a plan can fail because one (or more) of its preconditions is altered in such a way as to make it false over part of the time interval during which the action would be executed. This may be due to a precondition not becoming true until later than was expected. For example, I may have planned to meet a friend at 1 pm for lunch (which will take an hour). The plan requires that he be free at that time. Suppose he is detained in a meeting, and only becomes free at 1.30 pm. This may or may not cause problems depending on how tight my schedule is: if I can afford to take two hours for lunch, then I can wait; however, if I am restricted to an hour, the original plan will need to be modified. Within Nonlin, each of the actions of a plan is assigned a time window during which it should be executed. The length of this window may be larger than the duration of the action. For example, in Figure 6.3, action A3 has preconditions P1 and P2. P1 is active after time 4, but is not required by A3 until time 12. This means action A1 can be executed any time between time 2 and 10 and still have its precondition P1 in place by time 12, whereas any movement in A2 will cause a further knock on effect with A3.

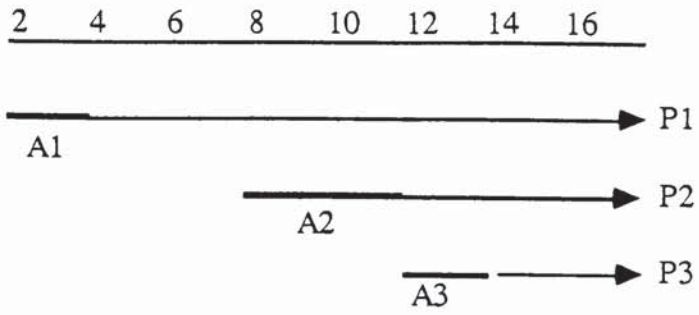



Figure 6.3

In Excalibur, actions are scheduled to execute as soon as possible so as to take advantage of any 'slack time' between the end of an action's execution and the execution of any other action which requires that action's effect as a precondition. This slack time can sometimes be used to cope with a late starting precondition, without causing any further problems, by moving the action dependent on the precondition forward in time to the new starting point of its faulty precondition. For example, if a precondition of A1 was late in starting and only became true at time 8, we could move A1 to time point 8 without causing any problems in the plan. However, there are circumstances when the action cannot be moved in time without causing further problems. The output from the Nonlin planner is effectively in the form of a PERT chart, so that there is a critical path of actions. If an action on this path is late starting, or the delay in such an action is such that it cannot be executed again by the time it is required, then the plan cannot be finished in the expected minimum time. To overcome this problem, the Plan Reasoner moves the faulty action, and all the actions which are affected by its late finishing, so as to re-align the whole plan. The time delay is propagated through the actions of the plan until the end of the plan is reached, or a particular path has enough slack time at one node to absorb the change. As the plan is specified as a PERT chart, there can be more than one path to a particular node which may cause the earliest and latest finish times at the node to be updated many times during the propagation of the delay. If this re-alignment causes a change in an action already sent to the TNMS, but not yet executed, a message is sent to the TNMS to move the action to its new execution point. This is easily done and requires no changes to the non-monotonic

justifications of the actions. (These justifications are generated and maintained by the non-monotonic reasoner which is a truth maintenance module within the TNMS). In order to avoid any unnecessary processing of the plan when a failure is detected, any other actions which will also fail as a result of the initial delay are found when the first error message of a possible set arrives. This displayed diagrammatically in Figure 6.4.

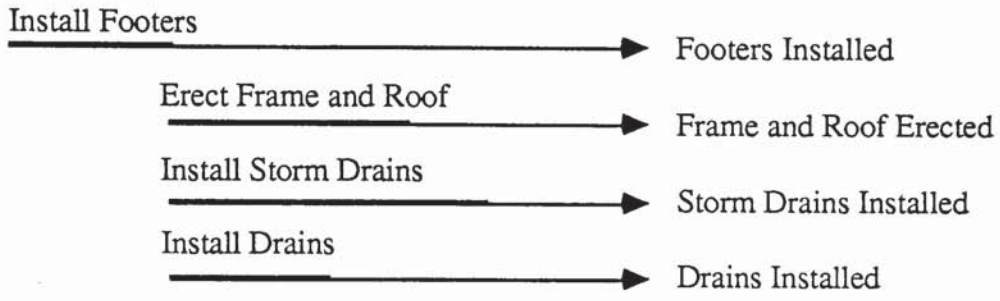



Figure 6.4

If the action 'Install Footers' were to to take longer than expected, the actions 'Erect Frame and Roof", 'Install Storm Drains' and 'Install Drains', which rely on its effect 'Footers Installed', as a precondition will have problems. The non-monotonic reasoner will send an error message for each of the problem actions. When the first error message arrives, which, for example, may be for action 'Erect Frame and Roof', the Plan Reasoner will search for any other actions which will fail as a consequence of the change to 'Footers Installed'. This stops the Plan Reasoner from re-analysing the plan more than is necessary and also stops the Plan Reasoner trying to analyse a failure message about a plan which has changed since the message was generated.

### 6.3.2 Re-execution of an action

An action can also fail because the end point of one or more of its preconditions is truncated within the time interval during which the action will execute. Unlike the previous problem of the precondition beginning late, this problem can affect both

future and currently executing actions. In the case of a current action, the Plan Reasoner must try and determine if the effects of the action have been achieved even though it was interrupted. Potentially, this can be solved by having the Process Reasoner record within the process tree any arcs which are dependent upon plan actions. In this way, the Plan Reasoner can check whether or not to remove the effect of the action from the TNMS. For example, if the preconditions to opening a valve are 'you are holding it' and 'it is moveable', and the valve becomes stuck, then the outcome of the action is in doubt. However, if the action was expected to generate a fluid flow process, then this or any of its effects can be checked against the process tree created by the Process Reasoner. If there are no side effects to be observed, then the Plan Reasoner queries the user (as in the present simulation no sensors are available to provide the necessary information).

If the problem is with a future action, then the Plan Reasoner must try to restore the precondition while minimizing the amount of disruption to the plan. For the faulty precondition to be restored, the action which provides the faulty precondition needs to be re-executed while still maintaining the continuity of the existing plan. The action which is re-executed is referred to as the 'repair action'. This can sometimes be achieved easily if the preconditions of the repair action are still in place at the required re-execution point. The repair action should be executed so that its end point is before or co-incident with the end point of the faulty precondition. In this way there is no break in the precondition required for the faulty action, and the overall plan will not be affected by its re-execution, since the effect of the repair action was required at the point of failure and by re-executing the action, the effect will be in place again so as to satisfy the requirements of the plan. However, the non-monotonic reasoner will still be using the interval which was clipped in the justification of the failed action. If the faulty precondition was clipped by the user asserting a new finish time for it, then there will be no negative of the failed precondition adjacent to or 'meeting' it in the time map. As detailed in Chapter 5, any token which is added to the time map will be automatically clipped by any negatives

already asserted. Thus if there is no negative token 'meeting' the precondition token which was clipped, we can assert its effect as now having infinite duration after the patch action is executed, thus satisfying the non-monotonic justification of the action. For example, suppose that the precondition 'Drains Installed' is required to be true throughout the duration of the action 'Finish Grading', but that it fails and is truncated as in Figure 6.5. If 'Install Drains' can be executed in parallel with 'Finish Grading', we can patch the plan and then move the end points of 'Drains Installed' to infinity; any negative token which is already asserted will have a justification to clip it if necessary.

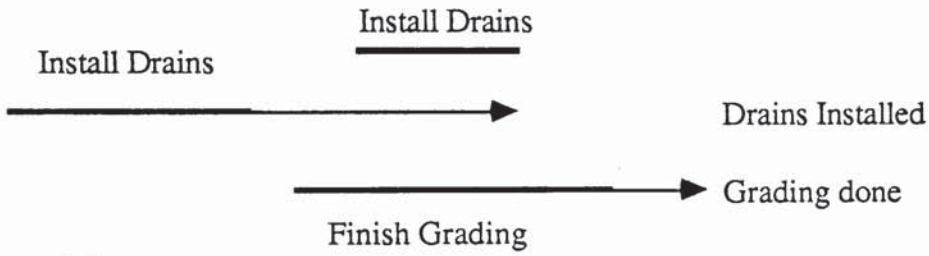



Figure 6.5

If however the precondition was clipped by the assertion of negative token then the precondition cannot be simply extended to infinity as above. Instead we first have to remove the negative token from the TNMS, together with any inferences and projections dependent upon it, before we extend the precondition's duration. However, even if the action is able to re-executed because its preconditions are still in place, there may still be problems. The action may have a duration greater than the time between the notification of the problem and the time the effect of the faulty action is required, or the repair action may require to be re-executed during the original execution of the action. This is outlined in Figure 6.6.

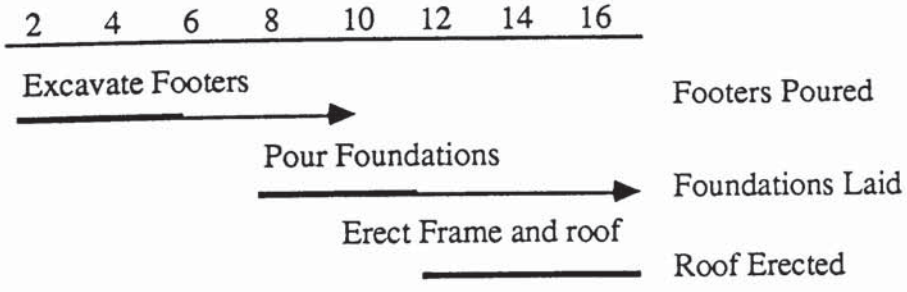



Figure 6.6

In Figure 6.6, the precondition 'Footers Poured', which is required to be true over the entire interval of the action 'Pour Foundations', is truncated from time 16 to time 10 causing an error. The action 'Erect Frame and Roof' requires 'Pour Foundations' effect, 'Foundations Laid', as a precondition, to begin at time 12. However, action 'Excavate Footers' which provides the faulty precondition 'Footers Poured' cannot be re-executed in the interval 10 - 12 as it has a duration of four units. In this case action 'Excavate Footers' will be executed and the rest of the plan re-aligned in accordance with the late start of the action 'Erect Frame and Roof'. The re-alignment is done in accordance with the method used to deal with the problem of a late starting precondition.

If the repair action cannot be executed in the required time so as to have the old precondition interval and the new precondition interval meeting, then there is a problem. To overcome this, the repair action is re-executed so as to minimize the gap between the precondition which has failed and the end of the repair action, thus minimizing the disruption to the overall plan. However, this modification to the plan also needs to be updated in the non-monotonic reasoner as it will still be using the interval of the failed precondition in its justification for the failed action. The Plan Reasoner sends a message to the non-monotonic reasoner about the plan failure and the new action whose effect is to be used to re-justify the failed action. The action can simply be moved in time, whereas the justification needs altering. As the action is moved in time it may cause further late start precondition problems. To avoid these, the Plan Reasoner re-aligns the rest of the plan in accordance with the changes in the repaired action. As mentioned earlier, the repair action is re-executed so as to minimize the amount of time the failed action is moved, thus reducing the amount of re-aligning required. This is displayed diagrammatically in Figure 6.7, where the precondition P1 ('Foundations Laid') is truncated within the action (Fig. 6.7a). Due to constraints on the repair action, it can only begin execution at the end point of the failed precondition. The repair action action is re-executed to reinstate the

precondition P1' (Fig. 6.7b). The failed action is then moved to be aligned with this new interval so as to satisfy its non-monotonic justification (Fig. 6.7c).

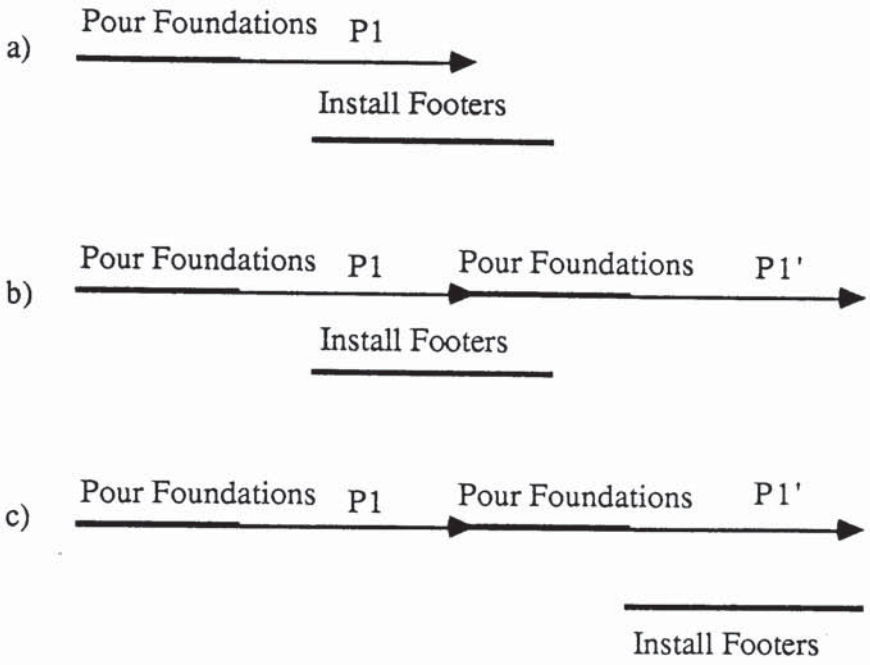



Figure 6.7

### 6.3.2 Patch Plan Generation and Integration

In the case of a simple repair action, there may arise the problem of the precondition of the repair action not being correctly aligned so as to allow the repair action to be re-executed. In this case the repair action itself needs to be repaired before any attempt is made to patch the failed action. However the repair action which repairs the repair action may itself be unexecutable, due to a problem with one of its preconditions, and this may cascade all the way back to the beginning of the plan. To avoid generating multiple requests to the Plan Reasoner to patch each of the failed repair actions, the replanner creates a 'patch plan' - a plan which contains only those actions required to execute the original failed action. This can be achieved by reasoning backwards through the predecessor links from the failed action to the beginning of the plan. These actions are the only ones required to patch any repair action which may have failed. The other actions of the plan which have already been

executed or provide preconditions not required in the patch plan can be ignored. Figure 6.8 shows part of a plan with the successor and predecessor links between its actions.

| No | Action | Predecessor | Successor |
|---|---|---|---|
| 1 | Plan_head | [] | [3] |
| 2 | Dummy | [13 12 11] | [] |
| 3 | Dummy | [1] | [23 14 4] |
| 4 | Excavate and pour footers | [3] | [5] |
| 5 | Pour concrete foundations | [4] | [16 15 6] |
| 6 | Erect frame and roof | [5] | [22 19 7] |
| 7 | Lay brickwork | [6] | [8] |

Figure 6.8

If we require a patch plan to achieve 'Lay brickwork' then we could follow the predecessor links back and obtain the actions 'Erect frame and roof', 'Pour concrete foundations' and 'Excavate footers'. The replanner uses this list of required actions to modify the plan schemas so that they are the only actions to be achieved. This can be done quite easily, because the structure of the Nonlin schemas allows actions to analysed individually, providing information on the ordering constraints between actions as well as on how the preconditions of a particular action are achieved. Thus the Plan Reasoner can remove actions from a schema and then re-calculate the ordering information and preconditions required for the remaining actions. The preconditions in the schemas are of two main types, 'Supervised' and 'Unsupervised'. Supervised precondition are preconditions which are achieved by another action in the same schema, and Unsupervised preconditions are those in which the precondition is met by an action in another schema. The preconditions can be modified by following a simple set of rules:

1. An unsupervised precondition:

   if the action is no longer required then

      remove the precondition

   else

      update the precondition to reflect the action's new number.

2. A supervised precondition:

   If the action is no longer required then

      remove the precondition

   elseif the action which provides the precondition has been executed then

      change to an unsupervised precondition and update the action numbers

   else

      update the precondition to reflect the new action numbers.

The updated schemas are then sent to Nonlin and a plan generated to achieve the failed action. The replanner uses this patch plan to patch any failed repair actions as required. This technique has been used sucessfully to generate patch plans for construction tasks involving house building but unfortunately due to space constraints cannot be shown in this thesis. The repair action cannot be executed immediately so as to repair the failed action so a patch plan is generated. In the example there is only one failed repair action but the technique is applicable to any number as the patch plan can be "cannibalised" for the required repair actions.

## 6.4 Plan process failure and avoidance

A patch plan can also be generated to solve two further problems, namely an undesired situation or an unforeseen circumstance. An undesired situation is a predicted occurrence in the world which the planner wishes to avoid. An example of

this is an explosion in a container, which could be averted by releasing the pressure build up by opening a valve, which is itself a simple plan. An unforeseen circumstance is a failure in the plan which requires a precondition to be satisfied by a method other than patching the failed action. An example of this would be a plan failure caused by the amount of water in a tank not being sufficient for a particular task. The patch plan would then be a method of putting more water in the tank. The re-execution of a set of actions or patch plan is the most common way of rectifying this type of problem with a plan. It is similar to a single action, but there are additional problems. With a single action, we know exactly the effects of the action and that these effects were required in the plan. However, when we integrate a patch into the original plan network, we obtain not only the effects we do want, but also other side effects of the patch, and these may interfere with the plan we already have. These two problems are dealt with in detail in the following sections.

### 6.4.1 Unforeseen circumstances

The patch plan may be requested by the Plan Reasoner because of an unforeseen circumstance. For example, if we are filling a cup with boiling water from a kettle and the kettle becomes empty before the required amount is dispensed, then the filling process has failed because we have not reached the required amount of liquid in the cup. This failure cannot be solved by the techniques outlined earlier for plan action failures and can only be achieved by analysing the requirements of the process. The failure is analysed according to its type and a strategy put forward to solve it. For example, in the case of a failure to reach the required level (as above), the Plan Reasoner attempts to find a way of achieving more of the individual which is lacking. This is solved by having the Plan Reasoner scan the schemas to find one which contains an action which would create 'more' of the individual (in reality, a new individual of the same kind). Once this action has been found, the Plan Reasoner scans the original plan to find all the changes which the new individual must go through to make it useful in the plan. In the case of the 'make coffee'

example, the new water must first be boiled, otherwise we would end up with cold water in the coffee.

### 6.4.2 Undesired outcome

Another situation requiring a patch plan is when a future event is predicted which the Process Reasoner wishes to avoid, e.g. an explosion. To deal with this problem the Plan Reasoner again uses a method which is dependent on the type of problem. These methods are implemented as demons and are awoken by the message indicating the problem from the TNMS. This informs the demon of the individuals involved which can then be used in the analysis. For example, an explosion is caused by a pressure increase in a sealed container, and thus the line of reasoning to be followed is that the pressure should be released. The demon will find the valves of the individual which is predicted to explode and will request a plan to open one of these. The Plan Reasoner then tries to find a plan which will achieve the required effect. It first has to identify those actions which will be in the new plan schemas. In the earlier discussion, the original plan could be used followed to provide this information. However, as this is a plan to deal with an unforeseen situation, there will be no plan for the Plan Reasoner to analyse. The Plan Reasoner could scan the schemas to find a primitive or schema which provides the required effect. Once this has been achieved the Plan Reasoner could then scan the condition lists to find the actions required to provide the preconditions. This is easy to calculate; however, the result can sometimes be ambiguous. For example, in the coffee making example in Figure 6.3, the preconditions for 'having the kettle under the tap' can be obtained by following back the chain of supervised preconditions to obtain 'go to the kitchen', 'grasp the kettle', 'pick up the kettle', 'take kettle to tap'. However there are situations when this can cause more actions to be picked up than is necessary. In the case of 'light the gas' there is no chain of conditions to follow back and if we simply execute the actions before it in the schema then we obtain filling the kettle as a prerequisite for lighting the gas, which clearly it is not. As a result the matching is

done against sub-schemas, e.g. we would look for a schema 'open_valve @*con' rather than a primitive. Thus when creating the schemas, the designer must be conscious that any part of the plan which can be generalised or re-used should be placed in a sub-schema. Once the actions have been identified then the plan analysis can proceed as before. In both the unforeseen circumstance and undesired situation, after requesting the plan the Plan Reasoner is suspended until the plan is available.

When the patch plan is completed by Nonlin, it is picked up by the Co-ordinator which passes it to the Plan Reasoner. The Plan Reasoner then has to carry out two main tasks. Firstly, it has to find a point in the plan where the patch plan can be integrated successfully, and secondly it has to modify the action numbers, predecessors and action links between the plan and the patch plan. The point of insertion is found by calculating the duration of the patch plan and integrating this so that the effect is in place before the time of the undesirable outcome is predicted. When the patch is integrated, care must be taken not to invalidate any dependencies in the plan. In the case of an unforeseen circumstance, if the point of insertion would cause any problems then the Plan Reasoner moves the point of insertion to a point where the dependency will not cause problems. For example, the patch may truncate a precondition part way through an action which requires it . The patch is then moved so as to avoid this problem. This may cause further problems with late starting preconditions, but these can be dealt with using the techniques outlined in Section 6.3. In the case of a patch to deal with an undesired situation the patch cannot be delayed. Hence the patch is inserted at the point required and any actions which are invalidated are re-scheduled to occur after the patch plan. This is solved using the techniques used to re-execute a repair action which cannot be aligned so as not to cause a break in a precondition, as outlined in Figure 6.7.

# Chapter 7: The Process Reasoner

## 7.1 Introduction to Qualitative Reasoning

The actions of a plan are executed to bring about a desired change in the world so that a goal is achieved. Reasoning about change is thus at the centre of execution monitoring. For a plan to be executed in the real world the planner needs to reason about what changes an action should bring about and compare these with information returning from the real world. Given this information and an internal world model, the planner should be able to decide whether the execution of the action did actually bring about the desired result. However, the information which returns from the real world is usually inexact and incomplete and thus existing numerical methods of simulation cannot be used to make the decision.

When generating the operator schemas of the planner, the designer can outline the changes which the action will bring about in plan terms, e.g. the action 'open door' brings about the effect 'door open'. However, the changes brought about as side effects of the action at execution time cannot be outlined in advance. For example, if we execute the plan action 'open valve 1', then a fluid flow will result, but only if the conditions are correct. Including checks for such side effects at plan time is impossible as it would create massive precondition lists for each action. A further complication is where the side effect of a process is required in a plan. For example, in making a cup of coffee, the side effect of boiling the water is that it is hot and it is this which is required in the plan and not merely the occurrence of the boiling process itself.

In the world in which the intelligent agent can be placed, there exist various kinds of change. The objects which the agent can use can move, collide, slide, etc. and other objects in the world may flow, bend, heat up, cool, stretch, compress or boil. These and other things which cause continuous change in objects over time are usually referred to as processes. As the object goes through changes so the way in

which we view it changes. For example, if we boil a can of liquid we obtain a gas, and our description of the contents changes from 'liquid' to 'gas'. Such a way of looking at an object is referred to as its view. The way in which we describe change in the world around us tends to be in a vocabulary which is qualitative and not quantitative. For example, we use descriptions such as large, small or tiny, or specify changes as increasing and decreasing, which are all in themselves partial and vague terms. By using these simple descriptions, we can construct patterns of behaviour for objects which are quite complex. For example, if I heat a container of water which is sealed, the result may be that the container will explode. This description of behaviour ignores all numbers and merely relies on the influences present: the heat causes the temperature of the water to rise, which may cause it to boil, and the boiling within the sealed container causes a pressure increase, which may cause the container to explode. This behaviour is not the only one which can arise, so that the container is not guaranteed to explode. The various outcomes through which a process may evolve can be defined in terms of a tree, which has the processes and views active at certain times as the nodes, and the conditions between quantities at which processes appear and disappear as the arcs. The problems facing a system monitoring such a situation are how to create an internal representation of the real world on which it can experiment, and how to interpret the changes which result from executing an action. It is the function of the Process Reasoner to analyse the effects and side effects of plan actions and other processes in the world, and to warn of any problems which may occur. For example, if we open a valve between two containers, a fluid flow may result. The flow will not arise if the pressure in both the containers is the same, or one of the containers is empty, or the pipe has more than one valve which may be closed. By analysing the preconditions of the fluid flow example we can see that they break down into three categories. Firstly, those involving individuals in the world, e.g. a fluid flow can only exist between two contained liquids; secondly, quantity conditions, e.g. the flow will only happen if the pressure of one liquid is greater than that of the other; and thirdly, world

preconditions involving facts which are outside of the process itself, e.g. the valves of the pipe are open. This type of precondition classification can be applied to most situations involving processes. For example, a heat flow involves two individuals which have heat, the temperature of one must be greater than that of the other and the two objects must be aligned; a motion process involves a individual, the force on the individual must be greater than the frictional resistance of the individual and the individual to be moved must not be blocked by some other individual. Viewed in this way, a plan action can be seen to affect a process by meeting or deleting its preconditions. From the preconditions of defined processes and knowing the processes and views already active in a given context, the Process Reasoner can identify which processes will come into being and which will fail. This involves not only identifying those processes which will fail directly as a result of plan intervention but also those processes which will fail as a result. For example, if we close the valve of a flame which provides a heat flow to a boiling liquid, the the effect of the action is that the heat flow stops, and hence as a further effect the boiling process stops. The Process Reasoner needs to carry out changes to the process tree in the light of changes reported by the user (concerning the real world) and by the TNMS. As outlined above, the processes and views can fail naturally, because of changes in quantities brought about by the influences exerted by the process. For example, a fluid flow stops when the pressure of the liquid in the two containers is the same, or when one of the containers drains completely.

The qualitative reasoning systems which have been implemented to date have several shortcomings:

1. They treat the world as entirely isolated and hence self-determined, i.e. no external events are dealt with. Thus the only way processes or views can change is by being influenced by another process or view. This makes their reasoning inadequate, as a planning system requires to reason about changes

to a system brought about by its own intervention as well as those changes brought about by processes.

2. They do not have the ability to synchronize the actual outcome of a process with the real world. This means that the number of futures which the qualitative reasoner must analyse are greatly more than can actually occur since many can be ruled out by examination of real data in the world model. For example, the reasoning system may decide that a gas generated by a boiling process may provide the source of a flow process. However, even though the individual and quantity preconditions of the flow process may be satisfied, the world precondition may not and the process can thus be ignored.

3. None of the existing systems has the ability to follow the progress of the real world they are modelling and thus spot any deviation from the reasoned outcome. This means once an outcome has been generated, it cannot be updated by new processes as the assumptions allow no outside intervention. This is a drawback as a planning system needs to check the effects of a plan action executed at various points in time.

The following sections aim to describe the Process Reasoner, which aids a planning system to carry out the types of activity outlined above, including a description of the methods used by the Process Reasoner to create the process tree, and then synchronize it with data from the real world.

## 7.2 Creating the Process Tree

The processes and views which are reasoned to occur in the real world are stored in the process tree. The process tree contains those processes which occur as a result

of other processes as well as those which are brought about by plan effects. The nodes of the tree are unbounded time intervals over which the processes, views and influences of the node are active. The process tree is merely a qualitative ordering, as the duration of any node is unknown when the tree is generated. As actual information arrives from the real world, time intervals can then be assigned. In most examples, the structure takes on a tree shape; if there is feed-back in the system, the nodes can forms loops in which case the tree becomes a graph. However, during this discussion it will be referred to as the process 'tree'.

Processes and views become active and fail in the real world because their preconditions are met and then invalidated. As described in the introduction to this chapter, the preconditions can be broken down into three types. By analysing the different types of precondition, it becomes obvious that the quantity preconditions are not the same as the other two. The precondition which defines the individuals required in the process and the preconditions required from outside the process are satisfied by finding predicates which are true over the required time. However the quantity preconditions are not static predicates, but can vary continuously. For example, the quantity condition may be 'A > B' and both the quantities A and B may be increasing, which means that the test is different each time we make it.

While the process is active, it causes a variety of effects. It may cause a new individual to come into being, e.g. a boiling process will cause a gas to be created, or it may cause a new quantity to be created, e.g. the motion process will cause the quantity 'velocity' to be created so long as the object is in motion. However, by far the most important effects of a process are the influences which it asserts on quantities in the world, causing them to increase as well as decrease. For example, a fluid flow process effects two quantities, the amount of the source and the amount of the destination. The former will decrease and the latter will increase. These changes in relationship between quantities in the world may cause new processes to appear and old ones to terminate. For example, the quantity condition for a fluid flow is source pressure > destination pressure. Obviously the process will stop at some

point because the pressure difference is reducing due to the influences on the source and the destination. This may seem to solve the problem of process evolution. However, if we look at the problem further we will see there are secondary or indirect influences active as well. We know, for example, that if the amount of the source decreases then so will the pressure of the source. These indirect or qualitative influences can effect a quantity positively (Q+), negatively (Q-), or in an unknown way (Q#). For example, assuming that the source is a contained liquid "pressure source Q+ level source" means the pressure of the source will increase as the level increases. Thus by examining the direct and indirect influences on an quantity or object the Process Reasoner can decide what changes a quantity will exhibit i.e. increase, decrease or unknown. The change can be unknown because in the absence of any quantitative data, a quantity which is influenced both positively and negatively will have an unknown result.

Having decided how a quantity can change, the Process Reasoner needs to be able to analyse what effect this will have on the processes and views active in the system. This analysis is performed on the quantity spaces which exist in a given state of the process tree. A quantity space gives us a partial ordering of 'states' for a given quantity, and there is a quantity space for each quantity of an individual. The states are ordered qualitatively in size from left to right. For example the temperature quantity space for a liquid would have the states: **melt_point**, **current_temperature** and **boil_point**. If however we have a gas, then the quantity space for its temperature would be: **melt_point**, **boil_point** and **current_temperature.** When a process is active the quantity precondition may cause the the quantity space of two individuals to be joined together. Figure 7.1 shows the quantity space for the liquid stuff_1 during a heat flow process, where stuff_1 is the destination of the heat flow.

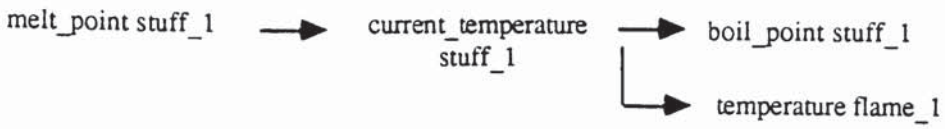



Figure 7.1

When a heat flow is active, the temperature of the flame which is the source of the heat flow (call it flame_1) can be added to the quantity space of stuff_1. As it is the source of the heat flow, the temperature of flame_1 must be greater than the current_temperature of stuff_1, and is thus placed to the right of it in the quantity space. However, no information is provided on the relationship between the temperature of flame_1 and the boil point of stuff_1 so they cannot be ordered. In a case such as this the quantity space branches and gives a diagram as in Figure 7.1.

The analysis of the influences on a quantity gives us the direction of movement in the quantity space. The quantities which are adjacent left and right to the influenced quantity are called neighbours. These are limit points and it is at these points that changes occur in processes and views, because the relationship between quantities change and thus can invalidate or create the quantity conditions required by a process. Taking Figure 7.1 as an example, the current_temperature of stuff_1 will increase due to the influence provided by the heat flow and as a consequence will move to the right. It will eventually reach either the boil_point of stuff_1 or the temperature of flame_1. Each of these outcomes needs to be analysed so that any process failures or creations can be found. In this case there are two possible outcomes, namely that stuff_1 will boil or the heat flow will stop, as there is no temperature difference between stuff_1 and flame_1. The assumption that stuff_1 will boil also relies on the fact that it is also a contained liquid. The creation of a boiling process will add further direct and indirect influences to the problem and as a result the Process Reasoner will need to analysis this new state. If a process is deleted from a state and the state still contains active processes then this needs to be analysed as well. The Process Reasoner will carry on with this analysis until all the states entered are final ones, in that all processes fail, or a more drastic situation is

encountered such as an explosion which destroys all the individuals of a given process state. As a further example, the behaviour of water heated in a sealed container is outlined in Figure 7.2.

The arrows in Figure 7.2 show transitions between the qualitative states and the limit hypothesis (L.H.) shows the quantity relationship required to move into the state. The Process Structure field (P.S.) shows the processes active in the qualitative state and the Individual Structure field (I.S.) shows the individuals which are involved in the processes of the P.S. field. As the evolution proceeds, various processes appear and disappear and the evolution finishes where a state contains no active processes, i.e. {}, or the individuals are terminated, e.g {explosion}.

The description outlined here is greatly simplified and does not outline the problems of circularity, variable binding and influence duplication which the actual Process Reasoner has to deal with.

Alternative behaviours for the Boiler by repeated Limit Analysis
PS: (heat flow)
LH: (A [temperature (WATER)] = A [temperature (SOURCE)]
IS: (CAN ,WATER)
PS: ( )
LH: A [temperature(WATER)] = A [t-boil(WATER)]
IS: (CAN,WATER,STEAM)
PS: (heat-flow, boiling)
LH: A [amount-of(WATER)] = ZERO
IS: (CAN, STEAM)
PS: (heat-flow)
LH: A [temperature(STEAM)] = A [temperature(SOURCE)]
IS: (CAN, STEAM)
PS ( )
LH: A [temperature(WATER)] < A [t-boil(WATER)]
IS: (CAN, STEAM, WATER)
PS: (heat-flow)
LH: A [pressure(CAN)]
= A [p-burst(CAN)]
IS: (CAN, STEAM)
PS: (Explosion)
LH: A [temperature(WATER)] = A [temperature(SOURCE)]
IS: (CAN, WATER, STEAM)
PS: ( )
LH: A [pressure(CAN)] = A [p-burst(CAN)]
IS: (CAN, WATER, STEAM)
PS: (Eplosion)

Figure 7.2

## 7.3 Representation of Processes and Views

The knowledge about processes and views in the world needs to be accessible to both the TNMS and the Process Reasoner. The TNMS uses the knowledge to find active processes in the knowledge base and the Process Reasoner uses it to reason about evolution and changes in the process tree. The preconditions and effects of a process are collected together to form a process schema which the Process Reasoner can use as well as the TNMS search routines. An example of a schema required for the heat flow process is described in Figure 7.3. The three types of precondition field, as described earlier, outline the facts necessary for the process to be active. The effects of a process can be broken down into two types: a set of relations and a set of influences. The relation fields outline those individuals, quantities, and indirect influences which are active while the process itself is active. The influence fields outline the quantities which the process affects directly while it is active. An influence field describes the type of influence one quantity has over another. It consists of a type indicator, + or - for positive or negative influence respectively, and the two quantities involved. The first quantity is the one asserting the influence and the second is the quantity the influence acts upon. For example, in a fluid flow the amount of the source will influence the amount of the destination to increase, which gives (I + amount_of source amount_of destination). The final field or assertion field of the schema indicates which individuals in the instantiation of the preconditions are involved in the assertion of the process. For example, a fluid flow between two contained liquids via a pipe will involve all of the individuals used to instantiate the preconditions of the process, i.e. the source, the destination and the path, which gives fluid_flow (stuff_1 stuff_2 pipe_1). However the process 'boiling' occurs within a container, but the container itself is not mentioned in the individuals asserted with the process, e.g. boiling (stuff_1). The assertion field tells the Process Reasoner which individuals are to be associated with the process when asserting that the process is active.

The TNMS search routines and the Process Reasoner use the schemas to instantiate new processes. To instantiate a schema, a valid set of individuals must be found in the knowledge base or process tree respectively. The numbers within the schemas provide an index to the variables bound in the instantiation of a precondition. The rules for variable binding are as follows:

* These variables are global to the entire schema and can be referenced in any field of the schema. The first occurrence of a * variable allows it to bind to any value; however once a value has been bound to the variable it keeps it for the entire scope of the schema. Where there are multiple matches to an unbound * variable all combinations are covered by the Process Reasoner or TNMS search routines. For example,

[*1 a contained liquid] [process heat flow *2 *1 +1]

This finds all objects which are contained liquids and also the destination of a heat flow process. From now on any mention of the variables *1 and *2 within the schema will cause the values bound here to be substituted in.

\+ These variables carry a value which is local only to the current field, e.g. one of the precondition fields, the relation field or the influence field. As in the above case the first occurrence can bind to any value but if it is mentioned later within the same field the bound value is used.

The quantity spaces with which the Process Reasoner carries out its reasoning change during the course of the process evolution. As described earlier in Figure 7.1 the quantity space gained a new individual when a heat flow into the liquid was found. Not only do new individuals get added to a quantity space, but the values within a quantity space change their relationships. For example, if the liquid in Figure 7.3 boils then the relationship that the temperature is less than the boil point is no longer true. These types of change do not need to be represented explicitly in a

schema but can be found implicitly within the quantity preconditions of the process. For example, the quantity condition for a boiling process is 'temperature A > boil point A' so the Process Reasoner can update the temperature quantity space in the qualitative state which reflects a boiling process. When a new qualitative state is generated, only the changed quantity spaces are copied into it; any other quantity spaces are inherited from the parent

```
[boiling    [[*1 a contained liquid][process heat flow *2 *1 +1]]
            [[temperature *1 > boil point *1]]
            [[]]
            [[view definition [[container *1 +1 ][not open +1]]
                              [[[contained gas £1][[pressure *1 Q+
                                                   pressure £1]]]
                              [[gas £1] [] ]]]
            [ I + heat *1 flowrate *2]
            [ I - amount_of *1 generation_rate £1]
            [ I + amount_of £1 generation_rate £1]
            [ I - heat *1 generation_rate £1]
            [ I + heat £1 generation_rate £1]
            [ temperature *1 = temperature £1]
            [substance *1 = substance £1]]
            [boiling *1]] ]
```

Figure 7.3

The relations indicate those facts and individuals active during the time the process is active. The variables which need to be bound in the relations can nearly all be found from the variables bound in the precondition search. For example, within a fluid flow process, the amount of the source will cause the amount of the destination

to increase i.e. (I + amount_of *1 amount_of *2). When we use the variables bound in the preconditions we obtain (I + amount_of stuff_1 amount_of stuff_2) assuming the fluid flow is from stuff_1 to stuff_2. However, the problem is made more difficult when a process creates a new individual as an effect. A process like melting, solidifying, boiling, etc. will create a new individual. For example, in the boiling process, a gas will be generated and depending on the condition of other individuals this may have different effects. If the gas is generated inside a sealed container, it will give rise to one set of influences, and if generated in an open container, a different set of influences. The view definition indicates to the Process Reasoner to which view the new individual belongs and also any indirect influences to be added which are not in the schema definition. It operates like an if...then...else rule, the first part being the conditional test, the second the view if the test is true and the third the view if the test is false. In the example in Figure 7.3, the test requires finding the container of the first individual (i.e. the contained liquid) and then finding out if the container is not open. The use of a view definition avoids the Process Reasoner having to scan all the schemas and rules (as in all of the current systems) to find the required information, when the context in which the process is active gives us the information required. Any new individuals are denoted by a £ symbol and an index number denoting which new individual it is. For example, in a boiling process the amount of the gas will influence the amount of the liquid to decrease which gives us

[ I - amount_of *1 amount_of £1]

which will be instantiated to

[ I - amount_of stuff_1 amount_of gas_1]

assuming that stuff_1 is the liquid which is boiling.

The outcome of a process can not only create new individuals, but destroy existing individuals. For example, a process causing a pressure increase or decrease in a sealed container may result in the container exploding or imploding respectively. The Process Reasoner is informed of the loss of an individual by the schema

containing a **terminate** directive. This will cause the Process Reasoner to remove the individual from the current state in the process tree and to terminate any process which involves the destroyed individual. This may cause all the active processes within a given state to fail if the individual is mentioned implicitly or explicitly in their list of individuals. For example, in the case of the heat flow to a sealed can of water, if the can explodes then the boiling and the heat flow both stop. If however we consider two tanks containing liquid which are connected together and both the liquids are boiling, then if one explodes, the boiling process in the second tank will continue uninterrupted if the valves between the tanks are closed and no fluid flow is currently active between them.

## 7.5 Synchronization of the Process Data

The synchronization process is required by the PMS to align its world model with the real world and is a major component of the Excalibur system. To date, planners have treated changes as either all being known or as occurring over a zero duration so that the problem never arises. However when we attempt to deal with planning occurring over time, we require a world model which is not only logically consistent but also a true reflection of the real world. Excalibur has the ability to check progress through a process tree and to react to any changes which are contrary to those expected. The process tree which the Process Reasoner creates contains the possible outcomes which an initial process might evolve through. The process tree will contain all possible outcomes from this initial state, but the real world will create a single path through the tree as it evolves. Part of the Process Reasoner's function is to align itself with the correct path in order to synchronize its internal model with the external world. In order to accomplish this, there exists a synchronization module which takes information from the real world and attempts the alignment. The information from the current world may indicate that the path is correct, or it may reveal a new path through the tree, or it may cause the process tree to be altered as

new processes evolve which were unexpected. The final case is the most complex, as it may occur because of changes in processes brought about by plan actions or from a quantity whose influences could not be resolved when the process tree was created. In Section 7.2, it was noted that a quantity which was influenced both positively and negatively would not have a known outcome when the process tree was created. At a later date this situation will be revealed, and consequently the change brought about might cause a new process to run or a process to fail. For example, if we were filling a tank with cold water while at the same time heating the tank then the effect on the temperature will be unknown. When the actual numbers are known or a relationship is asserted then the influences can be resolved. For example, if the temperature increases the water will either become warmer at which point the heat flow may stop or it may boil and if it decreases then the heat flow will carry on as predicted.

The synchronization process receives data from the real world about various changes which are taking place or will take place. The types of information it can analyse are:

1. Changes in the spatial position of objects.
2. The relationship between two quantities.
3. The assertion of the occurrence or ending of a process in the real world.
4. The current direction of change of a quantity i.e. increasing, decreasing, stable.

Each of these inform the synchronization module of a possible change to the process tree. The synchronizer first examines the states of the process tree and tries to match the data to its expectations. If timing information is available on future states then these can be checked against the timing information in the input data. For example, the TNMS or the user may indicate that a movement process involving a crate will occur in five minutes, hence the state in which the crate is involved in a motion process can be temporally defined. In the case where no timing data is available, it starts with the current state as its initial point. In the case of a process which is

asserted by the user to happen or end at a given point, the synchronization module tries to match the process and its individuals to the process tree. The process may be expected or it may be a deviation from expected behaviour. However, the information about a process may not prove conclusive as a process which disappears may do so for various reasons. If we take as an example water flowing between two tanks as in Figure 7.5, then if the synchronization process is told that the fluid flow has stopped, the next state is still unknown as both involve a loss of the fluid flow process. We could arbitrarily pick one, as they both reflect the state in which the process is missing. However, if we analyse them, we find they do have the same outcome, but the worlds they represent are not the same. In one state, we have two tanks of liquid whose pressure is equal whereas in the second, one tank is empty. Under such circumstances, the synchronization process cannot make a certain commitment as to which state the real world has entered, so it prompts the user with the possible reasons why the process failed and the user then indicates the reason for failure. The user is prompted at this point because in the present system there are no real sensors which can be checked to provide the missing information. This unknown next state problem only arises when a process fails, as the state to which a newly active process is assigned is unique.

The information asserting a relationship or a change in quantity provides more information. Within the process tree are kept the quantity relations necessary for a change of state in the process tree. The relationship is compared against the quantity relation for each of the next states from the current state. If a match is found then the synchronization process checks to see if any of the processes in the new state are known, i.e. whether a process which should disappear in the next state been noted as such or whether a new process is active. This might arise if the process changes are noted before the relationship is input. If not, then for each process in the new state a message is set to the TNMS to request it to inform the Process Reasoner should the process change required show up within the specified time period. If the process is seen within the time window, or the processes have already been seen, then the

synchronization process will move the current state to the new state. If the process is not seen within the time out window, the synchronization process attempts to check if any of the effects have been seen e.g. "if we cannot tell if the liquid is boiling, is there a pressure increase to indicate so?". If none of the effects can be seen, the synchronization process choses to ignore the fact as a faulty input.

When analysing a quantity change, the synchronization process must check if the direction of change is correct. If the expectation is that a quantity will increase and it begins to decrease then there is a problem in the internal model. The Process Reasoner keeps the quantity changes for each state in the process tree and can easily find out if the direction of change is correct. If the direction of change is correct then the synchronization process takes no action; however if the change is incorrect, then it raises an error for the Process Reasoner to check.

The majority of changes to the process tree are caused by the effects of plan actions. The effects cause changes in the precondition field of a process and so can usually be identified. The changes will either effect the process directly or indirectly. The indirect effect would be via a rule which fills the precondition slot of the process, which has as a precondition the negative of the plan effect. For example, the precondition to a fluid flow is that the path is aligned. This is posted via a rule which has as its precondition that any valve the pipe may have is open. If the effect of the plan action is to close one of the valves, then the pipe will no longer be aligned and the process will fail prematurely. Under such circumstances the process tree is amended and the facts in the knowledge base are automatically truncated by the truth maintenance module of the TNMS. A process which is brought about by the plan action will be reported by the TNMS if the schema can be instantiated and if not then will be picked up by the synchronization process from data reported by the user.

The synchronization process will continue to match changes until the real world enters a state in which no processes are active. The Process Reasoner then marks the individuals of the state as dormant and marks the final state into which the process has evolved.

## 7.6 Error Recovery and Analysis

Once a faulty process or quantity change has been identified the Process Reasoner must try and analyse the world model in order to explain the unexpected behaviour. This may be the current world or some future time interval. In the case of a quantity change, the Process Reasoner must first identify if the change is indeed an error or is a quantity whose influences could not be resolved when the process tree was created. In the later case the Process Reasoner adds in the new influences to the relevant state and re-analyses the process tree, which may cause new branches to be added or existing branches to be deleted. The Process Reasoner must also send information to the TNMS to truncate some facts and to remove others from the TNMS completely. This allows the Process Reasoner to react to changes in the world as more information on a quantity becomes available. The state against which a fact is compared in the process tree is dependent on the information already supplied. As the states of the process tree are unbounded, the occurrence of a fact cannot be definitely defined unless the data is unique to one state or the information is asserted relative to an already asserted piece of information. Any fact which cannot be added as the state is not unique is held in a suspended list until the information required to find a unique position is known.

If the quantity change is indeed an error, then the Process Reasoner must first find which influences it thought were active are not now active, and which influences could be active to cause the problem. In the first case a process may have failed in the real world and this was not reported to the Process Reasoner. In the second case, a process may be active because its preconditions are true in the real world and these facts have not been reported to the Process Reasoner. The way in which a process can arise or fail is due to its preconditions being met or deleted respectively. In the case of a process which is suspected to have failed, the Process Reasoner must analyse only those which contribute influences to the quantity at the point of failure. For example if we are analysing a decrease in fluid flow we do not examine processes involving motion. If there are no processes active at the point of failure

then the influence must be asserted via an unknown process. If there are processes active then these are candidates for having failed. The preconditions of each of the processes is instantiated and each of the preconditions is displayed to the user. The user then replies as to the continued persistence of the fact or that it has indeed failed. As in the above case the user is prompted because there are no real sensors in the current system. The facts are then altered according to the user's input. If the user cannot suggest a rogue process, then the Process Reasoner assumes the influence is coming from another process. This could be an influence from within the current reasoning context or from another reasoning context. For example, suppose two tanks are initially in separate reasoning contexts, and an unexpected temperature increase occurs in one of them. The Process Reasoner will look for a heat flow e.g. the other tank may contain a hot liquid and there may be a heat flow between them. In general the Process Reasoner needs to find all processes which are active in adjacent reasoning contexts which provide an influence akin to the one we are trying to explain. If the process can be found then the user is prompted with each of its preconditions in turn to indicate whether this is a possible source of the problem. If no process can be identified then the Process Reasoner suggests the type of process which may be involved and the user is required to provide further analysis. If there is an effect of one reasoning context on another then the various data structures which make up the contexts are collapsed into one. This is relatively simple with the structures of the TNMS, but in the case of the Process Reasoner, the two process trees will need to be collapsed and re-analysed.

In the case of a process change, the analysis required is quite similar. The process in error, whether because it failed prematurely or because its occurrence was unexpected, is controlled by its preconditions, which are either true or untrue. In the case of a process which becomes true the preconditions can be asserted as active in the TNMS and the TNMS will set up the dependency and clipping automatically. When the TNMS reports finding the process it will be added to the process tree and the tree re-analysed. A process which fails has similar reasoning requirements and

the user is required to indicate which of the process preconditions have failed. This will be asserted in the TNMS which in turn will report its failure to the Process Reasoner. For example, if in a cooling plant there was an heat increase then the Process Reasoner would suggest as possible causes both a heat flow and a fluid flow process, as they can both affect the temperature of a substance.

This method of analysis, i.e. using a fixed hierarchy of methods to find a solution to a problem, seems to follow what humans would do in the same situation, i.e. check the simplest things or most usual possibilities first, then check for external influences and finally perform a blanket search of all information. The order in which we carry out such analysis is usually fixed. We first analyse those things which most commonly cause errors, e.g. a fault due to non-reporting of information. If this fails then we try to find events in our reasoning context and beyond that in adjacent contexts. Only as a last resort do we fall back on more weaker methods such as a blanket search with all information.

The Process Reasoner has a simple explanation facility to allow a fixed set of questions to be answered. The questions which it can answer include: what changes there are from a given state, the processes active in a given state, the influences active in a given state and finally why a given state exists. From these the user is able to find out the history through which the process is expected to evolve and the changes brought about in the process tree by plan actions.

# Chapter 8: Discussion of Results

## 8.1 Main Goals and Methods of Experiment

Previous planning systems have concentrated mainly on creating efficient and correct plans in response to a user request. However, these plans will only execute correctly if the changes in the world are as expected in the plan, and no changes occur which interfere with the plan. Such planners are capable of producing different solutions to a problem should they exist (i.e. replanning the whole plan) but are usually incapable of replanning only part of the plan. To achieve this, the planner must be able to analyse the cause of any plan failure and the actions which are affected, as these form the context for any partial replan which may be required. Changes in a plan can also be introduced to allow the plan to interact with events occurring in the world. Previous systems consider the preconditions of plans to be independent of events in the world and do not address the problem of actions having preconditions dependent on events occurring in the world whose completion is unknown to the planner.

The main goals of this research are:

1. To create a representation of planning which allows the plan to take into consideration preconditions provided by external events not brought about directly by one or more plan actions.
2. To create a planning system capable of monitoring the execution of a plan and overcoming any problems which may occur by re-scheduling parts of the plan or by adding new actions to the plan.

To this end, the research has been structured into three main phases. The first phase was to design and validate a planning world model based on Process Theory and to reproduce the results obtained by Forbus (1983), and also to examine more of the underlying problems caused by trying to monitor the execution of a plan with an

internal world model. The second phase involved the creation of a planning system capable of using schema which allow the actions of the plan to interact with the world and then allow the plan to be modified if required due to plan failures. The third phase involved testing the parts on realistic planning problems which so far have proved difficult or impossible with current planning systems.

The examples described in this Chapter show the system's ability to solve problems involving processes, which was the main aim of the research. However, the system has also been tested with analysis and replanning problems which can be handled by the existing Nonlin system such as 'block moving' and 'house building'. These were handled successfully but unfortunately due to space constraints cannot be reported here.

The following sections in this Chapter provides details of the procedures used to test the correctness of the system. Each of the examples details the plan and process schemas used. Output collected during a session with system attempts to show how the system reacts to various situations, as well as to illustrate the techniques put forward. Space unfortunately prevents the inclusion of the full output for each example.

## 8.2 Examples of Process and Planning Reasoning

The following sections aim to show how the system described in this thesis is capable of carrying out planning and execution monitoring in planning involving processes. The examples show that even in simple problems a great deal of extra reasoning outside of making sure all plan preconditions are met is required for the plan to succeed.

### 8.2.1 Making a cup of coffee

During the course of this thesis, the example 'making a cup of coffee' has been used to demonstrate that even a simple problem such as this requires complex

reasoning for it to succeed. This section outlines how the task of making a cup of coffee can be handled by the system and completed successfully. The plan schemas to achieve this task are the ones outlined in Figure 6.8. The process schemas which will be used by the system are as follows.

1. View liquid.
2. View contained liquid.
3. View gas.
4 View open/closed container.
5. Process Fluid_flow between two objects which are aligned.
6. Process Heat_flow between two objects which are adjacent via a path.
7. Process Boiling.

The world in which the plan is to be executed starts with five individuals. These are : -

1. tap_1, from which the water to make the coffee is obtained.
2. stuff_4, which is the actual water inside tap_1.
3. kettle_1, in which the water is to be boiled.
4. cup, in which we make the coffee.
5. gas_1, which is used to heat kettle_1.

During the course of execution of the plan other individuals will be created and/or destroyed. For example, there will be a new individual created when the water from the kettle is poured into the cup and when steam is generated while boiling the kettle. The description of the world which we have before the plan is requested is as in Figure 8.1.

```
[can contain tap_1 stuff_4]
  [fpath a fluid path]
  [hpath a heat path]
  [tap_1 closed]


  [[temperature stuff_4][80 1 0 0]]
  [[temperature gas_1][1000 1 0 0]]
  [[amount_of_in tap_1][10 1 0 0]]
  [[boil_point stuff_4][100 1 0 0]]
```

Figure 8.1.

The plan to make the cup of coffee consists in essence of :-

1. Filling kettle_1 from tap_1
2. Taking kettle_1 to gas_1 and then lighting gas_1.
3. Waiting until kettle_1 has boiled and then placing some coffee in the cup.
4. Finally filling the cup to the required level by pouring the water from kettle_1 into the cup.

The plan generated by Nonlin is described in Figure 8.2. (Note that the numbering is Nonlin's and does not correspond to the ordering of the actions.) The rest of the Section shows the solution of the problem by the system with annotated notes to explain the reasoning the system is carrying out.

The system receives the plan from Nonlin and tries to assign time slots to the first 10 actions of the plan. However the eighth action of the plan is a 'waitfor' and thus the system has no way of knowing when the actions which follow this 'waitfor' can begin. So at this point the plan is suspended and no further actions are sent to the TNMS. This results in the state outlined in Figure 8.3.

4
Go to the kitchen
5
Grasp Kettle
6
Pickup kettle
7
Take kettle to tap
8
Put kettle under tap
16
Grasp tap
17
Turn on tap
18
Waitfor {level contents kettle = fill level kettle
1
1
19
Turn off tap
20
Ungrasp tap
21
Take kettle to gas
14
Put down kettle on gas
9
Ungrasp kettle
10
Turn on gas
11
Waitbegin {boiling contents kettle}
12
Turn off gas
2
2
13
Place coffee in cup
22
grasp kettle
23
pickup kettle
24
take kettle to cup
25
pour contents of kettle intocup
26
Waitfor {level contents cup = fill_level cup
15
Put kettle back on gas

Figure 8.2.

```
The current time is now 0 mins 3 seconds


suspended :
plan number 1 has the goal
  {make a cup of coffee kettle_1 cup}


The actions executed so far are
        No actions have yet been executed


        No action is currently executing


The actions sent to the loader are
        [[go to the kitchen] 4 1]
        [[grasp kettle_1] 5 1]
        [[pick up kettle_1] 6 1]
        [[take kettle_1 to tap_1] 7 1]
        [[put kettle_1 under tap_1] 8 1]
        [[grasp tap_1] 16 1]
        [[turn on tap_1] 17 1]
        [[waitfor {level contents kettle_1 = fill level
                   kettle_1}]18 1]


The actions awaiting to be sent to the loader are
        [{turn off tap_1} 19 1]
        [{ungrasp tap_1} 20 1]
        [{take kettle_1 to gas_1} 21 1]
        [{put down kettle_1 on gas_1} 14 1]
        [{ungrasp kettle_1} 9 1]
        [{turn on gas_1} 10 1]
        [{waitbegin {boiling contents kettle_1}} 11 1]
        [{turn off gas_1} 12 1]
        [{place coffee in cup} 13 1]
        [{grasp kettle_1} 22 1]
        [{pick up kettle_1} 23 1]
        [{take kettle_1 to cup} 24 1]
        [{pour contents of kettle_1 into cup} 25 1]
        [{waitfor {level contents cup = fill level cup}} 26
         1]
        [{put back kettle_1 on gas_1} 15 1]
```

Figure 8.3.

The effects of actions sent to the TNMS cause various preconditions to be met in process and view schemas. The kettle being placed under the tap causes a fluid alignment to occur, and by turning on the tap, a fluid connection is created and as a result a fluid flow begins. These are found by the TNMS and the appropriate facts and their justifications are created in the TNMS. When this is complete, messages are sent to the Process Reasoner to indicate these new processes are active so that their

consequences can be analysed. The messages indicate the processes involved and the individuals on which they act. As the Process Reasoner has no previous knowledge of any of these individuals being involved in a process, it creates a new process tree in order to analyse the effects of the fluid flow between the kettle and the tap. This analysis is described in Figure 8.4.

```
The current time is now 0 mins 4 seconds

** [The processes active in State 1]

** [process 1 involves a fluid_flow with individuals
    [stuff_4 individual1 fpath]]



** [The state changes active in State 1]

** [change 1 involves the relation [amount_of stuff_4 <
        zero] to state 2]
```

Figure 8.4.

The only outcome which the Process Reasoner can find is that the amount of water in the tap might drain away so causing a failure in the fluid flow process. The system then waits until a condition in the process tree is true or there is a change in the world which causes a change in the plan or process tree. After a wait of a few seconds the system is told, via some sensor input (simulated by the user) that the levels have equalised and this causes the Plan Reasoner to unsuspend the plan. This it does by sending the 'waitfor' message back to the loader to indicate the condition is now true. The loader then tries to send down another ten actions from the plan but again finds a suspension event, i.e. to wait until the 'kettle boils'. This results in the status of the plan in the TNMS as outlined in Figure 8.5.

```
The current time is now 1 mins 13 seconds


The plans known to the system are as follows


suspended :
plan number 1 has the goal
  {make a cup of coffee kettle_1 cup}


The actions executed so far are
        [[go to the kitchen] 4 1]
        [[grasp kettle_1] 5 1]
        [[pick up kettle_1] 6 1]
        [[take kettle_1 to tap_1] 7 1]
        [[put kettle_1 under tap_1] 8 1]
        [[grasp tap_1] 16 1]
        [[turn on tap_1] 17 1]
        [[waitfor {level contents kettle_1 = fill level
               kettle_1}]18 1]
        [[turn off tap_1] 19 1]
        [[ungrasp tap_1] 20 1]
        [[take kettle_1 to gas_1] 21 1]
        [[put down kettle_1 on gas_1] 14 1]
        [[ungrasp kettle_1] 9 1]
        [[turn on gas_1] 10 1]


The action currently executing is
        Waiting for a suspension event


The actions sent to the loader are


The actions awaiting to be sent to the loader are
        [[turn off gas_1] 12 1]
        [[place coffee in cup] 13 1]
        [[grasp kettle_1] 22 1]
        [[pick up kettle_1] 23 1]
        [{take kettle_1 to cup} 24 1]
        [{pour contents of kettle_1 into cup} 25 1]
        [{waitfor {level contents cup = fill level cup}} 26
         1]
        [{put back kettle_1 on gas_1} 15 1]
```

Figure 8.5.

Some of the actions which were sent down after the plan was unsuspended have effects which cause failures in the justifications of the processes outlined above. By closing the tap, we cause the fluid connection to fail, and by moving the kettle to the gas, we cause the fluid alignment to fail. These are both required as justifications of

the fluid flow, and as a result the fluid flow fails as well. The TNMS truncates the persistence of these processes to the point in time at which the justifications failed and sends a failure message for each of them to the Process Reasoner. Even though the process has failed, the effect which we required has been seen, so this process failure can be ignored as it causes no failure in the plan. By moving the kettle to the gas and lighting the gas, we have created three further processes. Firstly, a heat alignment between the gas and the water inside the kettle, secondly, a heat connection between the same individuals when the gas is turned on, and thirdly, a heat flow from the gas to the water via the kettle. Again, these are set up in the TNMS and messages are sent to the Process Reasoner about their creation. The Process Reasoner now adds this new process information to the tree already created. As the fluid flow has stopped information held within State 1 is true at the point at which the heat flow began, so the Process Reasoner choses this state as the point in the tree at which to analyse the effects of the heat flow. The predicted outcomes of the heat flow are that the water in kettle_1 will boil, in which case the water might boil away, or that the water will merely become warmer. Since these outcomes cannot be decided between without actual sensors, input from the user is required. The system was informed that the water in kettle_1 was boiling. The boiling of the water in kettle_1 was required as part of the plan to make a cup of coffee and now it has been achieved the plan is unsuspended. This causes more actions to be sent to the loader with the effect that the boiling stops and a fluid flow occurs between the water in kettle_1 and the cup. A new individual is created as the water is poured into the cup and the fluid flow exists until the water in the cup has reached the required fill level, at which point the kettle is put back on the gas, and the problem is finished. The complete process tree for this problem is outlined in Figure 8.7; the changes above involve States 4, 5 and 7 (Figure 8.6).

```
The current time is now 1 mins 22 seconds


 ** [The processes active in State 4]

 ** [process 1 involves a heat_flow with individuals
     [gas_1 individual1 kettle_1]]

 ** [process 2 involves a boiling with individuals
           [individual1]]



 ** [The state changes active in State 4]

 ** [change 1 involves the relation
    [amount_of individual1 < zero]
    to state 6]

 ** [change 2 involves the relation
    [temperature gas_1 < temperature individual1]
    to state 5]



 ** [The processes active in State 5]

 ** [process 1 involves a fluid_flow with individuals
     [individual1 individual7 fpath]]



 ** [The state changes active in State 5]

 ** [change 1 involves the relation
    [amount_of individual1 < zero]
    to state 7]


 ** [The processes active in State 7]

 ** [There are no processes active in this state]
```

Figure 8.6.

The Process Tree for the coffee making example
1
PS : heat flow
IS : Gas_1 Individual1 Kettle_1
3
LH : Temperature Gas_1 < Temperature Individual1
IS: Gas_1 Individual1 Kettle_1
PS {}
4
LH : Boil point Individual1 = Temperature Individual1
IS: Gas_1 Individual1 Kettle_1
PS {heat flow boiling}
5
LH : Temperature Gas_1 < Temperature Individual1
IS: Individual1 Individual7 Fpath
PS {fluid flow}
6
LH : Amount of Individual1 < ZERO
IS : Gas_1 Individual1 Fpath
PS : {}
7
LH : Amount of Individual1 < ZERO
IS : Individual1 Individual7 Fpath
PS : {}

Figure 8.7.

### 8.2.2 Avoiding an unforeseen circumstance

The example outlined above (in Section 8.2.1) shows how the system can create and successfully execute a plan to make a cup of coffee. However, as described earlier in Chapter 5, the world of a planner is constantly changing so events could occur which may interfere with its plans. The next two examples demonstrate the system's ability to analyse failures in a plan and then to replan to recover from the error. In the first example, the system is told that the kettle has emptied while it is waiting for the level of the cup to reach the required fill level. This means the system will have to reason about the requirements of the fluid flow process in order to solve the problem. This problem has several points of interest namely:

1. In order for the replan to be successful, it must boil the water before it pours it in the cup.
2. Several actions do not need to be repeated as they are already true at the point of failure. For example, in the original plan we had to go into the kitchen, grasp the kettle and pick it up before we took it to the tap. However all of these needed to be true in order to pour the water from the kettle into the cup, so are not needed when we re-fill the kettle from the tap.
3. Part of the original plan was to put coffee in the cup which the plan has already succeeded in doing, so this need not be repeated again. However even though the preconditions for picking up the kettle from gas_1 are still true we cannot remove these from the plan because otherwise the plan would contain no actions to allow us to pour the water into the cup.

A patch plan was created by the replanner as in Figure 8.8 and was successfully integrated into the plan network as can be seen in Figure 8.9. The structure of the process tree did not need to be altered as the patch was designed to bring about the filling of the cup. The process tree was moved to State 7 as in Figure 8.10 to reflect the fact that the amount of individual1 had become zero. The Process Reasoner had to realise that there were two changes from the original process tree. Firstly, as

individual1 now no longer exists a new individual had to be created when the water was poured into the kettle, and secondly that the cup now contained a liquid and thus the fluid flow process did not need to create a new individual as it did before. This it successfully did and the plan was solved correctly.

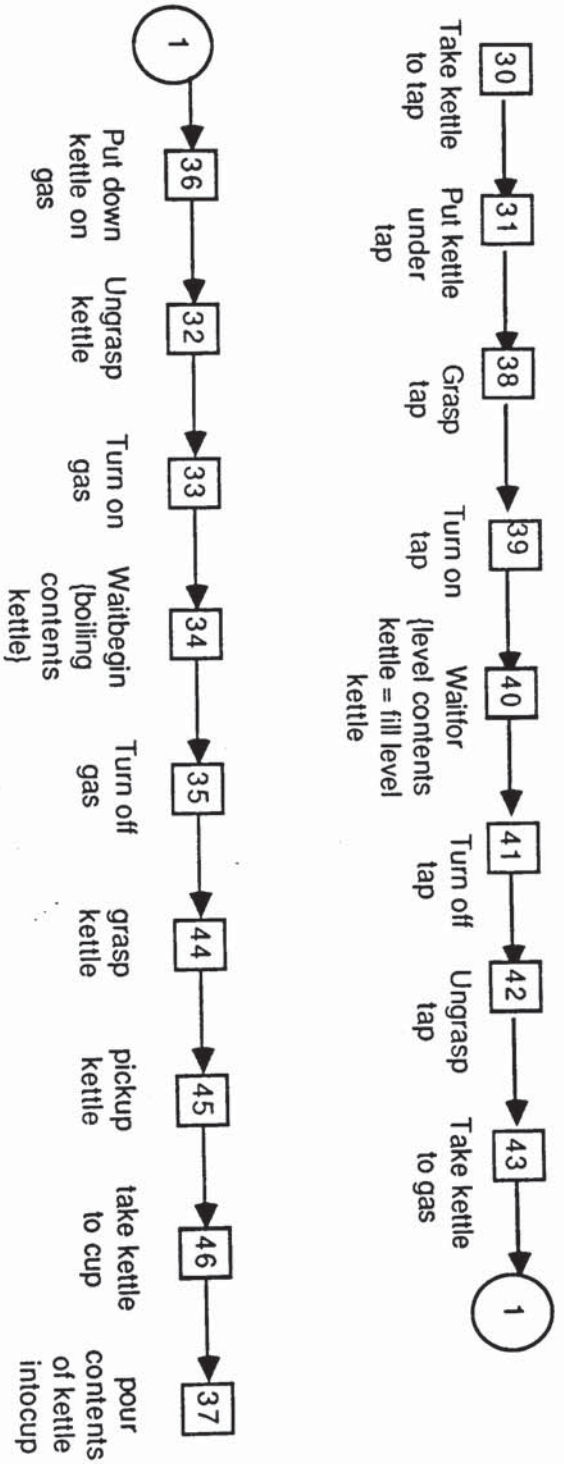



Figure 8.8.

```
The current time is now 2 mins 0 seconds


 The plans known to the system are as follows

 plan number 1 has the goal
   {make a cup of coffee kettle_1 cup}

 The actions executed so far are
         [[go to the kitchen] 4 1]
         [[grasp kettle_1] 5 1]
         [[pick up kettle_1] 6 1]
         [[take kettle_1 to tap_1] 7 1]
         [[put kettle_1 under tap_1] 8 1]
         [[grasp tap_1] 16 1]
         [[turn on tap_1] 17 1]
         [[waitfor {level contents kettle_1 = fill level
                          kettle_1}]18 1]
         [[turn off tap_1] 19 1]
         [[ungrasp tap_1] 20 1]
         [[take kettle_1 to gas_1] 21 1]
         [[put down kettle_1 on gas_1] 14 1]
         [[ungrasp kettle_1] 9 1]
         [[turn on gas_1] 10 1]
         [[waitbegin {boiling contents kettle_1}] 11 1]
         [[turn off gas_1] 12 1]
         [[place coffee in cup] 13 1]
         [[grasp kettle_1] 22 1]
         [[pick up kettle_1] 23 1]
         [[take kettle_1 to cup] 24 1]
         [[pour contents of kettle_1 into cup] 25 1]


        No action is currently executing


 The actions sent to the loader are


 The actions awaiting to be sent to the loader are
         [{take kettle_1 to tap_1} 30 1]
         [{put kettle_1 under tap_1} 31 1]
         [{grasp tap_1} 38 1]
         [{turn on tap_1} 39 1]
         [{waitfor {level contents kettle_1 = fill level
                          kettle_1}}40 1]
         [{turn off tap_1} 41 1]
         [{ungrasp tap_1} 42 1]
         [{take kettle_1 to gas_1} 43 1]
         [{put down kettle_1 on gas_1} 36 1]
         [{ungrasp kettle_1} 32 1]
         [{turn on gas_1} 33 1]
         [{waitbegin {boiling contents kettle_1}} 34 1]
         [{turn off gas_1} 35 1]
         [{grasp kettle_1} 44 1]
         [{pick up kettle_1} 45 1]
         [{take kettle_1 to cup} 46 1]
         [{pour contents of kettle_1 into cup} 37 1]
         [[waitfor [level contents cup = fill level cup]] 26
          1]
         [[put back kettle_1 on gas_1] 15 1]
```

Figure 8.9.

```
The current time is now 2 mins 0 seconds

** [The processes active in State 7]

** [There are no processes active in this state]

** [The reason that state 7 comes about is
    amount_of individual1 < zero]
```

Figure 8.10.

The second problem which the system was given was how to deal with the gas being turned out while it was waiting for the water in the kettle to boil. The gas being turned out causes the heat flow to stop and with it the boiling process, so there is a failure of the 'waitbegin' action on which the plan is suspended. The process tree again does not need to be altered, except that the Process Reasoner must realise that the message it receives from the TNMS about a heat flow from gas_1 to individual1 via kettle_1 is not a new process so that it can leave the process tree unaltered. The plan itself can be repaired by re-executing the light gas action and creating a new waitbegin action for the boiling process. The updated plan is outlined in Figure 8.11 and shows the state of the plan after the extra waitfor has been executed and the subsequent actions sent to the TNMS. The process tree is described in Figure 8.12 and shows the state of the system at the point where the boiling and heat flow processes have finished, but before the fluid flow into the cup has been identified.

```
The current time is now 1 mins 31 seconds


The plans known to the system are as follows


suspended :
plan number 1 has the goal
  {make a cup of coffee kettle_1 cup}

The actions executed so far are
        [[go to the kitchen] 4 1]
        [[grasp kettle_1] 5 1]
        [[pick up kettle_1] 6 1]
        [[take kettle_1 to tap_1] 7 1]
        [[put kettle_1 under tap_1] 8 1]
        [[grasp tap_1] 16 1]
        [[turn on tap_1] 17 1]
        [[waitfor {level contents kettle_1 = fill level
                          kettle_1}]18 1]
        [[turn off tap_1] 19 1]
        [[ungrasp tap_1] 20 1]
        [[take kettle_1 to gas_1] 21 1]
        [[put down kettle_1 on gas_1] 14 1]
        [[ungrasp kettle_1] 9 1]
        [[turn on gas_1] 10 1]
        [[waitbegin {boiling contents kettle_1}] 11 1]
        [[turn on gas_1] 27 1]
        [[waitbegin {boiling contents kettle_1}] 28
              1]
        [[turn off gas_1] 12 1]
        [[place coffee in cup] 13 1]
        [[grasp kettle_1] 22 1]
        [[pick up kettle_1] 23 1]


The action currently executing is
        [[take kettle_1 to cup] 24 1]


The actions sent to the loader are
        [[place coffee in cup] 13 1]
        [[grasp kettle_1] 22 1]
        [[pick up kettle_1] 23 1]
        [[take kettle_1 to cup] 24 1]
        [[pour contents of kettle_1 into cup] 25 1]
        [[waitfor [level contents cup = fill level cup]] 26
          1]


The actions awaiting to be sent to the loader are
        [[put back kettle_1 on gas_1] 15 1]
```

Figure 8.11.

```
The current time is now 1 mins 31 seconds

** [The processes active in State 9]

** [There are no processes active in this state]


** [The processes active in State 1]

** [process 1 involves a heat_flow with individuals
    [gas_1 individual1 kettle_1]]

** [The state changes active in State 1]

** [change 1 involves the relation
   [temperature individual1 > boil_point individual1]
   to state 8]

** [change 2 involves the relation
   [temperature gas_1 < temperature individual1]
   to state 7]

** [The processes active in State 8]

** [process 1 involves a heat_flow with individuals
    [gas_1 individual1 kettle_1]]

** [process 2 involves a boiling with individuals
        [individual1]]

** [The state changes active in State 8]

** [change 1 involves the relation
   [amount_of individual1 < zero]
   to state 10]

** [change 2 involves the relation
   [temperature gas_1 < temperature individual1]
   to state 9]
```

Figure 8.12.

### 8.2.3 Avoiding an undesirable circumstance

The following two examples show how the system can reason about a planning problem involving a small process plant. The individuals in this problem are displayed diagrammatically in Figure 8.13. The basic plan is to let water flow into Con_2 from Con_1 until the levels are equal and to then boil the water in Con_2 to provide steam for Turbine_1.

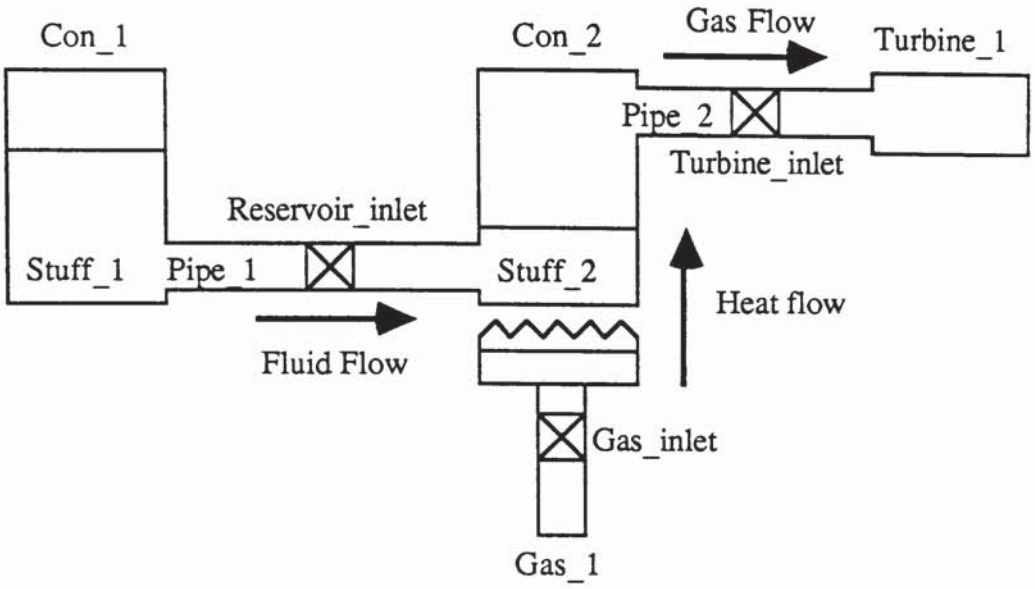



Figure 8.13.

As in the previous example, there will be views held for the individuals such as 'Stuff_1 a liquid' and 'Stuff_2 a contained liquid' as well as views held for the pipes such as 'Pipe_1 aligned' when the reservoir inlet is open.

The plan causes four processes to occur :-

1. A fluid flow from Stuff_1 to Stuff_2 via Pipe_1.
2. A heat flow from gas_1 to Stuff_2 using Con_2 as a heat path
3. A boiling process involving Stuff_2
4. A gas flow involving the stream generated from the boiling process to turbine_1 via Pipe_2.

The initial state of the world before we request the plan is outlined in Figure 8.14.

The plan to generate the steam is described in Figure 8.15.

```
[can contain con_1 stuff_1]
[can contain con_2 stuff_2]
[reservoir_inlet a valve of pipe_1]
[turbine_inlet a valve of pipe_2]
[pipe_1 a fluid path]
[pipe_2 a gas path]
[fpath a fluid path]
[hpath a heat path]
[reservoir_inlet closed]
[turbine_inlet closed]
[reservoir_inlet a valve of con_1]
[reservoir_inlet a valve of con_2]
[turbine_inlet a valve of con_2]
[turbine_inlet a valve of turbine_1]
[fluid connection stuff_1 stuff_2 pipe_1]
[gas connection con_2 turbine_1 pipe_2]
[gas aligned con_2 turbine_1 pipe_2]
[heat aligned gas_1 stuff_2 con_2]
[con_1 rigid]
[con_2 rigid]
[turbine_1 rigid]


[[temperature stuff_1][80 1 0 0]]
[[temperature stuff_2][80 1 0 0]]
[[temperature gas_1][1000 1 0 0]]
[[amount_of_in tap_1][10 1 0 0]]
[[amount_of_in con_1][20 1 0 0]]
[[amount_of_in con_2][10 1 0 0]]
[[boil_point stuff_1][100 1 0 0]]
[[boil_point stuff_2][100 1 0 0]]
[[pressure stuff_1][100 1 0 0]]
[[pressure stuff_2][80 1 0 0]]
```

Figure 8.14.

16
Go to R_inlet
17
Grasp R_inlet
18
Turn R_inlet to open
3
Ungrasp R_inlet
4
Waitfor {pressure contents Con_1 = pressure contents Con_2
19
Go to R_inlet
20
Grasp R_inlet
21
Turn R_inlet to closed
5
Ungrasp R_inlet
22
Go to G_inlet
1
1
23
Grasp G_inlet
24
Turn G_inlet to open
6
Ungrasp G_inlet
7
Turn on gas_1
8
Waitbegin {boiling contents Con_2}
25
Go to T_inlet
26
Grasp T_inlet
27
Turn T_inlet to open
9
Ungrasp T_inlet
2
2
10
Waitfor {pressure contents Turbine_1 = 100
11
Turn off G_inlet
28
Go to G_inlet
29
Grasp G_inlet
30
Turn G_inlet to closed
12
Ungrasp G_inlet
13
Go to T_inlet
14
Grasp T_inlet
15
Turn T-inlet to closed
2
Ungrasp T_inlet
T_inlet = Turbine_inlet
G_inlet = Gas_inlet
R_inlet = Reservoir_inlet

Figure 8.15.

The actions of the plan are dealt with in the usual way by the loader, and from the plan effects asserted, the TNMS finds a fluid flow process between Stuff_1 and Stuff_2, via Pipe_1. From this the Process Reasoner produces the process tree described in Figure 8.16. The plan is then suspended, as in Figure 8.17 while the system waits for the levels to equalise.

```
The current time is now 0 mins 4 seconds

** [The processes active in State 1]

** [process 1 involves a fluid_flow with individuals
    [stuff_1 stuff_2 pipe_1]]


** [The state changes active in State 1]

** [change 1 involves the relation [amount_of stuff_1 <
        zero] to state 3]

** [change 2 involves the relation
   [pressure stuff_1 < pressure stuff_2]
   to state 2]
```

Figure 8.16.

The system then moves the process tree to State 2 and unsuspends the plan so that further actions of the plan can be asserted. From these effects, the TNMS finds an heat connection between gas_1 and Stuff_2 via Con_2 and when gas_1 is lit, a heat flow is found along the same individuals. This process is then analysed by the Process Reasoner which uses State 2 as the starting point of the analysis and from this produces the process tree as described in Figure 8.18 (and diagrammatically in Figure 8.19).

```
The plans known to the system are as follows


        suspended :
        plan number 1 has the goal {generate power}

        The actions executed so far are
                No actions have yet been executed


                No action is currently executing


        The actions sent to the loader are
                [[go to reservoir_inlet] 16 1]
                [[grasp reservoir_inlet] 17 1]
                [[turn reservoir_inlet to open] 18 1]
                [[ungrasp reservoir_inlet] 3 1]
                [[waitfor {pressure contents con_1 = pressure
                            contents con_2}] 4 1]



        The actions awaiting to be sent to the loader are
                [[go to reservoir_inlet] 19 1]
                [[grasp reservoir_inlet] 20 1]
                [[turn reservoir_inlet to closed] 21 1]
                [[ungrasp reservoir_inlet] 5 1]
                [[go to gas_inlet] 22 1]
                [{grasp gas_inlet} 23 1]
                [{turn gas_inlet to open} 24 1]
                [{ungrasp gas_inlet} 6 1]
                [{turn on gas_1} 7 1]
                [{waitbegin {boiling contents con_2}} 8 1]
                [{go to turbine_inlet} 25 1]
                [{grasp turbine_inlet} 26 1]
                [{turn turbine_inlet to open} 27 1]
                [{ungrasp turbine_inlet} 9 1]
                [{waitfor {pressure contents turbine_1 = 100}} 10
                      1]
                [{turn off gas_1} 11 1]
                [{go to gas_inlet} 28 1]
                [{grasp gas_inlet} 29 1]
                [{turn gas_inlet to closed} 30 1]
                [{ungrasp gas_inlet} 12 1]
                [{go to turbine_inlet} 13 1]
                [{grasp turbine_inlet} 14 1]
                [{turn turbine_inlet to closed} 15 1]
                [{ungrasp turbine_inlet} 2 1]
```

Figure 8.17.

```
The current time is now 1 mins 6 seconds

** [The processes active in State 1]

** [process 1 involves a fluid_flow with individuals
    [stuff_1 stuff_2 pipe_1]]


** [The processes active in State 2]

** [process 1 involves a heat_flow with individuals [gas_1
   stuff_2 con_2]]



** [The state changes active in State 2]

** [change 1 involves the relation
   [temperature stuff_2 > boil_point stuff_2]
   to state 5]

** [change 2 involves the relation
   [temperature gas_1 < temperature stuff_2]
   to state 4]



** [The processes active in State 4]

** [There are no processes active in this state]


** [The processes active in State 5]

** [process 1 involves a heat_flow with individuals [gas_1
   stuff_2 con_2]]

** [process 2 involves a boiling with individuals
    [stuff_2]]

** [process 3 involves a heat_flow with individuals [gas_1
   general con_2]]



** [The state changes active in State 5]

** [change 1 involves the relation[pressure con_2 > p_burst
   con_2] to state 9]

** [change 2 involves the relation [amount_of stuff_2 <
   zero] to state 8]

** [change 3 involves the relation [temperature stuff_2 <
   boil_point stuff_2]to state 7]

** [change 4 involves the relation [temperature gas_1 <
        temperature stuff_2] to state 6]
```

```
** [The state influences active in State 5]

** [Influence set 1 involves the process
      [heat_flow]
       with influences
      [[I + heat stuff_2 flowrate gas_1]
       [I - heat gas_1 flowrate gas_1]]]

** [Influence set 2 involves the process
      [boiling]
      with influences
      [[I + heat general generation_rate general]
       [I - heat stuff_2 generation_rate general]
       [I + amount_of general generation_rate general]
       [I - amount_of stuff_2 generation_rate general]
       [I + heat stuff_2 flowrate gas_1]]]

** [Influence set 3 involves the process
      [heat_flow]
      with influences
      [[I + heat general flowrate gas_1]
       [I - heat gas_1 flowrate gas_1]]]

** [Influence set 4 involves the process
      [contained gas]
      with influences
      [[pressure general Q + heat general]
       [pressure general Q - volume general]
       [pressure general Q + amount_of general]
       [temperature general Q + pressure general]
       [pressure stuff_2 Q + pressure general]]]

** [Influence set 5 involves the process rule with
   influences
      [[pressure general = pressure con_2]]]

** [Influence set 6 involves the process rule with
   influences
      [[temperature general = temperature con_2]]]


** [The processes active in State 6]

** [There are no processes active in this state]


** [The processes active in State 7]

** [process 1 involves a heat_flow with individuals [gas_1
   stuff_2 con_2]]

** [process 2 involves a heat_flow with individuals [gas_1
   general con_2]]


** [The state changes active in State 7]

** [change 1 involves the relation[temperature stuff_2 >
        boil_point stuff_2]to state 5]
```

```
** [The processes active in State 8]

** [process 1 involves a heat_flow with individuals [gas_1
   general con_2]]


** [The state changes active in State 8]

** [change 1 involves the relation[pressure con_2 > p_burst
   con_2] to state 11]

** [change 2 involves the relation [temperature gas_1 <
        temperature general]to state 10]


** [The processes active in State 9]

** [process 1 involves a explode with individuals [con_2]]


** [The state changes active in State 9]

** [There are no changes from this state]


** [The processes active in State 10]

** [There are no processes active in this state]


** [The processes active in State 11]

** [process 1 involves a explode with individuals [con_2]]
```

Figure 8.18.

Alternative behaviours for the Boiler by repeated Limit Analysis
PS: (heat flow) 2
LH: (A [temperature (WATER)] = A [temperature (SOURCE)]
IS: (CAN ,WATER)
PS: ( ) 4
LH: A [temperature(WATER)] = A [t-boil(WATER)]
IS: (CAN,WATER,STEAM)
PS: (heat-flow, boiling) 5
LH: A [amount-of(WATER)] = ZERO
IS: (CAN, STEAM)
PS: (heat-flow) 8
LH: A [temperature(STEAM)] = A [temperature(SOURCE)]
IS: (CAN, STEAM)
PS ( ) 10
LH: A [temperature(WATER)] < A [t-boil(WATER)]
IS: (CAN, STEAM, WATER]
PS: (heat-flow) 7
LH: A [pressure(CAN)]
= A [p-burst(CAN)]
IS: (CAN, STEAM)
PS: (Explosion) 11
LH: A [temperature(WATER)] = A [temperature(SOURCE)]
IS: (CAN, WATER, STEAM)
PS: ( ) 6
LH: A [pressure(CAN)] = A [p-burst(CAN)]
IS: (CAN, WATER, STEAM)
PS: (Explosion) 9

Figure 8.19.

The tree is displayed diagrammatically in Figure 8.19 along with the state numbers assigned by the Process Reasoner. As can be seen from the tree, States 11 and 9 both contain explosion events which the system must avoid. Fortunately, at the start stuff_2 has not yet begun to boil so there is no immediate danger. However, when the system is informed that Stuff_2 has begun to boil the danger is now apparent and the system reacts by checking if the plan contains actions which will release this build up of pressure. Fortunately the plan does contains such actions, i.e. 'open the turbine_inlet' so the system can ignore the problem and leave the plan to deal with the situation. Opening the turbine_inlet causes the characteristics of Con_2 to change and with it the behaviour of the process tree. As a result the behaviour is re-analysed and the tree is modified to the one in Figure 8.20, which now also contains the gas flow process which was found by the TNMS when the turbine_inlet was opening, thus allowing the steam to flow into the turbine.

```
The current time is now 1 mins 8 seconds

** [The processes active in State 5]


** [process 1 involves a gas_flow with individuals
    [individual1 individual2 pipe_2]]

** [process 2 involves a heat_flow with individuals [gas_1
   stuff_2 con_2]]

** [process 3 involves a boiling with individuals
    [stuff_2]]

** [process 4 involves a heat_flow with individuals [gas_1
   individual1 con_2]]
```

Figure 8.20.

When the steam in the turbine reaches 100 bar the plan is again unsuspended and further actions are again sent to the TNMS for assertion. The effect of closing the gas_inlet is to cause a failure in the heat flow process and as a result a failure in the boiling process which causes the system to move the process tree to its final point in State 6. The state of the plan and the process tree are outlined in Figure 8.21. (To

make this example clearer, the failure messages generated by the Dependency System in response to changes in the justifications of actions and processes in the TNMS have been ignored).

```
The current time is now 1 mins 27 seconds

The plans known to the system are as follows
plan number 1 has the goal {generate power}


The actions executed so far are
        [[go to reservoir_inlet] 16 1]
        [[grasp reservoir_inlet] 17 1]
        [[turn reservoir_inlet to open] 18 1]
        [[ungrasp reservoir_inlet] 3 1]
        [[waitfor [pressure contents con_1 = pressure
                contents con_2]] 4 1]
        [[go to reservoir_inlet] 19 1]
        [[grasp reservoir_inlet] 20 1]
        [[turn reservoir_inlet to closed] 21 1]
        [[ungrasp reservoir_inlet] 5 1]
        [[go to gas_inlet] 22 1]
        [[grasp gas_inlet] 23 1]
        [[turn gas_inlet to open] 24 1]
        [[ungrasp gas_inlet] 6 1]
        [[turn on gas_1] 7 1]
        [[waitbegin {boiling contents con_2}] 8 1]
        [[go to turbine_inlet] 25 1]
        [[grasp turbine_inlet] 26 1]
        [[turn turbine_inlet to open] 27 1]
        [[ungrasp turbine_inlet] 9 1]
        [[waitfor {pressure contents turbine_1 = 100}] 10
              1]

The action currently executing is
        [[turn off gas_1] 11 1]

The actions sent to the loader are
        [[go to gas_inlet] 28 1]
        [[grasp gas_inlet] 29 1]
        [[turn gas_inlet to closed] 30 1]
        [[ungrasp gas_inlet] 12 1]
        [[go to turbine_inlet] 13 1]
        [[grasp turbine_inlet] 14 1]
        [[turn turbine_inlet to closed] 15 1]
        [[ungrasp turbine_inlet] 2 1]

The actions awaiting to be sent to the loader are
       No more actions are waiting to be sent

The current time is now 1 mins 27 seconds

** [The processes active in State 6]

** [There are no processes active in this state]
```

Figure 8.21

A second example shows the system's ability to deal with a situation in which an explosion is predicted to occur and no action was initially present in the plan to stop this occurring. The events are as above except that once the gas flow has begun, the user informs the system that the turbine_inlet has been closed. This causes a change in the characteristics of Con_2 and with it a change in the process tree behaviour, as described in Figure 8.22. As the un-executed actions of the plan do not stop the explosion occurring, the system intervenes and generates a patch plan to open the turbine_inlet so as to release the pressure. The modified plan is outlined in Figure 8.23. The actions are sent to the TNMS to be asserted even though the plan is suspended, so as to stop the explosion happening. The process tree is again modified to reflect the change in behaviour of Con_2 as described in Figure 8.24.

```
The current time is now 1 mins 31 seconds

** [The processes active in State 5]

** [process 1 involves a heat_flow with individuals
    [gas_1 stuff_2 con_2]]

** [process 2 involves a boiling with individuals
    [stuff_2]]

** [process 3 involves a heat_flow with individuals
      [gas_1 general con_2]]


** [The state changes active in State 5]

** [change 1 involves the relation
     [pressure con_2 > p_burst con_2]
     to state 9]

** [change 2 involves the relation
     [amount_of stuff_2 < zero]
     to state 8]

** [change 3 involves the relation
     [temperature stuff_2 < boil_point stuff_2]
     to state 7]

** [change 4 involves the relation
     [temperature gas_1 < temperature stuff_2]
     to state 6]
```

Figure 8.22.

```
The current time is now 1 mins 33 seconds


The plans known to the system are as follows


suspended :
plan number 1 has the goal {generate power}


The actions executed so far are
        [[go to reservoir_inlet] 16 1]
        [[grasp reservoir_inlet] 17 1]
        [[turn reservoir_inlet to open] 18 1]
        [[ungrasp reservoir_inlet] 3 1]
        [[waitfor [pressure contents con_1 = pressure
                    contents con_2]] 4 1]
        [[go to reservoir_inlet] 19 1]
        [[grasp reservoir_inlet] 20 1]
        [[turn reservoir_inlet to closed] 21 1]
        [[ungrasp reservoir_inlet] 5 1]
        [[go to gas_inlet] 22 1]
        [[grasp gas_inlet] 23 1]
        [[turn gas_inlet to open] 24 1]
        [[ungrasp gas_inlet] 6 1]
        [[turn on gas_1] 7 1]
        [[waitbegin {boiling contents con_2}] 8 1]
        [[go to turbine_inlet] 25 1]
        [[grasp turbine_inlet] 26 1]
        [[turn turbine_inlet to open] 27 1]
        [[ungrasp turbine_inlet] 9 1]


The action currently executing is
        [[go to turbine_inlet] 33 1]


The actions sent to the loader are
        [[grasp turbine_inlet] 34 1]
        [[turn turbine_inlet to open] 35 1]
        [[ungrasp turbine_inlet] 32 1]
        [[waitfor {pressure contents turbine_1 = 100}] 10
               1]


The actions awaiting to be sent to the loader are
        [[turn off gas_1] 11 1]
        [[go to gas_inlet] 28 1]
        [[grasp gas_inlet] 29 1]
        [[turn gas_inlet to closed] 30 1]
        [[ungrasp gas_inlet] 12 1]
        [[go to turbine_inlet] 13 1]
        [{grasp turbine_inlet} 14 1]
        [{turn turbine_inlet to closed} 15 1]
        [{ungrasp turbine_inlet} 2 1]
```

Figure 8.23.

```
The current time is now 1 mins 33 seconds


** [The processes active in State 5]

** [process 1 involves a gas_flow with individuals
    [individual1 individual2 pipe_2]]

** [process 2 involves a heat_flow with individuals
    [gas_1 stuff_2 con_2]]

** [process 3 involves a boiling with individuals
    [stuff_2]]

** [process 4 involves a heat_flow with individuals
      [gas_1 general con_2]]



** [The state changes active in State 5]

** [change 1 involves the relation
     [pressure turbine_1 > p_burst turbine_1]
     to state 12]

** [change 2 involves the relation
     [amount_of stuff_2 < zero]
     to state 8]

** [change 3 involves the relation
     [temperature gas_1 < temperature stuff_2]
     to state 6]
```

Figure 8.24.

As these examples show the planning system described in this thesis represents a very significant advance over existing systems. The particular features demonstrated by the examples are :

1. The ability to handle continuous processes, including waiting for them to begin or end as well as waiting for events caused by processes.
2. The ability to synchronise an internal world model with events occurring in the world and to predict these changes correctly.
3. The ability to handle plan actions with side effects which do not need to be described in the plan.

4. The ability to replan sections of a plan to overcome problems caused by plan precondition failure as well as precondition failures in processes on which the plan is dependant, while keeping as much of the original plan as possible.

5. The ability to modify a plan and to re-schedule some of its actions so as to avoid a situation which the systems finds undesirable. In the examples shown, the situation to avoid was an explosion but it could easily have been to avoid over filling the kettle or cup.

6. The representation scheme used can easily be modified by adding new individuals, views and schemas. The examples shown here use only a subset of the possible schemas which the system can accept. For example process schemas have been written to represent motion processes which would obviously be necessary in the plans of a mobile robot.

# Chapter 9: Conclusions and Proposals for Further Research

## 9.1 Conclusions

The planning system described in this thesis has been implemented and tested. The system performed as expected, and can deal with complex problems involving planning and process theory. The time representation scheme performed better than was expected with no noticeable degradation in search and update times until around 500 tokens were asserted; the degradation then was only marginal.

The advantages of this time representation over Dean's system (1983, 1984, 1987) are as follows:

1. Processes and continuous events to be represented and reasoned about.

2. The token justification system has ben extended to allow processes to be justified using quantity relationships rather than just temporal relationships.

3. The representation scheme has the ability to synchronise its internal world model with the real world as time passes and to analyse any deviation from the expected path, thus re-aligning the knowledge base with the correct future.

4. The explicit linkage between the effects of actions and the side effects and processes with they bring about in the real world.

The system was tested using simple examples from everyday reasoning and more complex examples involving construction and process tasks. Any plan or process errors were dealt with successfully with the integrity of the TNMS being maintained at all times. Hence a set of techniques for execution plan monitoring and error

analysis have be derived and put into practice. These could be adapted, by means of the relevant process schema definitions, to any chosen planning domain.

The intentions of this project were to devise and verify an execution monitoring and error analysis philosophy based on process theory. To this extent it has been successful. Also, a useful by-product has been that a fast and efficient temporal knowledge data base has been produced which could easily be used in other applications such as natural language.

It is interesting to contrast the approach taken here with that of Vere's (1983) Devisor (see Chapter 2). The approaches differ in two respects. First, Excalibur allows plan to be suspended for varying lengths of time awaiting an external precondition. Second, the events the plan has to interact with do not have to be specified as occurring at a specific time as they were in Devisor.

It is also interesting to note that Hogge (1987), a colleague of Forbus - the inventor of "Process Theory", has tried to implement a planning system capable of handling processes and the changes they bring about. His system integrated quantity preconditions as part of the plan schema preconditions, and thus goals of the type 'increase the level of water in tank2' could be achieved. However, his system did not deal with planning over time and the plans were never executed in the real world. The system he developed used Nonlin-type schema definitions and with only minor modifications could be easily integrated into the Excalibur system detailed here.

Based on the work described in this thesis, it is possible to put forward several advantages that this system of execution plan monitoring and error analysis has over existing systems. These are:

1. The user is able to define changes like 'boiling', 'flowing', 'heating', 'moving', etc., outside of the plan, which means they do not have to be copied into each new application.

2. Changes which a plan should bring about in the real world can be defined within the plan without the need for massive precondition lists to check that the change will indeed come about.

3. Changes can occur in the world without the planner having to initiate them, so that actions can cause quite complicated side effects.

4. The system reacts in a conditional way to changes in the world as it plans both to avoid and to create conditions which it requires.

5. The time representation scheme allows a clear definition of token types and the tokens themselves can have varying and possibly infinite durations.

6 The Plan Reasoner can monitor a suspended plan which is waiting for an external event, but will never allow the plan to wait for an event which can never come about.

7. The system has the ability to react to situations by creating a new plan to overcome problems caused by failures of plan preconditions or failures caused by a process not bringing about required effects.

8. The Process Reasoner has the ability to reason about unwanted situations (as in the explosion example) and to carry out the analysis necessary to create a plan to avoid it.

## 9.2 Proposals for further Research

During this research project, several ideas for further research in this field have come to light. These suggestions can be divided into two categories. The first

contains improvements to the existing system and the second outlines new problem areas which could be explored using the techniques outlined in this thesis. These are possible extensions to the work carried out to date, but have not been pursued due to the limited amount of time available.

### 9.2.1 System Improvements

The modifications which could be made to improve the function of the program can be divided into two parts. Firstly, in the current system the TNMS, Plan Reasoner and Process Reasoner are each executed in turn to check if there is any problem for them to analyse. This sequential ordering can sometimes lead to timing problems caused by information being present in the TNMS which is analysed before all of its consequences have been analysed. In the coffee making example, we create a new individual by pouring the water from the tap into the kettle. This new individual will have all the attributes of the water in the tap and thus will be identified by the TNMS, firstly as a liquid and then as a contained liquid. The timing problem occurs if the Process Reasoner tries to analyse the effects of the fluid flow between the tap and the kettle before the TNMS has asserted the two views outlined above for the new individual. Normally, this would have no effect; however there are situations where it might. A contained liquid has a relation that its level is proportional to its amount. If the relation is not available when the fluid flow is analysed, then the Process Reasoner will not deduce that the level of the new individual will increase due to this influence. If the level of the new individual is required as part of a waitfor directive, as in the case of filling a cup to a specified level, then the Plan Reasoner will not know the process was helping bring about the condition it is waiting for, should the process fail. To solve this problem, in the current system the TNMS scans for the views of the individual which creates the new individual and uses these templates on the knowledge base. This could be overcome by implementing the three main modules of Excalibur as separate

processes within the UNIX or VMS operating systems. The current system uses a blackboard architecture to pass information between modules which could be replaced by 'pipes' (data channels between processes in the VMS or UNIX system). The system was designed so that neither of the three major modules accesses data held in another module and thus the modules can be run in parallel. Once such a parallel system had been achieved, the problem of scheduling the operations of the system would arise. This involves problems such as how much time should be spent in each module and should this be varied based on the type of problem being solved. One possibility would be to analyse the process tree; for example, should this show that the current state may lead to an undesirable state, then more time should be spent on the task of picking up earlier (via the sensors) any changes towards the undesirable state.

Secondly, in the current system the user must provide information about events happening in the real world in place of sensors. This is because no system of sensors was available when Excalibur was implemented. The change to using real sensors could be achieved quite easily by adding extra software to poll the devices on a cyclic basis, and on demand from the Process Reasoner. The data could then be entered in the TNMS at the time point it was collected, which in the case of sensor data would be the time point 'now'.

### 9.2.2 Further Research

At present, the world model of the TNMS and the world description used by Nonlin are held in separate models. One of the functions of the Plan Reasoner is to find all the tokens which are active at the point in time when any replanning is required, so providing the initial context to the plan. Ideally, Excalibur and Nonlin should share the same world model avoiding this extra processing. To overcome this problem would require changes to Nonlin to allow it to be re-started from any point in the time map, and to have direct access to the time map.

In the current system the Plan Reasoner is able to reason about facts and events happening in the real world. However the world is not only populated by objects such as tanks, pipes, etc. but by other intelligent agents. The beliefs and intentions of other agents, whether they are human or other planners could be represented and non-monotonically justified in the TNMS, like any other token. The research problem would be how to reason about this belief and to detect conflict between the plan of one agent and another. The research in belief and intention by Konolige (1983) and Moore (1980) could provide such a reasoning mechanism, but it would have to be modified to deal with the temporal aspects of the problem.

The present system is capable of detecting and repairing problems arising from interference with the execution of a plan. However, this is not the only reason why a planner may chose to alter the order of execution of a plan. A plan may be altered because it is no longer required, because the goal it achieves has been subsumed. That is, there may exist a sub-goal which, by being satisfied, will contribute to one or more higher level goals. For example, the goal of buying a car may subsume the goal of getting to work on time and the goal of being able to visit relatives regularly. At present Excalibur is unable to carry out goal subsumption, and implementing it is not an easy problem to solve. It would require a priority system to be placed on goals, which is difficult to achieve in any realistic domain, and also the ability to reason about the side effects which a plan to solve the higher level goal may bring about, especially in the presence of processes.

# Appendix A: Communication within the TNMS

## A.1 Statement descriptor

The function of the statement predicate is to assert a token in the TNMS. The token to be asserted can be one of two types:

1. The assertion of a fact, event, action or process that is occurring or will occur in the real world.
2. The assertion of quantitative data from sensors or the user about actual values of quantities in the real world.

The assertion of a token in the TNMS can be accomplished in one of two ways. The token can be asserted to occur relative to the time point 'now' or the token can also be asserted relative to an already asserted base token. The message descriptor is broken down into three parts.

1. The token to be asserted.
2. The time the token will begin, asserted either from the time point 'now' or the base event.
3. The duration the token is expected to persist for.

The following sections shows three examples of the statement descriptor and its uses. The syntax of the statement descriptor is :-

[Statement_<n> [predicate to be asserted]
[start time of the asserted predicate]
[the expected duration of the asserted token]]

```
[Statement_1    [Occ after [Heat_flow Flame Water Boiler]]
                [Pressure Boiler1 Increasing]
                [elt [distance cbegin send 0.0 0.5]]
                [elt [distance cbegin cend 0.0 pos_inf]]]
```

Figure A.1.

This states that the pressure of the boiler started to increase between 0.0 and 0.5 seconds after the start of the heat flow in to the boiler. The pressure increase is expected to increase for a possibly infinite time. If the user does not specify any distance information the TNMS assumes it begins immediately after the start event and exists for an infinite duration.

```
[Statement_2    [Occ [Level tank2 = Zero]]
                [elt [distance begin 10.0 15.0]]
                [elt [distance cbegin cend pos_inf pos_inf]]]
```

Figure A.2.

This example states that the level of tank2 will reach Zero within the next 10 to 15 seconds and will remain empty for a infinite amount of time. Again any missing information is defaulted as example 1 above.

```
[Statement_3    [[temperature][stuff_1]]
                [10 1 0 0]
                [elt [distance cbegin send 0.0 0.0]]
                [elt [distance cbegin cend 0.0 pos_inf]]]
```

Figure A.3.

This is an example of the input of quantitative data and states that the temperature of stuff_1 is 10 units, it is positive and is not changing

## A.2. Query descriptor

The query descriptor is used by the Plan Reasoner to request the TNMS to search for an interval which has a certain set of attributes concerning its duration, start time and relation to other tokens already asserted.The relationships which the query system can check for are 'After', 'Before' and 'During'. The interval is required by the Plan Reasoner in order to execute an action from a given plan. The message descriptor can be broken down into six parts.

1. The facts which must hold true over the interval required, i.e. the preconditions of the action.
2. The context to be scanned for the required interval.
3. The minimum and maximum duration of the required interval.
4. The minimum and maximum start times of the interval relative to the time point 'now'.
5 The action to be asserted in the TNMS.
6. The effects of the queried action which are to be asserted as beginning after the action's completion.

Two examples of the interval query are described in Figures A.4 and A.5. The syntax of the query descriptor is as follows.

[Query [preconditions to be met]
[the context to be scanned]
[the minimum and maximum duration of the required interval]
[start time of the interval][the action and its effects]]

```
[Query      [Occ [[frame and roof erected][foundations laid]]]
            [1]
            [elt  [distance begin end]  [ 10.0 15.0 ]]
            [elt  [distance now begin] [ 2.0 10.0 ]]
            [[lay storm drains] 4 5][[storm drains laid]]]]
```

Figure A.4.

This query request the query analyser to find an interval which is between ten and fifteen seconds long and begins within the next two to ten seconds. The preconditions which must be true are the 'frame and roof erected' and the 'foundations laid'. The action itself 'lay storm drains' has only one effect namely the 'storm drains laid'. The two numbers associated with the action inform the reasoner of which plan this action is from and the number of the action within the plan.

```
[Query      [Occ [after valve1 open]
            [before pressure tank1 = Zero]
            [during fluid flow tank1 tank2]]
            [1]
            [elt [distance begin end][ 10.0 15.0 ]]
            [elt [distance now begin][ 20.0 pos_inf ]]
            [[open valve2] 1 12][[valve2 open]]]]
```

Figure A.5.

This query request the analyser to find an interval which has specific relationships to other intervals. Again the duration is ten to fifteen seconds but the start time is extended to cover a period from twenty seconds into the future to an infinite time. If the interval search is successful then the action is asserted to occur in the interval found and a message is sent to the Plan Reasoner that the search was successful. If however the search fails then the message returned annotates which

preconditions failed and the point in time at which, if the failing preconditions were satisfied, then together with the satisfied preconditions the action could be executed. Two examples of the return messages are detailed in Figure A.6.

[plan action [[lay storm drains] 4 5] has been set up successfully]

[plan action [[lay storm drains] 4 5] has precondition failure [[foundations laid]][20.0 20.0]]

Figure A.6.

## A.3. Time out descriptor

The time out descriptor is used by the Process Reasoner to determine if changes in the real world are occurring within the time scale expected. The time out refers to a window of time during which an occurrence should take place. The tests can be made between

1. a quantity table entry and a context table entry.
2 two entries in the quantity table.
3. two entries in the context table.

The test lies dormant in the TNMS until either the relationship becomes true or the window times out. The test can fail because the first condition or trigger event is not seen within the required period of time or the second condition does not arise within the required time window after the trigger event.

The message can be broken down into three parts :-

1. The message type from the table above either 1, 2 or 3.
2. The fields which are to be tested for in the time out.
3. The time out window for the first entry and the time window to the second event.

Examples of the use of the time out descriptor are detailed in Figure A7, A8 and A9. The syntax of the request predicate is as follows:

[request_<n> [the event which is to act as the trigger]
[the event which should be seen]
[time after the trigger event the second event should be seen]
[time from now to scan for the trigger event]]

[Request_1 [elt [distance [valve_1 closing]
[valve_1 closed]]][30.0 50.0][0.0 0.0]]

Figure A.7.

This request is to check that a token 'valve closed' is asserted in the context table within thirty to fifty seconds from 'now' asserting that the 'valve is closing'

[Request_2 [elt[Distance [fluid flow stuff_1 stuff_1]
[pressure stuff_1>pressure stuff_2]]]
[0.0 10.0][10.0 20.0]]

Figure A.8.

This request is to check that after the token asserting a fluid flow process between the individuals defined begins, within the next ten to twenty seconds and within ten seconds of this occurring, the quantity table reflects the change in pressure or the token asserting the quantity change is asserted in the context table.

[Request_3 [elt [Distance [pressure tank1 increasing]
[level tank1 = level tank2]]]
[0.0 10.0][10.0 15.0]]

Figure A.9.

This request is to check that after the pressure change in tank1 is noted, then there is a change in the level of the tanks reflected either in the quantity table or the context table. The pressure increase should begin within the next ten to fifteen seconds and the level should equalise within ten seconds of the pressure increasing.

The request messages lie dormant in the TNMS until they are triggered by the required information or removed by the TNMS due to the fact that they are redundant. The reply messages sent by the TNMS are detailed below.

1. [Time out within bounds with relationship : <request message>]

This indicates to the reasoner which sent the message that the test fired successfully and the tokens and/or quantities are in place in the TNMS.

2. [Time out error with second event in relationship : <request message>]

This indicates the trigger event was seen but the second event was not seen within the required time interval.

3. [Time out error with relationship : <request message>]

This indicates the trigger event was not seen within the required time interval.

## A.4. Diagnostic descriptor

The diagnostic descriptor is used to inform either the Process Reasoner or Plan Reasoner of a failure in the justification of a process or action which subsequently threatens a token projection. This descriptor is unlike the previous descriptors as it is not sent in response to a request from either reasoner. The types of failure can be broken down into those occurring with currently executing actions or processes and those concerning future actions or processes. A failure occurs because:-

1. a precondition is constrained to begin after the asserted begin point of the action or process.
2. a precondition is truncated to terminate before the end point of an action or process.

The message indicates the type of failure which has occurred and the precondition responsible so the reasoner can re-establish the justification if required. In the case of a process failure this might not be necessary as the process might be expected to fail due to natural causes. For example a heat flow would stop because the destination and source temperatures become equal and not because of any external influences.

Examples of the use of this descriptor are outlined in Figure A.10.

[Future action [[lay storm drains] 4 5]
threatened by late start of precondition [[frame and roof erected]]]

[Future action [[lay storm drains] 4 5]
threatened by truncation of precondition
[[frame and roof erected][foundations laid]]

Figure A.10.

The two descriptors in Figure A.10. indicate a failure with a precondition of an action asserted to start in the future. Given this information the Plan Reasoner can evaluate the error and if deemed necessary re-establish the precondition(s) which are invalidated.

[Future process [motion crate_1]
threatened by late start of precondition
[crate_1 moveable]]

[Future process [heatflow flame water boiler]
threatened by truncation of precondition
[[valve_1 open ]]

Figure A.11.

The two descriptors in Figure A.11 indicate a failure of a process which may or may not be expected. The second example shows how the TNMS must recognise that the closing of valve_1 which is part of a flame assembly will cause the heat flow to fail even though the current influences and quantity values indicate it can still remain active

## A.5. Process descriptor

In order to maintain a clean distinction between the function of the Process Reasoner and the TNMS, the detection of a process becoming active in the knowledge base is achieved by the TNMS and not by the Process Reasoner scanning the contexts of the TNMS for the required facts. The detection of processes is carried out by the TNMS on a cyclic basis. It checks each of the process schemas which have been defined against facts within a single context to try and fill the precondition slots in the process schema. If a process is detected then a message is sent to the Process Reasoner indicating :-

1. The type of process e.g. heatflow, motion, boiling etc.
2. The individuals involved in the process.
3. The context and time points of the newly asserted process.

Examples of the use of this descriptor can be found in Figure A.12.

[process boiling [water1] [1 pt301 pt302 pt303 pt304]]

Figure A.12.

This indicates a boiling process involving water1 has been found in context number 1 and has been set up in the TNMS with the points outlined in the descriptor.

# Appendix B: Data Structures and Algorithms

The representation of tokens within a given context is held in three separate data structures: -

1. **The context table line.**

   The context table line is a list of tokens which are true in a given context. It is used to check if a set of facts are true in a given context without being concerned as to the actual time values of the tokens. The matching function used is extremely fast and there is an appreciable advantage in search time by scanning this first before checking to see if the facts are all true at a required point in time.

2. **The context table.**

   The context table contains the event and fact tokens (i.e processes and actions) and their associated time points. If the token is asserted more than once, then the data points are added to the first occurrence thus avoiding duplicate entries.

3 **The time line table.**

   The time line contains the actual values of the time points held in the context table and is split into two sections. The first contains the time points which have numeric values and the second contains those points which are defined as pos_inf. The numeric line is sorted into time order and the relationships between intervals can be easily found by comparing the values from this time line.

The three data structures used to hold the tokens and their associated time points are described in Figures B.1,B.2 and B.3

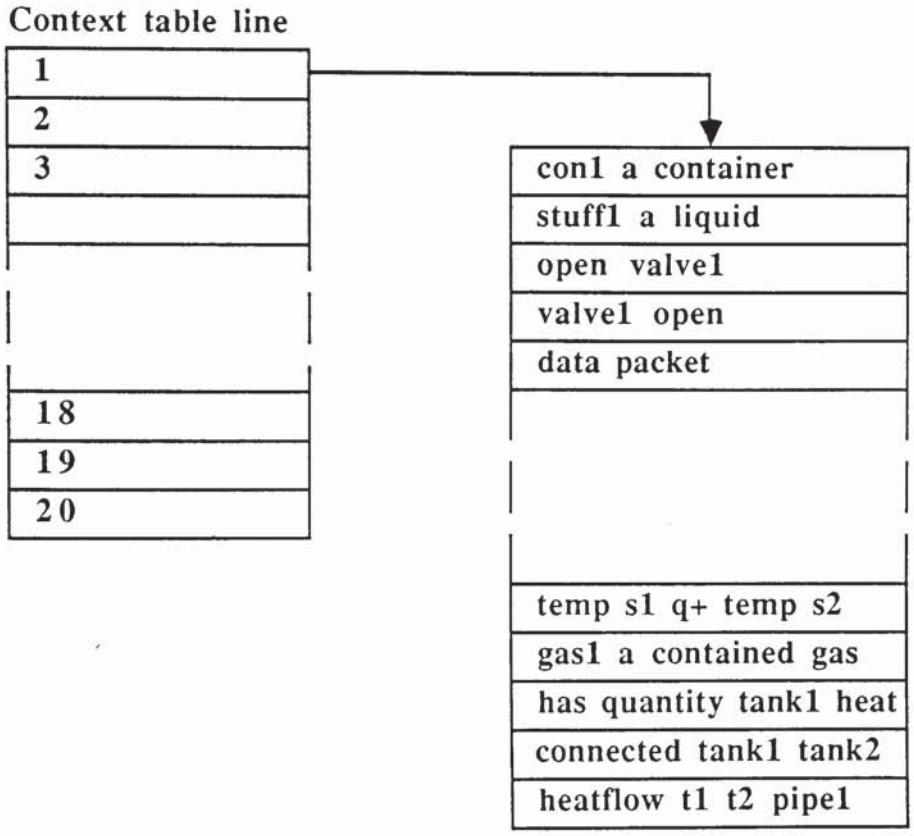



Figure B.1

Context table

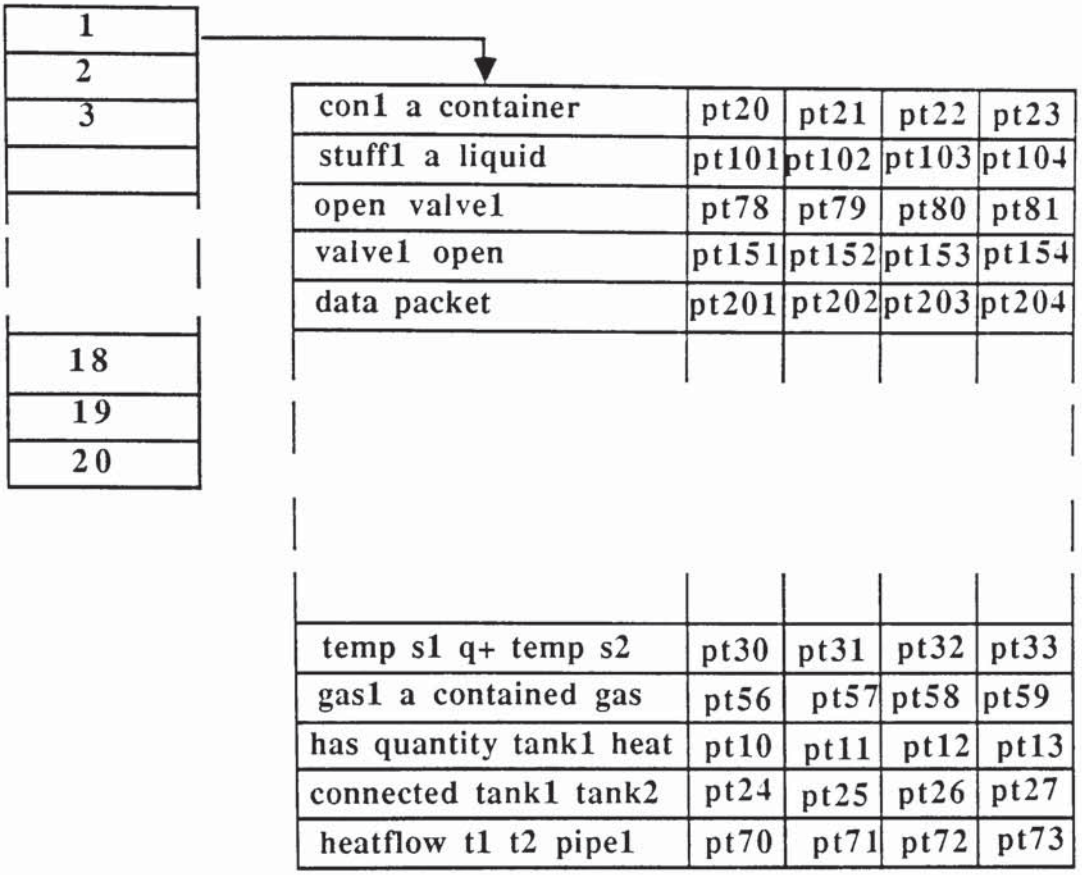



Figure B.2

Time Line Table

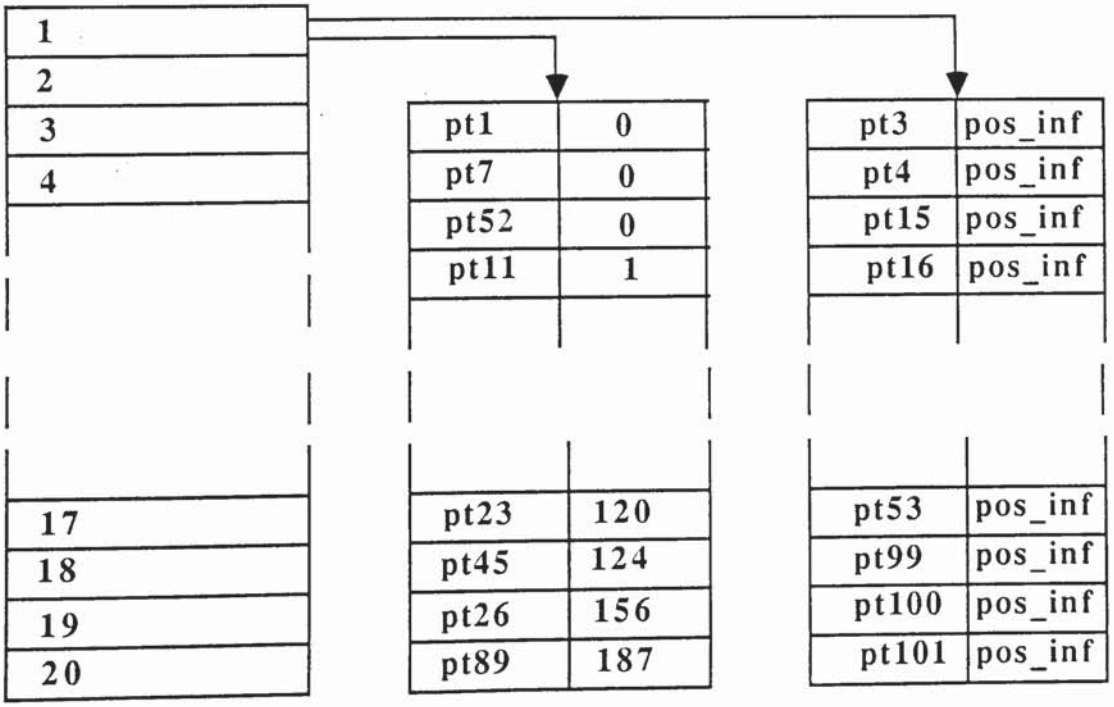



Figure B.3

The tokens which are held in the TNMS are justified according to the two algorithms outlined below in Figures B4 and B5. The '+' symbol means the field below is dependant on the field above being believed IN and a '-' symbol means the field is believed to be OUT.

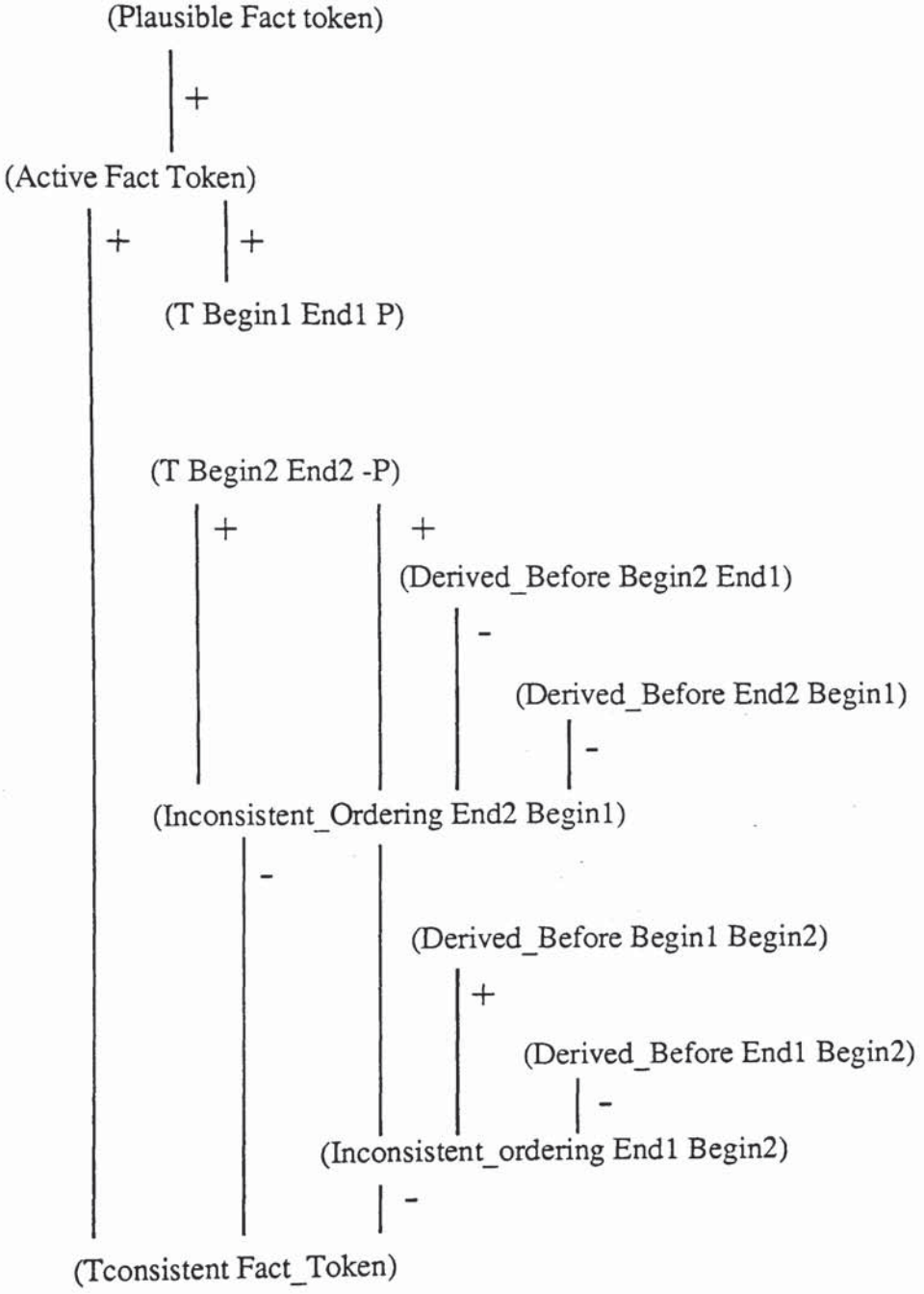
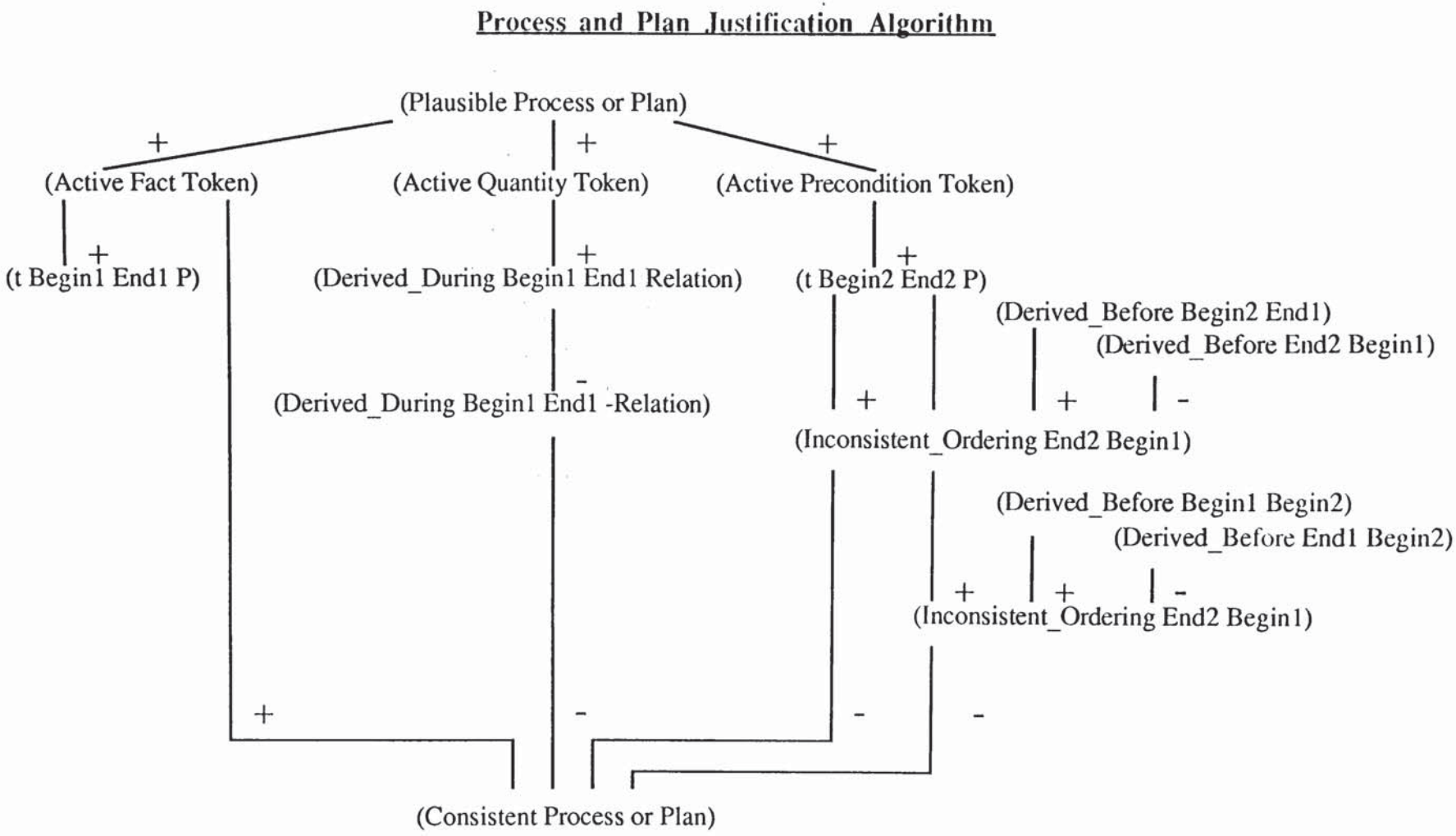



Figure B.4.

Figure B.5.


Process and Plan Justification Algorithm
(Plausible Process or Plan)
+
+
+
(Active Fact Token)
(Active Quantity Token)
(Active Precondition Token)
+
+
+
(t Begin1 End1 P)
(Derived_During Begin1 End1 Relation)
(t Begin2 End2 P)
(Derived_Before Begin2 End1)
(Derived_Before End2 Begin1)
-
(Derived_During Begin1 End1 -Relation)
+
+
-
(Inconsistent_Ordering End2 Begin1)
(Derived_Before Begin1 Begin2)
(Derived_Before End1 Begin2)
+
+
-
(Inconsistent_Ordering End2 Begin1)
+
-
-
-
(Consistent Process or Plan)

# References


Airenti, G. (1985)

"An Approach to Intelligent Action Execution", In: *Proceedings of the Third International Conference on Artificial Intelligence and Information Systems of Robots", June 11-15, Smolence, Czechoslavakia.* Amsterdam: North-Holland, pp81-86.

Allen, J.F. (1981)

"An Interval Based Representation of Temporal Knowledge", In: *Proceedings of the Seventh International Joint Conference on Artificial Intelligence, August 1981, Vancouver, British Columbia, Canada.* Los Altos: William Kaufman Inc, pp221-226.

Allen, J.F. (1983)

"Maintaining Knowledge about Temporal Intervals", *Communications of the ACM,* **26,** pp832-843.

Allen, J.F. (1984)

"Towards a General Theory of Action and Time", *Artificial Intelligence,* **23**, pp123-154.

Allen, J.F. and Kooman, J.A.(1983)

"Planning using a Temporal World Model", In: *Proceedings of the Eighth International Joint Conference on Artificial Intelligence, August 1983, Karlsruhe, West Germany.* Los Altos: William Kaufman Inc, pp741-749.

Allen, J.F. and Hayes, P.J. (1986)

"A Common sense Theory of Time", New York: Department of Computer Science and Philosophy, Rochester University, (NY-TR-257).

Bresina, J.L.(1981)

"An Interactive Planner that creates a Structured Trace of its Operation", New York: Computer Science Research Laboratory, Rutgers University, (Report CBM-TR-123).

Brooks, R.A. (1981)

"Symbolic Error Analysis and Robot Planning", *International Journal of Robotics Research,* **1**, pp3-22.

Bruce, B.C. (1972)

"A Model for Temporal References and its Application in a Question and Answering System", *Artificial Intelligence,* **3**, pp1-25.

Cheeseman, P. (1984)

"A Representation of Time for Automatic Planning", In: *Proceeding of International Conference on Robotics, March 1984, Atlanta, USA*. Stanford: SRI-International, pp98-105.

Davis, R. (1984)

"Diagnostic Reasoning Based on Structure and Behaviour", *Artificial Intelligence*, **24**, pp347-410.

Dean, T.L. (1983)

"Time Map Maintenance", New Haven: Department of Computer Science, Yale University, (Yaleu/CSD/RR-289).

Dean, T.L. (1984)

"Planning and Temporal Reasoning under Uncertainty", In: *Proceedings of the IEEE Workshop on Principles of Knowledge Based Systems", December 1984, Denver, Colorado*. Colorado: IEEE, pp210-213.

Dean, T.L. (1985)

"Temporal Reasoning involving Counter Factuals and Disjunctions", In: *Proceedings of the Ninth International Joint Conference on Artificial Intelligence, August 18-23 1985, Los Angeles, California*. Los Altos: William Kaufman Inc, pp1060-1062.

Dean, T.L. and McDermott, D.V. (1987)

"Temporal Database Management", *Artificial Intelligence*, **23**, pp1-58.

DeKleer, J. and Brown, J.S. (1985)

"Assumptions and Ambiguities in Mechanistic Models", In: *Mental Models*, Edited by Dedre Gentner and Albert Stevens, Cognitive Science Series, Hillsdale, New York: Lawrence Erlbaum, pp155-190.

Doyle, J. (1979)

"A Truth Maintenance System", *Artificial Intelligence,* **12**, pp231-272.

Fahlman, S.E. (1974)

"A Planning System for Robot Construction Tasks", *Artificial Intelligence,* **5**, pp1-49.

Falletti, J. (1982)

"Pandora: A program for doing Commonsense Reasoning Planning in Complex Situations", In: *Proceeding of the Second National Conference on Artificial Intelligence, August 1982, Pittsburgh, Pennsylvania.* Los Altos: William Kaufman Inc, pp185-188.

Fikes, R.E. and Nilsson N.J. (1971)

"Strips a New Approach to the Application of Theorem proving to Problem Solving", *Artificial Intelligence,* **2**, pp189 -208.

Fikes, R.E., Hart, P.E. and Nilsson, N.J. (1972)

"Learning and Executing Generalised Robot Plans", *Artificial Intelligence,* **3**, pp251-288.

Forbus, K.D. (1981)

"Qualitative Reasoning about Physical Systems", In: *Proceedings of the Seventh International Joint Conference on Artificial Intelligence, August 1981, Vancouver, British Columbia, Canada.* Los Altos: William Kaufman Inc, pp642-645.

Forbus, K.D. (1983)

"Qualitative Process Theory", *Artificial Intelligence,* **24,** pp85-168.

Forbus, K.D. (1986)

"*The Qualitative Process Engine*" : Illinois: Department of Computer Science, University of Illinois, (Technical Report UIUCDCS-R-86-1288).

Green, C.C. (1969)

"Application of theorem Proving to problem Solving", In: *Proceedings of the Second International Joint Conference on Artificial Intelligence, May 1969, Washington, USA..* Los Altos: William Kaufman Inc, pp219-239.

Hayes, P.J. (1973)

"The Frame Problem and Related Problems in Artificial Intelligence", In: *Artificial Intelligence and Human Thinking*: Nato Symposuim on Human Thinking, St Maximin, France 1971; Edited by Alick Elithon and David Jones. San Francisco: Jossey-Bass, pp45-59.

Hayes, P.J. (1975)

"A Representation for Robot Plans", In: *Proceedings of the Fourth International Joint Conference on Artificial Intelligence, September 1975, Tbilisi, USSR.* Los Altos: William Kaufman Inc, pp181-188.

Hayes, P.J. (1979)

"The Naive Physics Manifesto", In: *Expert Systems in the Microelectronic Age*, Edited by Donald Michie. Edinburgh: Edinburgh University Press.

Hayes-Roth, B. and Hayes-Roth, F. (1979)

"A Cognitive Model of Planning", *Cognitive Science*, **3**, pp275-310.

Hendrix, G.G. (1973)

"Modelling Simultaneous Actions and Continuous Processes", *Artificial Intelligence*, **4**, pp145-180.

Hogge, J.C. (1987)

"Compiling Plan Operators from Domains Expressed in Qualitative Process Theory", In: *Proceedings of the Sixth National Conference on Artificial Intelligence, July 1987, Seattle, Washington.* Los Altos: William Kaufman Inc, pp229-233.

Kahn, K, and Gorry, G.A. (1977)

"Mechanising Temporal Knowledge", *Artificial Intelligence*, **9**, pp87-108.

Konolige, K. (1983)

"A Deductive Model of belief", In: *Proceedings of the Eighth International Joint Conference on Artificial Intelligence, August 1983, Karlsruhe, West Germany*. Los Altos: William Kaufman Inc, pp377-381.

Masiu, S., McDermott, J. and Sobel, A. (1983)

"Decision making in Time Critical Situations", In: *Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 8-12 August 1983, Karlsruhe, West Germany*. Los Altos: William Kaufman Inc, pp233-235.

McCalla, G.I., Reid, L. and Schneider, P.F. (1982)

"Plan Creation, Plan Execution and Knowledge Acquisition in a Dynamic World", *International Journal of Man Machine Studies*, **16**, pp89-112.

McDermott, D.V. (1978)

"Planning and Acting", *Cognitive Science*, **2**, pp71-109.

McDermott, D.V. (1982)

"A Temporal Logic for Reasoning about Processes and Plans", *Cognitive Science*, **6**, pp101-155.

McDermott, D.V. and Doyle, J. (1979)

"An Introduction to Non-Monotonic Logic", In: *Proceedings of the Sixth International Joint Conference on Artificial Intelligence, August 1979, Tokyo Japan*. Los Altos: William Kaufman Inc, pp562-567.

Minsky, M. (1974)

"*A framework for Representing Knowledge*", Reading: MIT Press, Artificial Intelligence Laboratory Memo (MIT-AI-306).

Mittal, S. and Chandrasekeran, B. (1984)

"PATREC : A Knowledge directed Database for a Diagnostic Expert System", *Computer, September 1984*, pp51-58.

Moore, R. (1980)

"Reasoning about Knowledge and Action", Menlo Park: Stanford Research Institute Artificial Intelligence Centre, Menlo Park, California, USA , (Technical note 191).

Newall, A. and Simon, H.A. (1963)

"GPS : A program which simulates Human thought", In: *Computers and Thought*; Edited by Edward A. Feigenbaum and John Feldman. New York: McGraw Hill.

Rescher, N. and Urquhart, A. (1971)

"*Temporal Logic*", (Library of Exact Philosophy 3), Wein, Austria: Springer Verlag.

Sacerdoti, E.D. (1973)

"Planning in a Hierarchy of Abstraction Spaces", In: *Advance papers of the Third International Joint Conference on Artificial Intelligence, August 1973, Stanford, California, USA*. Los Altos: William Kaufman Inc.

Sacerdoti, E.D. (1975)

"The Non-Linear Nature of Plans", In: *Proceedings of the Fourth International Joint Conference on Artificial Intelligence, September 1975, Tbilisi, USSR*. Los Altos: William Kaufman Inc, pp206 -214.

Sacerdoti, E.D. (1977)

"A Structure for Plans and Behaviour". Amsterdam: Elsevier-North Holland.

Srinivas, S. (1978)

"Error Recovery in Robots through Failure Analysis", In: *Proceeding of the AFIPS National Conference, August 1978,* Amsterdam: North-Holland, pp275-282.

Stefik, M.J. (1981a)

"Planning with Constraints", *Artificial Intelligence,* **16**, pp111-140.

Stefik, M.J. (1981b)

"Planning and Meta-Planning ", *Artificial Intelligence,* **16**, pp141-169.

Sussman, G.A. (1973)

"*A Computational Model of Skill Acquisition*", MIT: Artificial Intelligence Laboratory Memo No (AI-TR-299).

Tate, A. (1975)

"Interacting Goals and Their Uses", In: *Proceedings of the Fourth International Joint Conference on Artificial Intelligence, September 1975, Tbilisi, USSR*. Los Altos: William Kaufman Inc, pp234-238.

Tate, A. (1976)

"Project planning using a Hierarchical Non-Linear Planner", Edinburgh: Department of Artificial Intelligence, Edinburgh University, (Report 25).

Tate, A. (1984)

"Planning and Condition Monitoring in a FMS", In: *Proceedings of the International Conference on Flexible Automation Systems, July 1984, London*. London: Institute of Electrical Engineers, pp62-69.

Vere, S. (1983)

"Planning in Time : Windows and Durations for Activities and Goals", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAM-**5**, pp246-267.

Vilain, M. (1980)

"A System for Reasoning about Time", In: *Proceeding of the First National Conference on Artificial Intelligence, August 1980, Stanford, California*. Los Altos: William Kaufman Inc, pp212-218.

Waldinger, R. (1975)

"Achieving Several Goals Simultaneously", Menlo Park: Stanford Research Institute Artificial Intelligence Centre, Menlo Park, California, USA , (Technical note 107).

Wesson, R.D. (1977)

"Planning in the World of the Air Traffic Controller", In: *Proceedings of the Fifth International Joint Conference on Artificial Intelligence, August 1977, Cambridge, Massachusetts*. Los Altos: William Kaufman Inc, pp473-479.

Wilkens, D.E. (1983)

"Representation in a Domain Independent Planner", In: *Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 8-12 August 1983, Karlsruhe, West Germany*. Los Altos: William Kaufman Inc, pp733-744.

Wilkens, D.E. (1985)

"Recovering from Execution Errors in SIPE", *Computer Intelligence,* **1,** pp33-46.