EXCALIBUR: A Program for Planning and Reasoning with Processes

Brian Drabble *
Artificial Intelligence Applications Institute
University of Edinburgh
Edinburgh EH1 1HN

Revised February 1992

*The research reported here was undertaken by the author for his Ph.D. in Artificial Intelligence at Aston University, Birmingham, England

Abstract

This article describes research aimed at building a hierarchical partial order planner which is capable of interacting with a constantly changing world. The main aim is to verify a planning and execution strategy based on Qualitative Process Theory which allows a greater level of interaction between the planner and the real world than exists within current planners. A variety of techniques are described which allow the planner to create a model of the world in which plan failures can be analysed and faulty plans repaired. These techniques also allow the planner to react to changes in the world outside of the plan which it has been told previously to avoid happening eg. an explosion.

1 Introduction

For knowledge based planners to be useful to industry the plans they generate must be robust as well as easily updated to reflect the constantly changing requirements of any real world situations. At present knowledge based planning systems are only capable of understanding a planning domain at a very *superficial* level and as such the plans they generate can only be used as a *guide* to the people using the plan. This is because only a small amount of knowledge about the domain is given to the planner usually in the form of the action schemata. This lack of knowledge leads to the further problem of plan failure as events occur in the world of which the planner had no previous knowledge.

All planning systems 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. Reliability may be essential to avoid halts in production and the subsequent loss in profit. However, for the command and control of spacecraft a planning systems failure could mean the failure of an 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 AI 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 applications, above that of the traditional block stacking type, the problem of monitoring the plan as it is executed in the real world become of greater importance. The inability to monitor the execution of a plan and to react to any unforeseen changes are major deterrents to the integration of knowledge based planning techniques into industry. The problems which industry hopes to solve with planning techniques tend to have:

1. **external events** which occur in the real world over which the planner has no direct control. i.e. a new delivery of widgets arrives which it must remove from the loading bay.

2. **complex interactions** between the actions of the plan and the world. For example "boil some water and make a cup of coffee with it" requires

the planner to realise that the boiling water is a resource which it can create from executing certain actions in the real world.

3. **complex resource reasoning** in which resources change their attributes, can be substituted or shared, etc.

4. **continuous events** which occur over many actions and may or may not be due to the effects of actions executed within the plan. In a chemical plant a vat of chemicals will cool if no steps are take steps to avoid this.

This article describes a planning system EXCALIBUR [9, 6, 7] which can create and execute plans capable of interacting with a continuously changing world which is described using Qualitative Process Theory [11, 12, 13]. The techniques used here have been used for plan execution monitoring but we feel that they also have great potential in the areas of plan and schedule generation as well as schedule monitoring.

2 Background

The aims of this section are to show the advantages of using a qualitative representation scheme for plan execution and repair. The article first describes those attributes which are present in most qualitative reasoning schemes and then goes on to describe the scheme used in EXCALIBUR.

2.1 Motivation for using 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 plan execution monitoring. The process of reasoning about changes in the world requires the ability to reason in an expert fashion about time and processes. 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 planners cannot handle these kinds of reasoning, thus limiting their usefulness.

The impoverished semantics of the worlds used by these planners is not their only problem. Many treat the world as a static one, in that nothing changes without the planners either initiating it or knowing about the change. This means 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 "run" in the real world with any degree of certainty that it will accomplish the task for which its 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 execution of a plan. As a result a plan of action may no longer be capable of execution, resulting in an execution error. (For a more detailed discussion of execution monitoring and error analysis, see [8]).

One way of rectifying some of the problems in creating an internal world model which more 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 using partial descriptions of the world ([4], [16], [11, 12, 13]). However, the physical systems must be considered in isolation, as current systems do not allow intervention by agents. For example, a state-of-the-art qualitative reasoning system can predict the possible behaviours of a boiler, but cannot plan to intervene if the boiler is about the explode. Just as qualitative reasoning systems need to understand planning, so planners 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 of boiling, if only to know when to begin pouring the water into the cup.

Some research has been carried out in the area of integrating actions into qualitative simulations. One notable system is a version of Ken Forbus's QPE which integrates actions into qualitative process theory envisionments [14]. The work is similar to that reported here, but EXCALIBUR differs in several major ways:

1. **Efficient Planning**
   The planning carried out by EXCALIBUR makes use of a partial order network which the Forbus system cannot handle. That system assumes a single sequential order of actions. EXCALIBUR has been tested on a cooking domain first put forward in the SIPE system. This involves cooking a meal using 6 saucepans but with only four gas rings. The

Forbus system would be incapable of reasoning about the inherent parallelism available within the plan.

2. **Realistic Models of Operators**
The operator language (TF) used by EXCALIBUR allows the implicit hierarchy present in the domain to be represented and reasoned with. The language also allows time and resource information to be represented which is important when we consider execution monitoring and plan repair.

3. **Execution Monitoring**
The plans generated by EXCALIBUR are executed and if necessary repaired should a problem occur. This was never attempted in the Forbus system.

2.2 Qualitative Reasoning System used in EXCALIBUR

Qualitative reasoning systems are primarily interested in predicting the behaviour of physical systems in non-numeric terms, preserving all important behavioural distinctions of the actual systems, given only a structural description of it. The qualitative representation used in EXCALIBUR [6] is based on the qualitative process theory designed by Forbus ([12, 13]). The behaviour of a system is specified in the form of the creation and termination of processes, (such as heatflow, fluid flow, boiling, motion, etc.) which defines the changes they bring about in the world and the individuals effected. The structure of the domain is described using individuals [1] and their spatial relationships which can obviously change over time. Examples of individuals are containers, tanks, pipes, heat sources, fluid sources, etc. As qualitative reasoning systems are not interested in numbers they have no meaning, but relationships of the form "A equal B", "C equal Zero", "A greater than B" have as they will result in the termination or creation of processes. For example, the fluid flow process can only exist while there is a pressure difference across it, if the pressure becomes equal it stops. A boiling process will become active if the temperature of the liquid becomes greater than its boiling point. It is by analysing how processes are created and destroyed and how processes effect each other that a qualitative reasoning system can predict the behaviour of a physical system.

[1] The term individual is used to describe an object in qualitative process theory

2.3 Problems which Excalibur can solve

It is not only complex industrial planning problems which cause planners difficulty, but even simple every day problems such as "making a cup of coffee" are difficult to analyse. For example a plan to make a cup of coffee requires that we obtain boiling water but no single action can assert this effect. However, it can be achieved by a sub-plan to fill a kettle with water and then to heat it. This however requires the planner to know how to fill a kettle with water! The problems are compounded when the desired external events have unknown start times and durations and other external events occur which may interfere with the plan. For example, the time taken to boil a kettle of water is only known at plan time and failures can occur such as the gas being turned out or there being insufficient water in the kettle to make the coffee. Some domains such as process control, contain undesired events which the planner may be instructed to avoid, e.g. overflows, explosions, etc. For example, a plan to close all the valves of a boiler may well cause it to explode. This type of plan "failure" is more difficult to analyse than simple precondition failures as it requires the planner to reason about the causes of an undesired event and to modify an existing and correct plan.

EXCALIBUR has been designed to handle problems such as these which involve:

- events and actions which have unknown durations
- external events and/or their effects which are required within a plan
- events occurring external to the plan which are undesired.

Apart from the two domains described above, problems such as these can be found in areas such manufacturing, logistics, spacecraft command and control, and assembly tasks.

The rest of the article will describe the structure of the EXCALIBUR planner as well as the operator schemata and world descriptions which it uses. This article also describes in detail some problems which EXCALIBUR has successfully solved, which previous planners have been unable to handle. A fully detailed description of the EXCALIBUR system and the types of problem which it can handle, can be found in [9, 6].

3 Introduction to the system

The architecture for a system able to handle planning with processes, that is a complete Plan Management System (PMS), is presented in Figure 1. A PMS 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 replanning. The PMS is 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.

The motivation behind EXCALIBUR was to design a planning system which could plan to interact with events in the real world and to use these interactions within its plans. The term we use to describe this type of system is a Model Based Planner.

> Model based planning involves developing plans which can interact with external events and whose effects may be required by actions within the same plan. An execution time model of the real world is created to check that these externally satisfied preconditions will indeed come about. If any are endangered or any undesirable events are detected they are resolved by modifying the affected parts of the current plan.

In the current work the planner is represented by Tate's Nonlin ([17]) which can be regarded as having two major parts: a set of schemata which form a World Model, albeit a very limited one; and a Plan Generator. Nonlin's world model is limited in that it cannot model continuous events, make use of events occurring external to the plan, model resources and their uses during planning. 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 choice of Nonlin for the plan generation part of EXCALIBUR was taken early in the project. At that time no planning system other than Nonlin was available. Planners such as Deviser, SIPE and O-Plan were not available as general release versions until much later. This work was carried out at the University of Aston, long before the author moved to AIAI at Edinburgh. O-Plan was therefore not an option available. However, I have since found that by modifying the variable binding and matching routines of O-Plan to handle process requirements it would be easy to integrate O-Plan into the EXCALIBUR framework.

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 (i.e Nonlin) and the Monitor: it accepts a complete plan from the Planner and stores it. The plan is complete in the sense that it achieves the higher level parts of the problem which the planner is capable of handling. The lower level actions which monitor for plan success or failure are handled by a separate execution module which has passed to it a measure of the success of the plan fragment. For example, a cup filling to a required level is a measure of success [2].

The filling of a cup as will be used as an example of the interaction between the planner and the execution agent. The planner is capable of constructing a sequence of actions which can bring about the preconditions required for water to flow from a kettle into a cup [3]. The planner checks that the effect(s) required in the plan can be brought about from the processes created by the effects of the actions, e.g. if the plan requires boiling water and there is no heat flow process created the effect may prove a little difficult to achieve! These preconditions can be found from the external preconditions list defined within the fluid flow schema e.g. `aligned kettle cup`. The low level actions necessary to monitor that these preconditions as well as other derived (through the qualitative causal theory [4]) preconditions remain true is the responsibility of the execution agent. For example, moving the kettle to the cup will create *two* views, `aligned kettle cup` and `fluid-connection kettle cup`. Each of these are used as preconditions for the fluid flow process. It is the job of the execution monitor to calculate the dependencies between actions and to pass them across to the knowledge base. It is the function of the knowledge base to maintain these dependencies and to report any problems to the execution monitor. Should these problems cause plan failures then the execution monitor raises a re-plan request which is dealt with by the planner, in this case Nonlin. Not all dependency failures are in fact *problems*. For example, if the plan is to fill a bath to a required level and this is achieved before the tap is closed then this is not a plan failure and can be ignored even though it was flagged by

[2]This leads to a further problem concerning "degrees of success" i.e. is a half full cup satisfactory? This problem will be dealt with in a later section.

[3]The sequence is generated using standard partial order planning techniques found within Nonlin.

[4]The preconditions include the existence of the kettle and the cup, any contents they may have and the path between the kettle and the cup.

the knowledge base.

Plan actions are then passed to the monitor for execution, one at a time. The monitor asserts a block of actions in the knowledge base which is used as a context should any replanning be necessary. In the present system a window of 10 actions "downstream" of the one currently being executed is held within the execution monitor. The only exception to this is when an action has a precondition which will be satisfied by an effect brought about by one or more actions of the same plan i.e. the start of ending of a process. If the execution monitor has no information as to when this precondition can be satisfied no further actions will be sent to the execution monitor until the effect has been seen or a failure has been raised. The effect detection and error recovery will be discussed in later sections.

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).

3.1 The Process Reasoner

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 Process Reasoner has several functions which are:

- To take the description of a physical system along with the processes active within it and to create a process tree to reflect the possible behaviours of the system.
- To take information from sensors and user input about changes in the real world and to synchronise the internal world model with this information.
- To modify the process tree in the light of changes made by plan actions which cause the behaviour of the system to differ from that which was originally predicted.
- When a plan is suspended waiting for a process or a process effect, the Process reasoner must check that the condition or process waiting will at some point be achieved so as to avoid locking up the system.

Plan Management System schematic layout

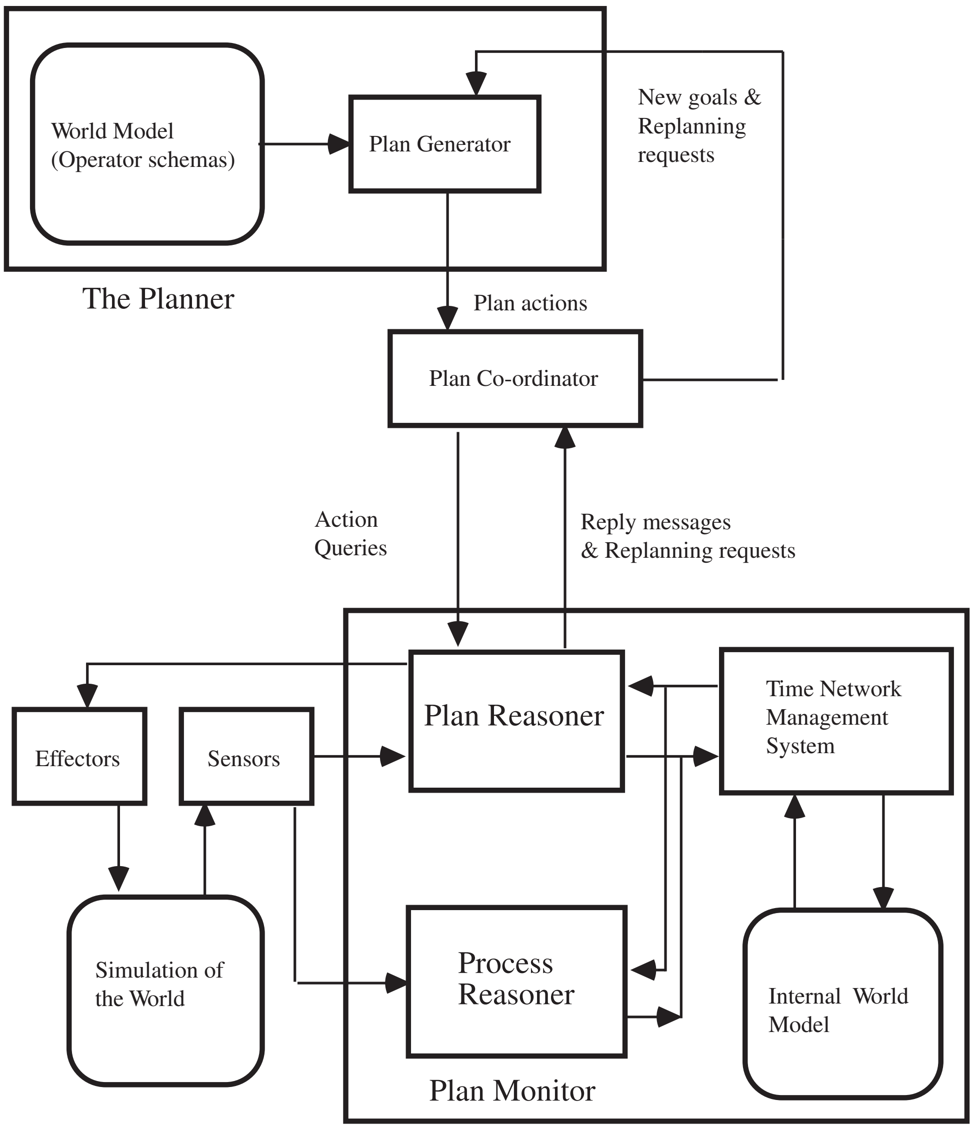

Figure 1: System Overview

- If a wait condition cannot be satisfied within a plan eg. you are waiting for a kettle to boil and the gas has gone out, then report to the Plan Reasoner those effects which are required in the plan for the wait condition to be valid.

In order to carry out the functions described above the process reasoner is required to maintain the *contexts* used in reasoning about the different types of change which can occur in the world. The contexts change dynamically as the structural make up of the domain changes. For example, when making a cup of coffee the kettle is moved around from one part of the kitchen to another. For example:

1. Firstly it is placed under the tap for filling
2. Secondly the kettle is placed on the gas to be boiled
3. Thirdly the kettle is taken to a cup in order to fill the cup.

In each of these situations the kettle (and its contents) will interact with other objects to be found in the vicinity. The changes which are brought (new processes and views created, existing processes and views which are terminated) are represented in a process tree, the structure of which will be described later in this section. As plan actions change the relationships between objects so new processes will be created and old processes destroyed. For example, while the kettle is under the tap we are concerned with fluid flows and water sources (as these are the processes and views instantiated) but as soon as the kettle is moved to the gas processes such as heatflow and boiling become of interest. The process reasoner creates contexts to break up the world into qualitatively interesting *pieces* called contexts. A new context is created under two different circumstances:

- **Statically**
  when an object is moved to another user defined structural context e.g. `under tap-1` to `on gas-1` where the sink and the oven are declared as separate contexts.
- **Dynamically**
  if the user does not wish to specify structural contexts, then the process reasoner will construct its own contexts according to a simple set of criterion:

1. Objects in the new location must not interact with any object in the previous location.
2. The new location must not be a subset [5] of the previous location. For example, moving a kettle from gas-1 to gas-2 of the same oven would not cause a new context to be generated.

If at any point information is derived or given to the process reasoner that a process is active between objects in previously separate contexts, then an instruction is sent to the knowledge base to collapse the contexts and to generate a single process tree.

The process tree reflects the possible behaviours of a physical system both in terms of changes brought about by processes and those brought about by plan actions. A process tree is generated in response to a new set of actions being passed to the monitor by the plan coordinator. The process tree reflects how the plan should affect the world and the processes and views which will be affected. It is this together with the dependency information from the plan which helps guide the execution monitor as to the success of a particular plan. The generation of the process tree is a two stage process. The process reasoner first has to decide which qualitative state reflects the state of the world at the point at which is it considering further envisionments. For example, in the coffee making problem the initial envisionment concerning pouring water from the tap to the kettle creates two states:

- one in which the water is flowing from the tap to the kettle (state 1)
- and the state which would result from the amount of water in the tap becoming zero [6] (state 2).

Assuming the kettle is filled to the correct level then the tap will be turned off and the plan coordinator having noted this fact will send a further set of actions to the monitor. This block of actions will move the kettle from the tap to the gas ring, light the gas and instruct the monitor to wait until the kettle has boiled. These actions will instantiate amongst other things a heat

[5] The domain can be expressed using a simple parts hierarchy e.g. tap part-of sink, gas-1 part-of oven-1

[6] possible if we have a very large kettle and a very small tank in the kitchen

flow process from the gas to the water via the kettle. The process reasoner must now decide what could result from this heatflow. The current process tree contains two states so the problem is to decide which one represents the state of the real world. The reason for state 2 existing is that the amount of water in the tap goes to zero and has the process reasoner has no knowledge to that effect then this state is ruled out. State 1 is therefore chosen as the point at which to start the envisionment. Providing there is at least one possible path through the tree which results in the change to the process or view required by the plan then no error is flagged by the process reasoner. In this case the result of the heatflow is either luke warm water or the water will be begin to boil as in required in the plan. The state in which the kettle contains luke warm water is viewed as a failure state as there is no path from it to a desired one. This allows the monitor to recognise a failure and provides a context in which to replan. In this case provide more heat to the kettle and a qualitative state from which to attach the resulting process.

Should the process tree fail to contain a state in which the desired process change occurs then the process reasoner flags an error to the Plan Reasoner. The message indicates the type of failure (process/view to terminate or start) and the process involved. It is then the function of the plan reasoner to extract the required process and checks the preconditions which the planner can achieve. For example, if a boiling process was required and none was noted then the plan reasoner would check the plan schemas for ones which generate a heatflow.

The process tree is generated using similar techniques to those in QPE. As a result the process tree created is a partial attainable action augmented envisionment. That is, it is an action augmented envisionment because it contains state transitions due to actions as well as dynamics. The tree is partial in the sense that only those states which are realisable are included. However, the main difference over the original QPE system is that the process tree also contains states which are inserted to accommodate changes brought about by actions which bring about qualitative states not represented by the dynamics of the domain. For example, in the coffee making example the plan contains steps to turn out the gas once the water has boiled. However, the process tree created by dealing only with the dynamics contains only one terminal state, namely the water in the kettle boiling away. To overcome this a new state is added to the boiling state to reflect the state in the world once the gas has been turned out. The state is tagged with both the dynamic reason for its existence i.e. `temperature gas < temperature` `water` as well as an action tag stating it was created to accommodate an action inspired change.

The states of the process tree represent generic qualitative states rather than episodes. This is because repeated actions are represented by cycles in the tree and *not* by repeated subsequences. In this representation the elements are viewed as qualitative states whose repeated occurrence define a history.

During the subsequent execution of the plan, the plan reasoner receives via simulated sensor input, from the knowledge base or from the user, information about the real world. As the real world changes the system moves from one qualitative state to the next providing the transition is valid. However, if information is input which either:

1. Counter to that which is expected.
   For example, (`temperature (contents kettle) decreasing`) [7] when the temperature is expected to increase at this point, then the process reasoner flags a possible error to the plan reasoner. The condition together with the process(es) which could be at fault i.e. in this case the loss of a heatflow process are passed across. If the loss of the process was under planner control i.e. it turned out the gas then it is ignored. However, if the failure is not of the planners' making the plan reasoner will at present enquire of the user if the preconditions are still present [8]. Once the failure(s) has been identified the plan reasoner sets about its task.

2. Consistent with the next state but the transition condition has not been met.
   For example, the user inputs (`contents kettle boiling`) before the system has noted that the temperature has reached 100 degree Celsius (assuming water). The process reasoner then moves to the next state and asserts the "missing" information in the knowledge base necessary for the process to occur or to be destroyed. The new information allows the process is then instantiated as its occurrence is reported to both the process reasoner and the plan reasoner.

[7] EXCALIBUR has a simple set of input routines which allow this type of input

[8] It is hoped in future to have a more elaborate physical system which EXCALIBUR can interrogate

3. Takes the process tree into a failure state.
   These are states (usually leaf nodes) marked by the process reasoner during the construction of the tree from which no path exists to a desirable state. The reason for failure and the process involved is passed to the plan reasoner for it to reason over.

Another function of the process reasoner is to report on states which contain process changes it has been told to avoid e.g. explosion, overflow, etc. If one of these processes is detected then a message is passed to the plan reasoner together with information as to how to stop it occurring. For example, an explosion occurs when (`pressure contents container > burst-pressure container`). Any process which is causing an increase in the pressure of the contents is a candidate for termination as are any processes which can be started so as to remove the problem e.g. in this case a gas-flow. In the current system the plan reasoner prefers to terminate current processes rather than to start counter-active ones.

3.1.1 Data Structures and Algorithms

The data structures used by the Process Reasoner use a method of context layering similar to that used by Nonlin and O-Plan [1] for its variables and values. This allows variables such as quantities (actual numeric values) and quantity spaces to be stored and retrieved efficient from their respective states. The states of the process tree are held within a cyclic graph to accommodate loops within the envisionment. Each of the states within the process tree holds:

1. The individuals taking part in processes active in that state
2. The processes active within the state
3. The influences both direct and indirect which each process adds to the state
4. The dynamic tag for the states existence and possibly an action tag to indicate the state was created to accommodate and actions execution.

New individuals which are created during an envisionment are labelled as `individual<n>` to distinguish them from named individuals. This makes the cycle detection slightly more complicated in that the matching function in the cycle detector must take into account that an individuals created in the same state will have different names. The planner must also describe the effects of actions in a way in which the monitor can use. For example, the planner has no idea of the name which the process reasoner will give to a new individual. As a result it must describe it in terms such as (`contents kettle`) which have meaning to the process reasoner.

The techniques used in the process reasoner allow failed plans to be analysed that could not normally be tested by classical planners which require all possible interactions with an action to be mentioned within the plan e.g. there is insufficient water in a kettle when making a cup of coffee. The process tree can also be used to detect errors outside of the plan itself, by detecting failures represented as a state within the tree. The detection is via reasoning about the requirements of the process itself, rather than simple problems arising with the preconditions as in classical planners. The execution monitor can therefore modify a plan to stop undesirable events occurring in the world. e.g. an explosion, overflow, etc. These undesirable events may occur as the side-effect of a plan or because of a change in the world not initiated by the planner.

3.2 The Time Network Mangement System

The EXCALIBUR world model (TNMS) is partitioned by the use of "contexts" - sets of facts, actions and processes which are inter-related by causal influences, where no such influences operate across contexts. The contexts are derived from the spatial and temporal bounds which can be placed on the individuals in the target domain. For example the context to reason about a sink is spatially isolated from a gas oven in the same room as these have no influence over each other. Changes in the world may cause contexts to require collapsing e.g. a new influence is asserted as occurring between previously isolated objects or dividing if sets of influences can be partitioned into subsets. Each fact within a particular context of the TNMS has associated with it an interval defined by two time points. Each of the time points has a range specified as a maximum and a minimum pair.

The basic representation of objects of the domain within the TNMS is as follows. An object within QPE has associated with it one or more views which define the "state" of the object. These views can be anything from "A is a piece of stuff", "A is a contained gas", "kettle is a container", etc. All of the views associated with a particular object are built into an hierarchy, e.g. "A is a piece of stuff", "A is a liquid" and "A is a contained liquid". Associated with each view are a number of attributes, e.g. pressure, volume, temperature, amount-of, etc. A fact associated with each of the attributes is asserted in the TNMS together with the basic description of the object.

```
(stuff-1 a piece-of-stuff) (has-quantity amount-of stuff-1)
(has-quantity temperature stuff-1) (has-quantity pressure stuff-1)
```

The attributes will have and end time of `pos-inf` indicating they exist for an unknown time into the future. Should the object disappear for example, the water drain completely from a tank, then all of the attribute facts will be clipped back to the time at which the object ceased to exist.

If any numerical information is asserted about a particular attribute then it is stored with the attribute of the object and an interval asserted within the TNMS. In the example below the attribute `amount-of` is originally specified as being of an `undefined` size and having and `undefined` change of direction (positive, negative, 0). Information was later added that from `pt3` to `pt4` the size was 12 units and it was increasing at a rate of 2 units.

```
(id-number stuff-1 a piece-of-stuff
                   (amount-of ((pt1 pt2 undef undef)(pt3 pt4 12 2)))
                   (temperature ((pt5 pt6 undef undef)))
                      .
                      .
                   (link-field (id-number pt16 pt17)
                               (id-number pt202 pt203)))
.
.
(id-number stuff-1 a liquid
                   (link-field (id-number pt26 pt27)
.
.
(id-number stuff-1 a contained-liquid
                   (level ((pt12 pt13 undef -)))
                   (container ((pt43 pt44 kettle))))
```

The hierarchy uses inheritance to reduce the amount of information stored at each level. For example, the attributes of a liquid are no different from those of a piece of stuff, but there are two extra attributes of a contained liquid i.e. level and container (i.e. the object which contains it). If during the plan A becomes a solid then a new link will be added to the “A is a piece of stuff” view to point at the “A is a solid view”. The clipping mechanism of the TNMS will truncate the end time of the previous view e.g. “A is a liquid” to the start time of the new view. The links in this hierarchy have associated with them time intervals during which the TNMS and process reasoner may transverse them. The interval is associated with the parent view. For example, the link from “A is a liquid” to “A is a contained liquid” can only be transversed while “A is a liquid”. This avoids information being processed which has no bearing on the current problem. For example if A is now a solid then there is no point in retrieving information about the level of A when it was a contained liquid. Thus each level maintains a history of the changes it has been through and the times as which they took place. This allows the TNMS to retrieve and update information quickly in order to maintain the integrity of the knowledge base.

Maintenance of these contexts requires extensive use of qualitative reasoning and a truth maintenance mechanism based on Doyle's system [5]. These have been modified to handle qualitative processes and views. The TNMS is similar to Dean's TMM [3], but has been modified to handle both processes and execution occurring over time. The functions of the TNMS are as follows:

- The TNMS must maintain the integrity of the temporal network contained in the world model in that no two tokens asserting contradictory information are allowed to overlap. This becomes quite complicated when dealing with facts such as “A is a liquid” “A is decreasing”, etc. At present the system uses list of negatives for matching against. For example, the opposites of decreasing are increasing and steady and the opposites of liquid are gas and solid. The clipping is done in accordance with the methods outlined in Dean's TMM though the justifications for doing so have been extended. The justification for a process or view can be in terms of:

  1. actual numerical values which allow a relationship to be derived for example, (`a > b`)

2. a relationship which is asserted between two quantities, for example (`A greater than B`)

If new numerical information is asserted it is added to the table described above and a new interval generated which clips the previously generated numeric interval. A check is then made for any processes or views which are reliant on this quantity. For example, a fluid flow can only exist for as long as the source pressure is greater than the destination pressure. These processes are then passed to a checking routine (described below) for further analysis. A further check is made for relationship predicates which may now assert contradictory information to the new numerical data. This is involves retrieving the predicates which are tagged to the numerical field and checking these against the relationships. Any which are now invalid are clipped to the required point.

In the case of a relationship being asserted the numerical information is checked and the numerical values aligned according to the direction of change. For example, if the relationship asserted is (`a = b`) and the direction of change is A is decreasing and B increasing then a value is calculated from the previous information, and the time difference until now. The relationship is then asserted in the TNMS as a fact. As in the above cases processes which are reliant on one or more of these quantities are checked for consistency. It is assumed the relationship has already been validated by the process reasoner as described in the previous section.

Using these two techniques a justification for a process can be built up out of temporal "pieces" of numerical information and the relationships between quantities in the domain rather than the signal interval which was the case in the TMM. The idea is similar to the idea of concise intervals which used in the system developed Williams [19]. His ideas have been extended to allows intervals to be made up of both relationship information and actual numerical information when it is available.

- When a change is made to a set of facts or new information asserted in the TNMS it must ensure that no action or process is believed beyond the range of the facts used to justify it. This done by checking the start and end points of the action or process against the end or start points of the preconditions respectively. In the case of an action this may result in it being unable to execute due to the interval becoming smaller than that which is needed. At the point the TNMS sends back a message to the module making the change indicating the action and precondition which is at fault. In the case of a process any change in the duration is reported to both the plan reasoner and process reasoner. The process may be a current one which is terminated by an action (as was described in the process reasoner) or a future one upon which part of the plan depends. It is then up to the reasoning module to decide whether the change in the process was required and if so to initiate the required replanning to restore it to its required value.

The TNMS must also check in the case of a process that its effects must be only be believed while the process is active. This means the TNMS must ensure the end points of the process are coincident with the end points of all of its effects. A minor change to this policy has to be used when any new objects in the domain are created. For example, if we boil water in a sealed container then we create an new object namely steam. If the boiling process stops then the steam should persist and not suddenly disappear. The TNMS does not truncate such new objects or their effects. In the case of an action the effects persist after the end of the action unlike a process.

- It must be able to respond to queries from the process reasoner, plan reasoner and plan coordinator concerning temporal relationships among facts and actions so that other information may be asserted in the knowledge base. When asserting the actions of a plan in the TNMS or checking is a process could be active, intervals need to be found of the required duration during which the preconditions hold. By using the context mechanism of the TNMS this search process does not become an excessive overhead.

Each process, action or fact asserted in the knowledge base has associated with it a *token* type. The classification of the tokens is as follows:

- 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 infinite in duration. For example, "open the valve" is an action event; "water flows" is a process event.

- A fact describes the results or preconditions of an event. For example "the door is open" is a fact whereas "open the door" is an event.

This token definition allows the TNMS to discern between token types and to represent clearly the distinctions between events and the changes they bring about. For example, the action "open door" leads to the process "in motion door" which leads to the fact "door open".

However, this scheme on its own is too simple to work on real world situations. For example, an action to lift a block may have the preconditions "block held" and "block on table". As soon as we execute the action the precondition "on table" is no longer true but the action still continues. However, if the precondition "holding block" were to fail then so would the action. This justification scheme is incapable of discerning the difference between preconditions required to *maintain* an action's execution and those merely required to *initiate* it. The TNMS solves this problem by checking a precondition failure against the process tree created by the process reasoner. In this example, the failure of the precondition "block on table" would not cause the failure of the motion process created as a side effect of the action. As the effect of the action is to move the object from one place to another and the motion process is unaffected the failure can be ignored. However, if the precondition "holding block" were to fail, then so would the motion process thus indicating that the plan has failed and needs to be repaired. This means that the plan generation phase does not need to deal with complex precondition structures to identify the different types of precondition.

3.3 The Plan Reasoner

The function of the plan reasoner is deal with changes to the plan during its execution. When the plan coordinator is given a plan fragment to deal with its asserts the action in the TNMS with proposed start times. During the execution of the plan this proposed schedule may require altering for a variety of reasons.

1. Actions are rescheduled to begin later or earlier than was expected in the original schedule. An action can start later due to a preceding action taking longer than was expected or new plan fragment was integrated into the plan thus requiring the rest of the plan to be resched-

uled. An action can be begin earlier than expected due to sections of the plan not being required due to fortuitous circumstances.

2. A process or external event with which the plan is required to interact with could occur at a different time thus requiring the plan to be rescheduled and or patched.

3. An external event which the planner was instructed to avoid such as an overflow, explosion, etc may be predicted and hence a new plan fragment is required in order to stop it occurring.

The plan reasoner has several tactics available to it and these include:

1. re-execution of a single action in the plan should its preconditions be true at the point of failure.

2. suggesting plan patches to overcome precondition failures in plan actions as well as processes should the above strategy fail. The plan patches are generated by using the existing plan and refitting parts of is for use as a repair plan. This is a two stage process as follows:

   (a) To edit the Nonlin Task Formalism TF schemata to only achieve those actions which are required in a patch plan by modifying when necessary the precondition satisfaction and sequencing information of the TF schemata. This is described in more detail in below. The plan schemas and the goals of the patch are then sent to the plan coordinator which requests a plan from the planner, in this case Nonlin. Once a plan has been generated is it passed back to the plan coordinator.

   (b) To accept a patch plan from the Co-ordinator and to integrate it into an existing plan without causing any further errors.

The failure of a plan may be caused by a variety of different reasons. For example, an action may not provide its required effect, an action may have an unknown side effect, a resource may not be available, etc. To overcome this EXCALIBUR uses a variety of different strategies from simply re-executing a failed action to dynamically creating its own operator schemata, tailored to the specific failure context from which repair plans can be generated. The operators of the EXCALIBUR are defined using the Task Formalism (TF) description language ([17]). The new schemata are created using a technique called Dynamic Schema Generation (DSG), which allows the planner to take the operator schemata defined by the user and to *edit* them to create new schemata capable of solving the problem with which it is faced. The plan reasoner can only make check that all of the effects of the actions in the required plan have some way of being achieved. It is the job of the planner to generate a plan which is then sent to the plan reasoner for further consideration. This can be achieved quite easily by using the internal information found within schemata. This provides information on:

- The sub-actions into which a schema can be decomposed.
- The explicit ordering of the sub-actions within the schema.
- The methods to be used to satisfy the preconditions of the sub-actions of the schema.

The new schemata take into account those actions which have already been executed and those preconditions which need to be satisfied by an alternative method. For example, in the plan to make cup of coffee a failure may occur in which there is insufficient water in the kettle and this is discovered when it is poured into the cup. The repair plan to take the kettle and fill it should not include the steps to pick up the kettle as the agent must already have done that in order to have tried to pour the water in the cup. However all actions which pick up the kettle should not be removed from the schemas as there would then be know way of moving the kettle from the sink to the gas ring!

The plan reasoner sorts this problem out by using the dependency information embedded within the original plan. If an action requiring a step which is already true is in the plan and a step which negates it is still in then the action must stay.

The condition achievement process used by Nonlin allows the user to embed domain information in the schemas as to how a particular condition should be tied up. For example two commonly used types are:

1. Supervised: informs the planner that a condition on a particular action can be achieved from an affect asserted by an action in the *same* schema

2. Unsupervised: informs the planner that a particular condition will have to be tied up from an effect asserted by a action outside of this schema.

Information can further be provided to the planner in the form of "initial conditions, i.e. those effects which are present in the real world at the start of planning. The process for generating a new set of plan schemas can be defined as follows:

1. the point of failure becomes the context in which the replanning will take place. Working from the start of the plan and moving towards the action which failed remove any action whose affects are the same as the point of failure.
2. remove any effect from the failure context which is negated by the effect of an action on the path back.
3. once the actions required have been identified modify the sequence information to reflect the actions which have been removed.
4. modify the condition typing information in accordance with the actions removed. For example, an action which was kept as part of the repair plan which required a SUPERVISED precondition in the original plan and that action has been removed should now have a UNSUPERVISED precondition

These new schemas are sent to the plan coordinator which in turn sends it to the planner (i.e. Nonlin) which generates the repair plan. This is then passed back the plan reasoner via the plan coordinator. The actions of the repair plan are renumbered so as to reflect they are part of the original plan. The plan reasoner then integrates the new plan fragment into the TNMS and has it checked by the process reasoner. Any new plan problems which occur are dealt with in the same way and as a result several plan repairs may be necessary for the original problem to be overcome.

Using this scheme means only those actions which are required are re-executed in the repair plan. The method must also be sensitive to the processes and views which it will create as a side effect and also to any processes which may be required in the repair plan. A simple example of this is the problem in which we attempt to make a cup of coffee and there is insufficient water in the kettle. EXCALIBUR can solve this problem by obtaining more of the resource i.e. water. However, if the plan does not contain the actions to boil the water then the plan will fail. This is noted by EXCALIBUR and the repair plan is amended to include the actions necessary to boil the water before it is placed in the cup. (For a more complete description of this problem refer to Section 6.) A simple example of this is the "tea bag" problem in which we try to obtain a cup of tea. On entering the kitchen we have a plan to make a cup of tea, but we find a cup of tea already on the work surface which only requires some milk to be added. Alternatively we may find the water in the kettle is boiling so all we have to do is add a tea bag to the cup, pour in the water and add the milk. This shows that plans in general need to be restarted from various points to take advantage of changes in the real world.

4 Benefits and limitations of the PMS

The following section aims to show the benefits and limitations which a PMS such as EXCALIBUR has.

4.1 Reasoning about action and effects

This architecture gives EXCALIBUR the ability to deal with types of problems outlined earlier which can be found in most industrial domains. Events and changes can occur in the world which are not initiated by the planner, but to which the planner can re-act. This allows the planner to reason about the side effects of its actions and to note any which may cause interactions with the plan. Suppose a plan calls for the action "open valve-1a". This action has the *direct* consequence of changing the state of the valve: if it were closed before (a precondition of the action), it will be open afterwards. However if the valve is connected to a container holding liquid, then opening the valve may cause other *indirect* changes in the world, e.g. the amount of liquid in the container may change. In either case there are qualitatively understandable influences between objects which are adjacent in some sense [2]: physical force applied to the valve causes a movement in its parts; a liquid which is in contact with the opening in the container will flow out of it. For example suppose the PMS has the goal of completely filling container2 with water. Container2 is currently empty, but is connected by a pipe to container1, which does contain water. There is a valve in the pipe, currently closed. The simple plan generated involves opening the valve. The process which occurs can easily be described in purely qualitative terms, i.e. the level of the water in container1 should decrease and the level of container2 should increase. Based on the appropriate qualitative description schemata the system can reason about what changes will take place within this process, e.g. while the pressure of container1 > pressure of container2 the process will be active, but at the boundary condition where the pressures are the same there will be a change in the process, i.e. it will stop. It is the changes in influences and the reaching of boundary conditions which the system must monitor in order to synchronise its internal world model (the predicted futures) with the actual world. Furthermore the ability to reason in this way enables error analysis to be performed. If the water stops flowing before container2 is completely full, so that the goal has not been achieved, the system will be able to reason about the possible reasons: the valve may have been closed, perhaps by another agent; there was insufficient water in container1; or the level of the water may have equalised before container2 was full because there was insufficient water in container1. Such error analysis can be used as the basis of re-planning.

4.2 Resource Reasoning

The use of qualitative reasoning provides a greater level of resource description than has been incorporated into previous planning systems. Resources can be reasoned about at their *view* [9] level which allows a *common* representation of resources which can either be fixed, shared or substitutable. The resource reasoning within EXCALIBUR also allows the definition of a new type of resource namely *transformable.* This is a resource which can be created from another resource or can be created by changing the attributes of an already existing resource. For example, we can create a new resource by transferring part of an existing resource to a new location, e.g. pump water into an empty container. Alternatively we can take a resource and changes its attributes by means of a process, e.g. water can be heated creating a new resource "boiling water". The planner can thus create new resources and transform known resources for use within its own plans.

[9] A view is a qualitative description of the state of an object, e.g. a liquid, a contained liquid, etc.

```
[process Fluid_flow
[[*src a contained liquid][*dst a contained liquid]
 [*path a fluid path][fluid_connection *src *dst *path]]
[[aligned *path]]
[[pressure *src > pressure *dst]]
[[flow_rate a quantity]
 [flow_rate Q+ [pressure *src - pressure *dst]]]
[[I + amount_of *dst flow_rate *src]
 [I - amount_of *src flow_rate *src]]
[fluid_flow *src *dst *path]
]
```

Figure 2: Process Schema

5 Operators and Schemata

5.1 Schemata

The description of the world which the Plan Management System (PMS) uses is provided by a set of process and view schema definitions. Processes which were defined earlier describe changes in the world and views describe the characteristics of an object which can change over time. For example, an object may be defined as a liquid if its temperature is between certain values and a contained liquid if it is then placed inside a container. If the water were to freeze then the object would change its characteristics and become a contained solid. The schemata are used to find active processes and views in the knowledge base by instantiating its preconditions with tokens found there and by the Process Reasoner when investigating the behaviour of a physical system. An example of a process schema is outlined in Figure 2.

5.2 Operators

An operator defines an action which the planner can perform and the direct effects which it can bring about. As described in the introduction an action may also have indirect effects as in the make coffee example. This example clearly shows that in any realistic domain, actions may be required to bring about an event external to the plan, one of whose effects is required by another action of the same plan. In the make coffee example there is no action which has the direct effect "water boiling" but this is required in the plan and can be brought about in the real world by executing a certain set of actions. The Nonlin TF description language has been modified so that the plan can handle interactions with external events. This is achieved by the use of three synchronising primitives, `Waitfor,` `Waitbegin` and `Waitend`.

Waitfor This indicates to the Plan Co-ordinator that it should wait until a condition is true. For example: `Waitfor level Con_1 = level Con_2`

Waitbegin This indicates to the Co-ordinator that it should wait until a certain process or view has begun. For example: `Waitbegin fluid_flow Con_1 Con_2 Pipe_1`

Waitend This indicates to the Co-ordinator that it should wait until a certain process or view has finished. For example: `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 [10] by the actions which follow them in the schema and which require the directives' outcome as preconditions. 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 making a cup of coffee, we boil "the water in the kettle". However, if pouring the water in to the kettle is part of the plan, the name the TNMS will give the 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 relationship with other individuals, For example, `Waitfor boiling contents kettle` can be used in the plan and the Plan Co-ordinator 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`. An example of the use of such primitives to make a cup of coffee is outlined in Figure 3 and Figure 4

[10] A supervised precondition in Nonlin is achieved by an action within the *same* schema whereas an unsupervised action is achieved by an action in another schema

```
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
   vars         con undef
                cup undef;
end;
```

Figure 3: Top level schema

```
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 {@*con held} 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 @*cup from @*con}
   expansion    1 action {grasp @*con}
                2 action {pickup @*con}
                3 action {pour contents of @*con into @*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 4: Schemata to fill the kettle and cup

6 Examples of the use of Excalibur

The following section aims to show how the EXCALIBUR system is capable of carrying out planning and execution monitoring using a qualitative process model of the real world. During this paper the example of `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 will serve as an example, but the system has also been tested on other domains such as house building and simple process control problems.

In the making coffee example the basic plan is to:

- take a kettle which can be found in the kitchen and fill it to a required level with water from a tap.
- boil the water placed inside the kettle, and finally
- pour the water, once it has begun to boil into a cup which has previously had coffee granules put into it.

The plan contains actions to move the kettle to the tap, and then to the gas and finally to the cup and to wait for various processes and effects.

- The water reaching the required level in the kettle.
- The water to begin boiling.
- The water to reach the required level in the cup.

As the actions of the plan change the spatial relations between objects, the histories which describe various episodes will be created and destroyed. As described earlier it is the function of the process reasoner to analyse the process tree it has created in the light of changes made by the planner. This may mean augmenting the process tree with new states or creating a completely new process tree.

The kitchen in which the agent is placed consists of a set of objects as follows: a kettle, a cup, sink with taps, a tank of water connected to the taps, a jar of coffee granules and a gas oven with four rings. For each of the objects in the domain a set of qualitative attributes is defined. In the case of the tank of water the contents of which have been named `stuff-1` in this example the attributes are as follows:

```
(id-number stuff-1 a piece-of-stuff
                   (amount-of ((pt1 pt2 undef undef)))
                   (temperature ((pt3 pt4 undef undef)))
                   (pressure ((pt5 pt6 undef undef)))
                   (heat ((pt7 pt8 undef undef)))
                   (container ((pt9 pt10 tank1)))
                   (link-field ()))
```

In the case of kettle the following has been defined:

```
(id-number kettle an object
                   (volume ((pt1 pt2 undef undef)))
                   (temperature ((pt3 pt4 undef undef)))
                   (pressure ((pt5 pt6 undef undef)))
                   (heat ((pt7 pt8 undef undef)))
                   (contents ((pt9 pt10 none)))
                   (link-field ()))
```

A schema for each of the objects is defined and a fact asserted in the TNMS for each numerical interval and for each attribute which the object possesses. Other information which the agent has include:

1. a set of process schemas which define events such as boiling, heating, flowing, cooling and motion
2. a set of views which define liquid, gas, contained-liquid, fluid-aligned and heat-aligned.

Each of these schemas contains the direct and indirect influences which are present while it is active.

From this information the TNMS can instantiate various views from the objects in the world. These include that the kettle is a container, `stuff-1` is a liquid and from that it is a contained-liquid, taps are sources of water, gas rings are sources of heat, etc. These are entered in the table described above and further facts are asserted in the TNMS. Each of the views has associated with it a set of justifications which only allow it to be believed over a certain interval.

A schedule for the actions of the plan is created by the plan coordinator which asserts the actions in the TNMS. This process stops when the plan coordinator is unable to decide the duration of the process `waitfor level contents kettle = fill level kettle` required to fill the kettle [11]. At this point the plan is suspended and the process reasoner called to check that that required `waitfor level contents kettle = fill level kettle` is valid. The process reasoner creates a process tree which predicts a fluid flow from the tap to the kettle which has only one future state in which the water in the tap can drain away. The suspension is allowed by the process reasoner because the fluid flow will create a contained liquid in the kettle whose amount will increase as it is the destination of the fluid flow. As the level of a contained liquid is proportional to its amount, the level of the water in the kettle will thus rise.

The actions of the plan are then executed in the order of the schedule. The plan will eventually reach the stage of waiting for the water in the kettle to reach the required level. Once the fluid flow has reached the required level a message is sent to the process reasoner. This information allows the process reasoner to unsuspend the plan and it sends a message to the plan coordinator to inform it that it can assert further actions in the TNMS. This is does but again has to suspend the plan when the time to boil the kettle cannot be calculated. The effects of the further actions causes the justifications of certain views and processes such as the fluid-aligned and fluid-flow to fail. The fluid-aligned view fails because the token in the TNMS asserting "under kettle tap" is clipped from pos-inf to the point at which the move action which takes the kettle from the sink to the oven is executed. In turn this view was used in the justification of the fluid-flow along with the facts"tap open" and "contains tap stuff-1". As a result the fluid-flow is clipped to the point at which the fluid-alignment fails. These failures are reported to the process reasoner for analysis. The termination of the process is not a plan failure as the condition it was required to bring about has been met, so the loss of the process can be ignored.

The next part of the plan involves creating the conditions necessary to boil the water which is now inside the kettle. The effects of the actions asserted allow the TNMS to discover a heat-aligned view and a heat-flow process. These are reported to the process reasoner and checked against

[11] this is due to it having no information concerning the size of the kettle or the flow rate

its process tree. The process tree for this section of the plan has to be anchored in a particular state. This process was described in the Section 3.1. concerning the process reasoner. As the process tree created contains a state in which the water in the kettle is boiling the `Waitbegin` is allowed to proceed. The process tree also contains states which if reached would cause the plan to fail. For example the temperature of the source may not be high enough for the water to boil or the water may boil away before the system has chance to pour it into the cup. If either of these states is reached then the system knows to replan to achieve the boiling process again. When the water begins to boil, the system will be informed of this via sensors so that the process reasoner can update the process tree to reflect the change in the world. In this way the internal world model can track changes going on in the external world.

The final part of the problem is to analyse the fluid flow once the action to pour the coffee into the cup has begun which has a similar structure to the process tree constructed for the fluid flow from the tap to the kettle. The process tree which is constructed for this entire problem is described in Figure 5. The fields marked *PS*, *IS* and *LH* indicate the processes active, the individuals they act upon and the change which brings the "state" about respectively. The individual "Fpath" refers to a fluid path through which a liquid can flow, in this case the space between the kettle and the cup. (Various views such as "aligned kettle cup" are created and monitored by EXCALIBUR to maintain the existence of this path). The individuals such as "Individual1", "Individual2", etc are created by EXCALIBUR when new entities are created in the world. For example the water in the kettle (Individual1) or the water in the cup (Individual7).

7 Replanning and Error Analysis

The process tree can be used to analyse plan failures not specifically tested for in the preconditions of the plan actions. For example, if someone were to turn out the gas while we are waiting for the kettle to boil then this would be recognised as a plan failure both in terms of a precondition failure i.e. the justification for the action has a precondition finishing before the end of the action and by the modification made to the process tree which now excludes the possibility that the water will boil. This failure would cause a patch to be generated and integrated into the plan. In the current system this

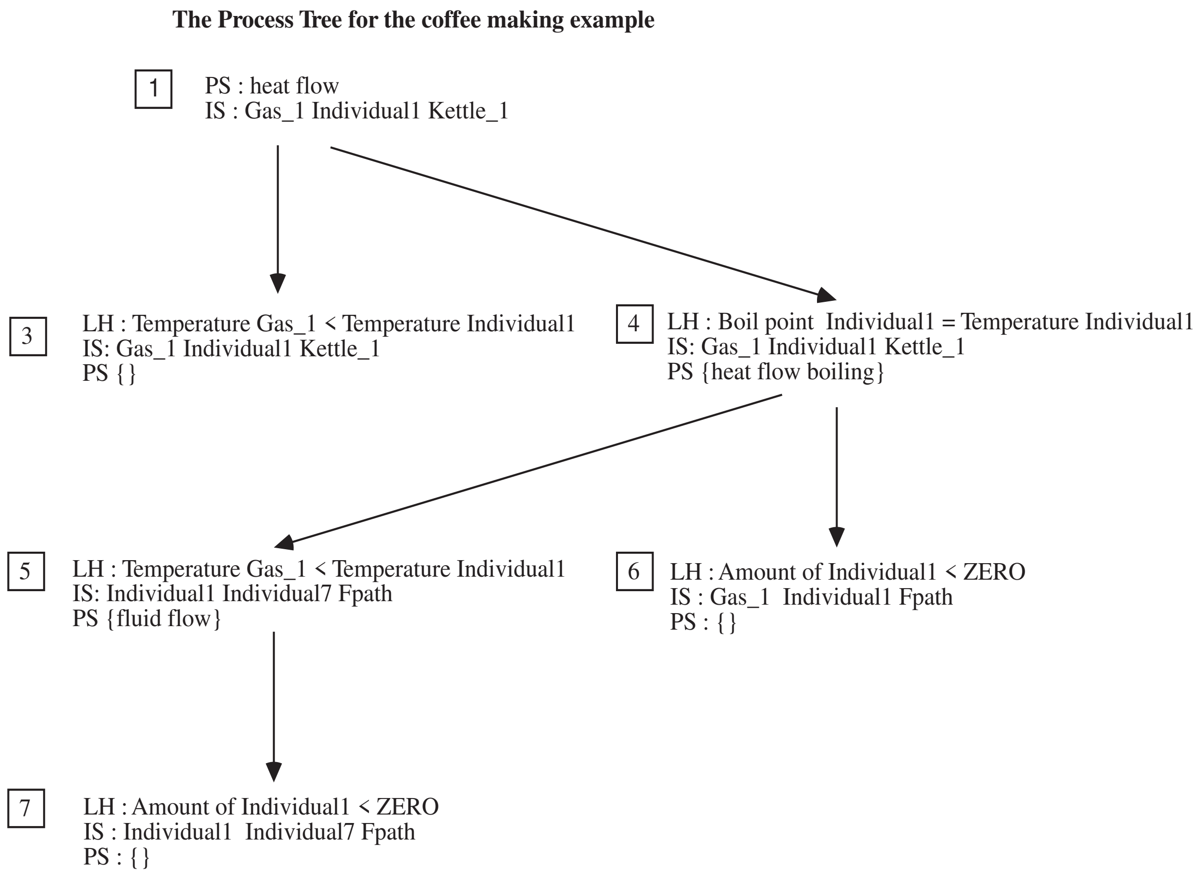

Figure 5: Process Tree for the make coffee example

involves re-executing the same action sequence to turn on the gas as before. However the representation system does allow other action sequences to be executed which repair the plan and do not exclude the possibility of the kettle boiling.

However the process tree is more useful when detecting errors outside of the plan. For example, when pouring the water from the kettle into the cup there is the possibility that the kettle may not contain enough water. During the execution of the plan the `amount-of` field of the individual was set to zero which caused the justification of the liquid view to fail. As a result the contained-liquid view failed and as a result the fluid-flow process failed. As a result the process reasoner reported to the plan reasoner that the plan had now failed as the wait condition could not be satisfied. This type of failure can be dealt with by the system since the loss of the water is represented as a state within the process tree. Reasoning about this type of failure involves reasoning about the requirements of the processes and views rather than as a problem with a precondition justification. In this case the plan requires to increase the `amount-of` water in the kettle to do this requires it to be the destination of a fluid flow. The water must then be boiled and moved across to the cup. The original plan is reused to provide a patch to solve these particular problems taking care not to re-execute and actions which are not required. The method to used to generate the patch is described in Section 3.3.

This second example shows how EXCALIBUR is capable of making use of the process tree to avoid failure situations it has been instructed to avoid, e.g. an explosion or overflow. The system can monitor for these types of event and if a qualitative state which contains one of them is predicted in the next step in the process tree can patch an existing plan thus avoiding the situation. This was tested on a simple process control application in which the plan consisted of opening and closing valves in a set sequence to allow water to flow into a boiler, which was then heated and the resulting steam used to drive a turbine Figure 6. The basic plan is to

- allow water flow from con-1 into con-2 and to wait until the water pressures have equalised.
- The water within con-2 is then heated to form steam which is held within con-2 until a specified pressure value is reached.
- When the specified value is reached the steam is released and is allowed

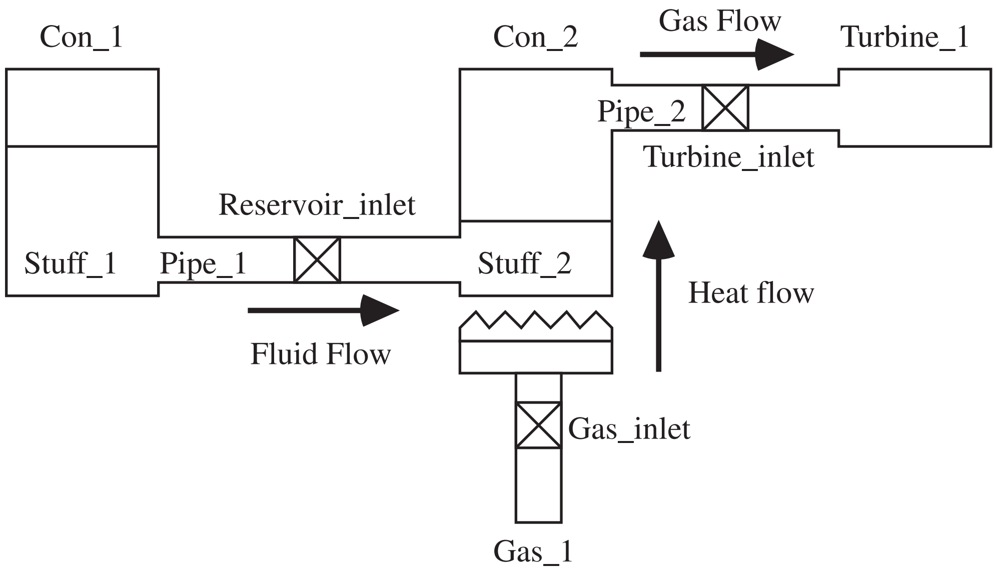

Figure 6: Process Plant Diagram

to flow into the turbine.

The schemata which were used to generate the process plan are outlined in Figure 7 and Figure 8. In this particular problem the plan generated is linear but other problems have been tested using non-linear, partial orders.

As in the coffee making example the plan reasoner and process reasoner work together to solve the problem. In this example an explosion is predicted i.e. when the plan is waiting for the pressure in con-2 to reach 100 but the process reasoner makes no request to the plan reasoner to stop it as the plan contains actions which will stop the explosion from happening. The process reasoner identifies which plan dependent effects are required to instantiate the processes and views which will cause the explosion. In this case its the facts that all of the valves are closed and thus the boiler is viewed as a sealed container. The heatflow and boiling processes give rise to a new individual, namely the steam which causes the pressure to rise. The process reasoner then searchs the plan form the current point to the end for an action(s) which have effects that are opposite to those required for the processes causing the problems. In this case it finds the open turbine-inlet actions which change the view of the boiler from a sealed-container to an open-container. As such

```
actschema generate_power
    pattern {generate power}
    expansion   1   action {open reservoir_inlet}
                2   action {waitfor {pressure contents con_1 = pressure
                             contents con_2}}
                3   action {close reservoir_inlet}
                4   action {open gas_inlet}
                5   action {turn on gas_1}
                6   action {waitbegin {boiling contents con_2}}
                7   action {waitfor {pressure contents con_2 = 100}}
                8   action {open turbine_inlet}
                9   action {waitfor {1000 sec}}
                10  action {turn off gas_1}
                11  action {close gas_inlet}
                12  action {close turbine_inlet}
    orderings   sequence 1 to 12
    conditions  supervised {achieve condition} at 3 from 2
                supervised {gas_1 alight} at 6 from 5
                supervised {achieve process} at 7 from 6
                supervised {achieve condition} at 8 from 9
                supervised {achieve condition} at 10 from 9
                supervised not {gas_1 alight} at 11 from 10

                unsupervised {reservoir_inlet open} at 2
                unsupervised not {reservoir_inlet open} at 4
                unsupervised {gas_inlet open} at 5
                unsupervised {turbine_inlet open} at 9
                unsupervised not {gas_inlet open} at 12
end;
```

Figure 7: Top level schema for the generate power problem

```
actschema open_valve
    pattern {open @*valve}
    expansion   1  action {go to @*valve}
                2  action {grasp @*valve}
                3  action {turn @*valve to open}
                4  action {waitend {motion @*valve}}
                5  action {waitfor {@*valve open}}
                6  action {ungrasp @*valve}
    orderings   sequence 1 to 6
    conditions  supervised {at @*valve} at 2 from 1
                supervised {holding @*valve} at 3 from 2
                supervised {@*valve open} at 4 from 3
                supervised {achieve process} at 5 from 4
                supervised {achieve condition} at 6 from 5
    vars        valve undef;

end;

actschema close_valve
    pattern {close @*valve}
    expansion   1  action {go to @*valve}
                2  action {grasp @*valve}
                3  action {turn @*valve to closed}
                4  action {waitend {motion @*valve}}
                5  action {waitfor {@*valve closed}}
                6  action {ungrasp @*valve}
    orderings   sequence 1 to 6
    conditions  supervised {at @*valve} at 2 from 1
                supervised {holding @*valve} at 3 from 2
                supervised not {@*valve open} at 4 from 3
                supervised {achieve process} at 5 from 4
                supervised {achieve condition} at 6 from 5
    vars        valve undef;
end;
```

Figure 8: Process Plan Schemata

the planner should take no action as the plan already contains the correct steps.

This simple scenario allows us to test a variety of other failure conditions where plan patching is required. Examples of failures are:

1. insufficient resources such as the water in con-2, the heat of the flame, etc

2. the valves failing to reach the desired position. The movement of the valves can be represented as motion process which allows the continuous nature of a valve state change to be represented. Both the schemata used to represent valves opening and closing have a process check that process has finished and that the valve has reached it required position. Failure in either of these checks allows the process reasoner to request a patch plan from the plan reasoner.

Apart from failures involving the preconditions of plan actions EXCALIBUR is capable of patching plans to avoid undesirable changes occurring in the world. To test EXCALIBUR'S ability to detect and avoid an *explosion* the following scenario was used. Once the steam in con-2 had reached the required pressure and was flowing into the turbine the process reasoner was informed that all the valves of the boiler were closed:

```
{turbine-inlet closed start  {19:10.00}
                      finish {19:10.30 19:10.32}}

{reservoir-inlet closed start  {19:10.00}
                        finish {19:10.30 19:10.32}}
```

This causes the justification for the Con-2 being an open-container to fail and be clipped. A new view of Con-2 as a sealed-container is instantiated by the TNMS and a message is sent ot the process reasoner. The process reasoner then modifies the process tree by adding the new influences of the view to the current state of the process tree. An new process tree is generated which contains the possibility of an explosion amongst its outcomes in Figure 9.[12]

[12] The diagram shows the process tree after the water in Con-2 has started to be heated and missing states 1 and 3 represent the fluid flow and the possibility of Con-1 emptying respectively.

This time the process reasoner can find no action(s) which will stop this explosion occurring. When the process reasoner receives a message from the user that the pressure in the boiler has begun to rise, the likelihood of the explosion increases. The sealed-container view (of Con-2) relies on the external preconditions `turbine-inlet closed` and `reservoir-inlet closed`. The process reasoner adds the opposite of each of preconditions to the TNMS and checks the resulting outcome. In the case of `reservoir-inlet open` the explosion is negated but there is also a loss of the boiling process and with it the increase in pressure. With the opening of the `turbine-inlet` the explosion is negated and no other harmful effects are noted.

The process reasoner then sends a message to the plan reasoner indicating the type of problem (unrequired event), the process/views which must be removed i.e. the view of Con-2 as a sealed container and the plan effect(s) which must be brought about i.e. `turbine-inlet open`. The plan reasoner then re-uses the required part of the existing plan to create a schema for opening the correct valve. This is then passed back in the usual way via the plan coordinator to Nonlin. A plan is then generated to open the turbine-inlet valve and allow the steam to escape thus causing the pressure to be released. The plan fragment is then integrated into the existing plan without causing further failures.

8 Further work

This section aims to put forward further topics and areas which could be investigated using these techniques. They basically fall into two main areas: those involves new domains and those involving the integration of new techniques.

8.1 Testing in other domains

The present EXCALIBUR system has been tested on problems involving *process control*, *house building* and *coffee making*. These have provided good domains in which the ideas behind EXCALIBUR could be tested. The results which have been produced have shown the possible uses for this type of system in dealing with complex industrial problems. In order to pursue this further it is hoped within the near future to test EXCALIBUR on more

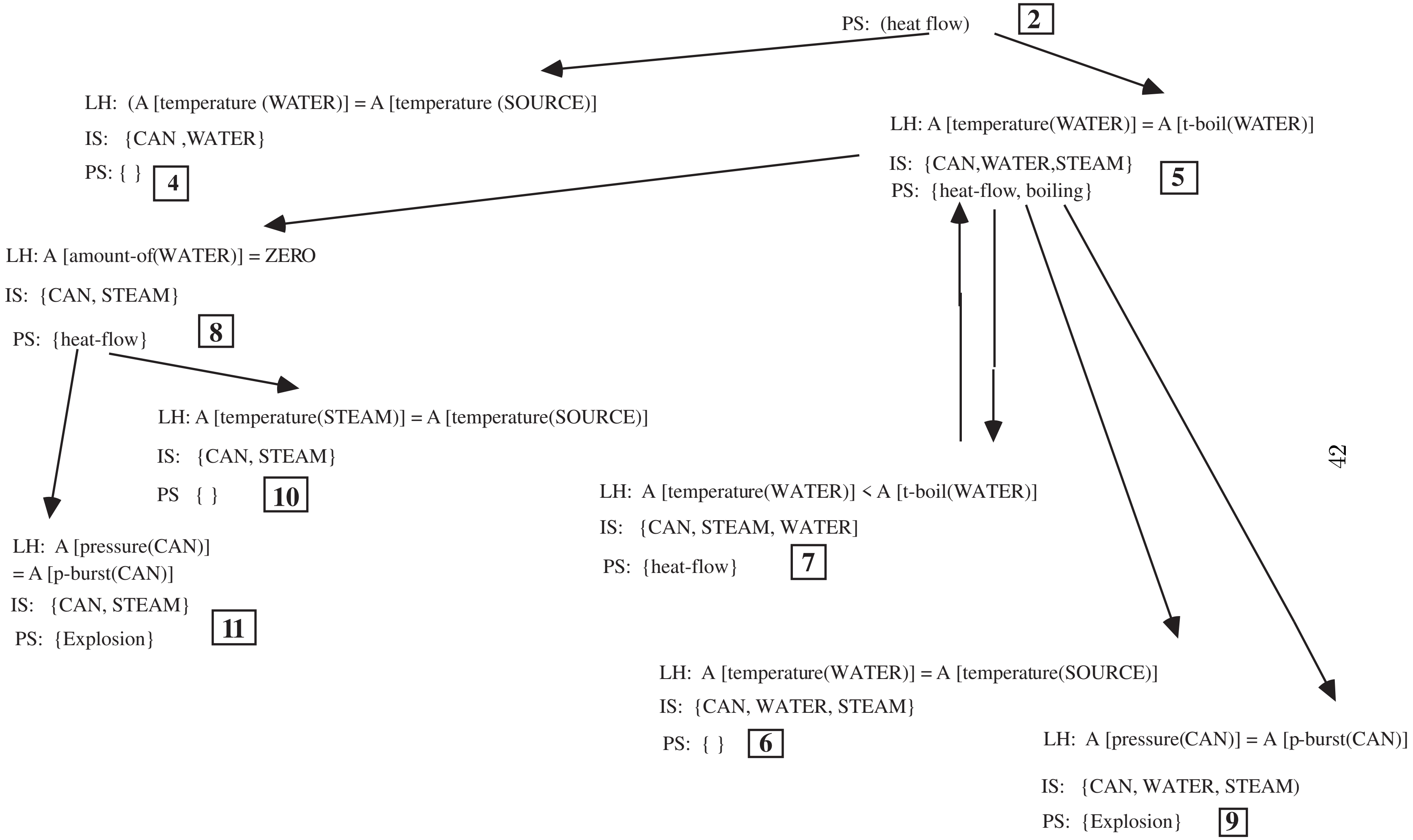

Figure 9: Process tree after an explosion has been predicted

complex problems other than the ones reported here. These involve planning tasks for a steel pipe manufacturing system and in continuous process manufacturing.

The present EXCALIBUR system uses only simple scheduling techniques to decide which actions within a partial order should be executed next. It is hoped to use some of the techniques outlined here to provide guidance in schedule generation and execution as well. The scheduling problem involves allocating resources to the actions of the plan. This is a very difficult task as there are usually many possible allocations which can be made. It is hoped to provide extra constraints to the scheduling process by the use of a qualitative model.

8.2 Plan Generation

Plan generation is a very large and complex *search* problem in which many choices have to be made. Many heuristics have been developed to prune parts of this search space but at the same time maintaining the completeness of the search. Such techniques have included *typed preconditions*, [1] where the user provides information on how to satisfy the preconditions of an action, and *temporal coherence* (TC), [10] which attempts to categorise the *integrity rules* of the domain to check for inconsistent states. In TC in a blocks world domain the rules would consist of "a block cannot be clear and under something at the same time", "a block cannot be on itself", etc. TC has been successful in defining rules at a very *shallow* level but most domains have far more complicated *causal* rules which TC is unable to capture. For example, such techniques allow the difference between precondition types to be identified, i.e. those required to initiate the action and those which must be maintained for the duration of the action. An example of this would be the action "pick up block" which has two preconditions "block on table" and "holding block". As soon as the block is moved the precondition "block on table" fails, but the action is still maintained, whereas if the precondition "holding block" were to fail then so would the action. Other planners such as DEVISER [18] have tried to use different precondition constructs to capture this. However, using the modelling techniques outlined here the same construct can be used for all preconditions and their types can be detected by modelling the changes which they bring about. In the last example the movement of the block would be modelled using a motion process whose preconditions would be capable of detecting the difference between the precondition types. Using such techniques would allow the user to write TF task descriptions without worrying about the precondition types and would allow the dependency recording mechanism and *goal structure* [13] to more accurately reflect the real world changes the plan is designed to bring about.

However by far the greatest advantage in reducing the search space would be achieved through reasoning about resources and time constraints. Time constraints are used in planners such as DEVISER, [18] O-PLAN [1] to prune parts of the search space when no feasible solution can be found in a particular branch. The use of qualitative reasoning and in particular *order of magnitude* reasoning would allow far more complex time constraints to be handled. Order of magnitude reasoning allows comparison of quantities whose exact numerical values may not be known but whose relative size can be specified fairly accurately. For example, heated steel destined for a rolling mill will cool in a matter of minutes so there is no point in heating it a few hours beforehand as the plan would be unusable, not to mention silly. This would allow plans to take into account time constraints which are independent of the schemata and whose format is domain independent.

In the present EXCALIBUR system a plan can be generated to interact with a resource which should be generated at execution time. However, in the present system no check is made that the resource will actually be generated when the plan is created. This means EXCALIBUR will have to immediately patch the plan to achieve the resource which is required. This can easily be achieved by checking during plan generation that the resource will indeed come about. If not, then alternative paths through the search space would be transversed until hopefully a method to achieve the resource was found.

Many of the ideas put forward in this section will be integrated into a new planning system which is currently being designed and implemented here at the AI Applications Institute. The new system aims to close the planning and execution loop within a single control structure and it is intended that the system will make heavy use of qualitative reasoning techniques both in plan generation and execution monitoring. The system is intended to be domain independent with its first target domain being satellite command and control. The main aims of the research are to investigate distributed planning with plans being generated by one agent for execution by another

[13] A Nonlin term - also often referred to as the teleology of the plan

possibly intelligent agent. Within such a system, plan repair becomes more complex as the execution agent needs to follow a plan which it does not necessarily know how to generate, and thus repair. Alternatively a failure may occur in a plan which was generated by the execution agent in response to a loosely defined goal passed to it by the plan generation agent. It is hoped to use model based reasoning techniques to:

- increase the expressive power of a plan's goal structure, i.e. the reason why actions are placed at certain points in the plan.
- provide attachment points within plans to which repair plans can be integrated.
- investigate their use for the modelling of data and power flow within a satellite and to provide a realistic set of scenarios on which to test the systems' capability.

9 Conclusions

The planning system reported here has been implemented successfully in POP11 on a VAX 8650 mainframe machine. The system can handle a wide variety of plans involving processes and certain execution errors which might arise. The system has been applied to domains other than the `make coffee example` and the `process example` described here. EXCALIBUR has been applied to problems involving `block stacking`, `cooking` and `house building`.

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

It is interesting to contrast the approach taken here with that of Vere's system DEVISER [18]. The approaches differ in two respects. First, EXCALIBUR allows the 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 is DEVISER.

It is also interesting to note that Hogge [15], a colleague of Forbus - the inventor of the Qualitative Process Theory ideas , 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 allow the plan to interact with the world or use changes in the world as part of the plan. Also the plans were never executed in the real world and thus the system had no execution monitoring or replan capability. The system he developed used Nonlin-type TF schema definitions and with only minor modifications could be easily integrated as the planning component within the EXCALIBUR system.

EXCALIBUR has been successfully applied to a range of problems from "making a cup of coffee" and "block stacking", to more complicated plans involving "steam generation process control" and "house building". This work brings in an added level of control knowledge, not previously exploited, for controlling the execution of plans. In achieving this it has shown useful, and potentially important, use of qualitative process theory in another area of AI application. This work greatly extends the "knowledge rich" paradigm of AI planning which is all important to flexible use of plan structures.

Based on the work described in this article, it is possible to put forward several advantages for an execution monitoring systems based on this architecture:

1. Changes which the plan should bring about in the real world can be defined within the plan without the need for large precondition lists. These events are then monitored for by the process reasoner that will indeed come about.

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

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

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

5. The plan reasoner can monitor a suspended plan which is waiting for

an external event, but will under most circumstances never allow the plan to wait for an event which will never come about.

6. The system has the ability to react to situations by creating new plan patches to overcome problems caused by failures of plan preconditions and more importantly failures caused by processes not bringing about the required effects.

7. 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. From this analysis a patch plan is generated and integrated into the original plan.

10 Acknowledgements

First and foremost I would like to thank Dr Peter Coxhead, my supervisor. The fact 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. I should also like to thank Roberto Desimone, Ken Currie and Austin Tate for their comments and suggestions on earlier drafts of this article. Finally I would like to acknowledge the financial support of the Science and Enginering Research Council of Great Britain.

References

[1] K.W. Currie and A Tate. O-Plan: the Open Planning Architecture. *Artificial Intelligence*, 51(1), Autumn 1991. Also available as AIAI-TR-67.

[2] R. Davis. Diagnostic Reasoning based on Structure and Behaviour. *Artificial Intelligence*, 24:347–410, 1983.

[3] T.L. Dean and D.V. McDermott. Temporal database management. *Artificial Intelligence*, 33:1–58, 1987.

[4] J. DeKleer and J. Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[5] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[6] B. Drabble. *Intelligent Execution Monitoring and Error Analysis in Planning Involving Processes.* PhD thesis, University of Aston in Birmingham, July 1988.

[7] B. Drabble. Planning and reasoning with processes. In *The Eighth Workshop of the Alvey Planning Special Interest Group*, pages 25–40, Savoy Hotel, Nottingham, November 1988. Institute of Electrical Engineers.

[8] B. Drabble and P. Coxhead. Error detection and recovery in an uncertain environment. In *Proceedings of the IASTED International Symposium on Expert Systems*, pages 201–206, Anaheim, June 1987. Acta Press.

[9] B. Drabble and P. Coxhead. Qualitative reasoning in planning involving processes. In *Qualitative Modelling in Diagnosis and Control*, pages 3/1–3/4, Savoy Place, London, January 1988. Institute of Electrical Engineers.

[10] M. Drummond and K. Currie. Exploiting temporal coherence in nonlinear plan construction. Technical Report AIAI-TR-23, AI Applications Institute, Edinburgh, 1987. A revised version to appear in Computational Intelligence Journal.

[11] K.D. Forbus. Qualitative reasoning about physical systems. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 642–645, Los Altos California, August 1981. William Kaufman Inc.

[12] K.D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[13] K.D. Forbus. The qualitative process engine. Technical Report UIUCDCS-R-86-1288, Department of Computer Science, University of Illinois, 1986.

[14] K.D. Forbus. Introducing actions into qualitative simulation. In N.S. Shridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1273–1278, Loas Altos, California, 1989. Morgan Kaufmann.

[15] J.C. Hogge. Compiling plan operators form domains expressed in qualitative process theory. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 229–233, Los Altos California, 1987. William Kaufman Inc.

[16] B. Kuipers. Qualitative simulation. *Artificial Intelligence*, 29:289–338, 1986.

[17] A. Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–893, Los Altos California, 1977. William Kaufmann Inc.

[18] S. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5:246–267, 1983.

[19] B. Williams. Doing time: Putting qualitative reasoning on firmer ground. In *Proceedings of the National Conference on Artificial Intelligence*, pages 105–112, Loas Altos, California, 1986. Morgan Kaufmann.