An integrated architecture for operating procedure synthesis

This item was submitted to Loughborough University’s Institutional Repository by the/an author.

Additional Information:

- A Doctoral Thesis. Submitted in partial fulfillment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: https://dspace.lboro.ac.uk/2134/7436

Publisher: © James Kinnaird Soutter

Please cite the published version.
This item is held in Loughborough University’s Institutional Repository (https://dspace.lboro.ac.uk/) and was harvested from the British Library’s EThOS service (http://www.ethos.bl.uk/). It is made available under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
An Integrated Architecture for Operating Procedure Synthesis

by

James Soutter

A Doctoral Thesis
Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University

November 1996

© James Kinnaird Soutter 1996
ABSTRACT

The task of creating the operating procedures for a processing plant is time consuming and requires the involvement of key members of the design team. As one of the consequences, the writing of operating procedures is often put off till the final stages of the design process. However, some operability problems will remain hidden in the design until the operating procedure is considered. These problems are expensive to fix because they require undoing some of the design decisions that have already been made.

This thesis reports on research into the automatic creation of operating procedures, a field of research sometimes called Operating Procedure Synthesis (OPS). One motivation for OPS research is to develop a tool that can detect operability problems in the design of a plant and thus allow operability problems to be considered earlier in the design process reducing the cost of resolving these problems.

Previous OPS systems are generally based around single techniques such as mixed integer linear programming. All the techniques that have been examined in the past are strong in some aspects of OPS and weak in some other aspects. There is no single technique that is strong in all areas of OPS. As a result, no previous OPS system is able to generate all the procedures used as examples in the OPS literature.

This thesis presents a new approach to OPS. In this approach, OPS is viewed as a set of distinct but related subtasks. Three subtasks have been identified and examined in this work, namely planning, safety and valve sequencing. Algorithms have been developed to address each of these three subtasks individually. These algorithms have been integrated to form a single OPS system by using a common representation of the operating procedure to be created.

Keywords: operating procedure synthesis, artificial intelligence planning, process plant design.
DEDICATION

This thesis is dedicated to both my parents but particularly to my Mother, Alison Soutter.

When my mother was my age now, she could have been writing her own Ph.D. in Psychology. She gave that up to marry and have children, my brother and me.
ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr Paul W. H. Chung, for valuable guidance and encouragement throughout the course of this work. I am particularly indebted to Paul for giving me the opportunity to work on this project.

I would like to thank my Director of Research, Professor Frank P. Lees, especially for providing valuable comments on a draft of this thesis.

I would also like to thank all my friends for their help and advice. I am particularly indebted to Roger Goodwin and Barbara Dobson, for cake and sanity; Svenja Hanson, for reading parts of this thesis at a moment's notice; and to Farish Lawrence.

This work was funded by British Gas, the Science and Engineering Research Council and the Economic and Social Science Research Council.
# Contents

1 Introduction

1.1 Motivations ................................ 2
  1.1.1 OPS as a formalisation of procedure synthesis .... 2
  1.1.2 Using OPS in new situations ................... 3
1.2 Contributions ................................ 4
1.3 Layout of the thesis ......................... 5

2 Introduction to Planning

2.1 Concepts .................................... 8
  2.1.1 A state ................................... 9
  2.1.2 The representation of an action ............... 11
2.2 Planning .................................... 12
2.3 Planning theory ................................ 17
  2.3.1 Correctness and completeness ................. 18
  2.3.2 The frame problem .......................... 18
  2.3.3 Time complexity ........................... 21
2.4 Least commitment search ........................ 23
  2.4.1 Partial order planning ...................... 25
  2.4.2 Intelligent variables ....................... 26
  2.4.3 Reasoning about partial plans ............... 27
2.5 Hierarchical planning .......................... 28
  2.5.1 Goal expansion ............................ 28
  2.5.2 Prioritised goals ........................... 31
2.5.3 Macro actions .................................. 33
2.6 Conclusion ........................................... 34

3 An Introduction to OPS .......................... 35
3.1 Three aspects of OPS ............................. 35
  3.1.1 Planning ........................................ 36
  3.1.2 Safety .......................................... 37
  3.1.3 Valve sequencing .............................. 39
3.2 Simplifying assumptions ....................... 40

4 Literature Review .................................. 43
  4.1 History of OPS research ......................... 45
  4.2 State graph paradigm ............................ 45
    4.2.1 Procedure synthesis ......................... 46
    4.2.2 Procedure analysis ........................... 47
  4.3 Forward chaining paradigm ..................... 48
    4.3.1 Rivas and Rudd ................................ 50
    4.3.2 Fusillo and Powers ............................ 54
    4.3.3 Tomita, Hwang, O’Shima and McGreavy ...... 60
    4.3.4 Aelion and Powers ............................ 64
    4.3.5 Conclusions on the forward chaining paradigm 67
  4.4 Action synergy ................................... 68
    4.4.1 O’Shima ...................................... 70
    4.4.2 Roach and Strimaitis ........................ 70
    4.4.3 Foulkes, Walton, Andow and Galluzzo .... 71
    4.4.4 Lakshmanan and Stephanopoulos ............. 72
    4.4.5 Conclusions .................................. 73
  4.5 Action ordering systems ....................... 74
    4.5.1 Lakshmanan and Stephanopoulos ............. 74
    4.5.2 Alsop and Macchietto ........................ 75
    4.5.3 Naka and others ............................. 76
C  CEP Example: The Compressor Problem  
   C.1  The task  ......................................... 223  
   C.2  The solution  ..................................... 226  
   C.3  Plan development in CEP  ........................ 226  
   C.4  Implementation  .................................. 228  
   C.5  Conclusion  ........................................ 232  

D  CEP Example: Computer-Aided Planning of Purge Operations  234  
   D.1  The task  .......................................... 236  
   D.2  The solution  ....................................... 237  
   D.3  Implementation  ..................................... 237  
   D.4  Conclusion  ......................................... 243  

E  CEP Example: The Sales Gas Filter  244  
   E.1  The task  .......................................... 244  
   E.2  The solution  ....................................... 245  
   E.3  Implementation  ..................................... 245  
   E.4  Conclusion  ......................................... 253
# List of Figures

1.1 The process of responding to an abnormal condition on a plant ........................................ 4

2.1 A double block and bleed valve arrangement ................................................................. 8
2.2 A forced reboiler .................................................................................................................. 11
2.3 State transition caused by starting pump-201 ............................................................... 11
2.4 A plan to start pump-201 .................................................................................................. 13
2.5 An AI planning loop .......................................................................................................... 14
2.6 The start heater operator ................................................................................................... 15
2.7 The start pump operator .................................................................................................... 15
2.8 All goals are unsolved ....................................................................................................... 15
2.9 The plan at the end of the first cycle .................................................................................. 15
2.10 Conflict in a plan ............................................................................................................. 16
2.11 The final plan ................................................................................................................... 17
2.12 A divider shown with neighbouring valves ..................................................................... 19
2.13 The time complexity of a hypothetical program ............................................................. 21
2.14 At T, p is necessarily true but q is not ............................................................................. 27
2.15 An example of the ramification problem ......................................................................... 29
2.16 A hierarchy of literals ..................................................................................................... 30
2.17 The use of goal expansion .............................................................................................. 30
2.18 A valve sequencing problem ........................................................................................... 32
2.19 The macro action representation of turning off a compressor ...................................... 33
2.20 The goal expansion representation of turning off a compressor .................................... 33

3.1 A forced reboiler ................................................................................................................. 36
7.7 The fox and goose are possibly left alone together .................. 129
7.8 The DetectViolation() Algorithm ........................................ 132
7.9 The fox may be left to eat the goose .................................... 133
7.10 The algorithm to clobber a goal of prevention violation .......... 135
7.11 Some interesting achievers in a goal of prevention violation ...... 136
7.12 Unification ............................................................... 138
7.13 Separation .................................................................... 138
7.14 Constriction ................................................................. 138
7.15 A plan which solves the Satisfiability Problem ..................... 144

8.1 The plant layout .............................................................. 147
8.2 CEP input classified by the representation used within CEP ....... 149
8.3 CEP input classified by re-usability ...................................... 149
8.4 The model of valve v06 — plant modelling information ............ 150
8.5 The model of a valve — unit modelling information ................. 151
8.6 The declaration of chemicals ............................................. 151
8.7 The class aperture in the declaration of concepts .................... 152
8.8 The model of opening a ball-valve ....................................... 152
8.9 A goal of prevention .......................................................... 153
8.10 Goal state ..................................................................... 155
8.11 The flow path created in solving the first problem ................. 156
8.12 Pumps 1, 2 and 4 are now off line ....................................... 156
8.13 The flow path created in solving the second problem ............. 157
8.14 An annotated version of CEP’s procedure for the third blender task 158
8.15 Start and end state for the final problem .............................. 159
8.16 The start state for the final problem ..................................... 160
8.17 An annotated version of CEP’s procedure for the fourth blender task 160

A.1 A forced reboiler ............................................................... 178
A.2 Reserved words in CEP ...................................................... 179
A.3 Frames and instances have three basic formats ....................... 179
A.4 The atoms and classes in the forced reboiler model. ................. 179
E.2 The procedure generate by CEP. 

246
List of Tables

2.1 Example statements in the first order predicate calculus .......... 9
2.2 Example statements in the function literal representation ......... 10
3.1 The effect of opening valve-a in a section of plant ............... 40
6.1 Plant Modelling in QUEEN ............................................. 95
A.1 The effect of opening valve-a in a section of plant ............... 197
B.1 Control variables .................................................. 213
B.2 Debugger commands ............................................. 220
Table of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Property</th>
<th>Informal Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>∧</td>
<td>Conjunction</td>
<td>And</td>
</tr>
<tr>
<td>∨</td>
<td>Disjunction</td>
<td>Or</td>
</tr>
<tr>
<td>¬</td>
<td>Negation</td>
<td>Not</td>
</tr>
<tr>
<td>∀</td>
<td>Universal quantification</td>
<td>For all</td>
</tr>
<tr>
<td>∃</td>
<td>Existential quantification</td>
<td>For at least one</td>
</tr>
<tr>
<td>□</td>
<td>Necessity</td>
<td>In all completions of the partial plan</td>
</tr>
<tr>
<td>◊</td>
<td>Possibility</td>
<td>In at least one completion of the partial plan</td>
</tr>
<tr>
<td>&lt;</td>
<td>Temporal order</td>
<td>Is earlier than (another action)</td>
</tr>
<tr>
<td>≈</td>
<td>Codesignation</td>
<td>Has the same value as (another variable)</td>
</tr>
<tr>
<td>→</td>
<td>Goal expansion</td>
<td>Has been expanded to (a list of goals)</td>
</tr>
<tr>
<td>?</td>
<td>Marker for variables</td>
<td>?x means 'the planner variable called x'.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This thesis is concerned with the automatic, computer-based creation of operating procedures for chemical processing plants. This field of research is sometimes referred to as operating procedure synthesis (OPS).

The long term objective of this project is to develop a practical OPS system. The motivations for this work will be presented in section 1.1. In general, an advanced OPS system will improve plant safety and reduce design costs.

In this thesis we propose a new approach to OPS based on current Artificial Intelligence (AI) planning ideas. An AI planning approach to OPS is attractive. Well understood planning techniques are available to address the areas in which current OPS work has had difficulties.

However, two aspects of OPS are not properly addressed by the AI planning literature. We will call these aspects safety and valve sequencing, but leave their formal definition until chapter 3. A significant contribution of this project has been to develop powerful, planning based algorithms to address the safety and valve sequencing aspects of OPS. These new algorithms allow the creation of OPS systems based on AI planning ideas.

A prototype OPS system has been created based on the ideas developed during this project. This prototype significantly improves upon the state of the art within many aspects of OPS. At the same time, the prototype is extensible to new problem areas by using additional AI planning techniques.
1.1 Motivations

There are two broad application areas for OPS. One area looks at the use of OPS to support the creation of safe operating procedures. The other area looks at OPS to provide procedure synthesis expertise in new situations.

1.1.1 OPS as a formalisation of procedure synthesis

An OPS system provides a standard way to create operating procedures based on an explicit model of a process plant. OPS systems behave in a well defined way and the information provided to an OPS system will never be ignored. As a result, automatically generated procedures will be of a consistent quality. Consistency is important if one wants to go on to look at methods for examining the safety of an operating procedure. If the team checking a procedure must waste time correcting obvious mistakes then they have less time to spend on more subtle problems.

OPS may help to improve the quality of operating procedures. An OPS system can be viewed as a formal method for creating procedures. An error in an automatically generated procedure is logically derived from the knowledge base used to generate that procedure. If a rule is used many times, then an error in the rule will manifest itself in many different ways. As a result, the error will be detected and the knowledge base can be refined. This is similar to the modular programming idea of improving the quality of software by relying on a library of standard functions.

During the lifetime of an operating procedure, ideas about safety and ideas about good plant operation may change. If procedures are created automatically then it is easier to record the operating knowledge used when creating each procedure. When practices change, it will then be possible to find all the operating procedures affected by the change. These procedures can then be rewritten.

An OPS system also provides a method for preserving expertise within an organisation. Plant operators and plant designers have detailed knowledge about how and why tasks are performed on a plant. It should be possible to capture at least part of this knowledge in the form of plant models and planning heuristics for OPS.
1.1.2 Using OPS in new situations

An OPS system captures the expertise needed to create operating procedures and allows this experience to be used in new situations. This expert knowledge could be used in a number of ways.

One idea is to use an OPS system to check a design in the same way that a grammar checker might be used to examine a report. Operating procedures are usually written very late in the design process. As a result, design problems which prevent a plant from being operated may go unnoticed for a long time. These problems will be relatively expensive to correct, as are all problems which are discovered late in the design process. An OPS system could be used to check the operability of a plant while it is on the drawing board.

An OPS system used to check a plant design need not be as complete as an OPS tool used to generate the procedure for a new plant. An analogy can be drawn with the use of a spelling checker. Spelling checkers do not warn about the word ‘their’ when one should have written ‘there’. Similarly, an OPS tool used to check a design would be useful even if it sometimes considered the plant to be operable when it was not. The use of OPS to analyse plant designs may be a valuable way to exploit OPS technology in the short term.

A second idea is to use OPS to examine the effect of plant modification on the operating procedures of a plant. When a plant is modified, the plant engineers must decide which procedures are affected by that change. There is a risk that the modification will not be properly understood, especially if the change is apparently simple. Hence plant modification may introduce hidden errors in the operating procedure manual. An OPS system could help prevent errors by automatically considering the validity of each existing procedure on the modified plant.

This same system could then be used to help recreate the affected procedures. If an OPS system was used to create the original operating procedures for a plant, then the cost of using OPS after modification should be very small. Most of the necessary plant modelling will have been done already.

Finally, OPS could be used to help the plant operator decide on how to act
if something goes wrong with the plant. When a plant deviates from its normal operation, the operator must infer the cause of deviation and then decide on how to respond. This cycle is shown in figure 1.1. Work has been done on fault diagnosis to automatically infer the cause of a discrepancy (see Rose, 1990). However, little work has been done to help the operator to decide how to respond to an incident. OPS could be used to fill this gap.

1.2 Contributions

The contributions of this project are:

- the idea that planning, safety maintenance and valve sequencing are separate activities. This idea is fundamental to the developments made during this project. For example, our prototype OPS system uses AI technology for planning and OPS technology for valve sequencing.

- an approach to OPS that allows AI planning techniques to be used. It is the cornerstone of this thesis that a general and reliable OPS system must be constructed around general and reliable planning technology.

- a novel valve sequencing algorithm. Previous valve sequencing algorithms re-
quire complete information about the state of the plant before generating the instructions needed to create a flow of a chemical. Examples from the OPS literature show that it is very hard for an OPS system both to be able to calculate this state information and to be able to resolve all the safety problems that can arise. We have developed a valve sequencing algorithm that does not require complete state information.

- a novel method for maintaining the safety of an operating procedure during synthesis. This algorithm is based on the explicit representation of goals of prevention (safety considerations) during planning. The algorithm overcomes many of the limitations of earlier OPS work by including general techniques for repairing unsafe operating procedures.

- the development of a prototype OPS system, called the Chemical Engineering Planner (CEP), which implements the new approach to procedure synthesis presented in this thesis.

- an extensive review of OPS research. This review has two purposes. First, to identify the ideas used in earlier systems that would be of benefit in the development of CEP. Second, to understand properly the limitations of earlier work, so that they can be avoided.

- an analysis of planning with safety restrictions. The problem of planning with the idea of unsafe situations has only been commented on briefly in the AI planning literature. We provide time complexity results, a literature review and an analysis of two of the possible methods of working with unsafe states in planning.

### 1.3 Layout of the thesis

This thesis consists of nine chapters and five appendices.

Chapter 2 provides an introduction to AI planning.
Chapter 3 presents an overview of OPS. The reader is introduced to the three main aspects of OPS upon which this thesis will focus. The basic assumptions made by OPS work are also presented.

Chapter 4 reviews the OPS literature.

Chapter 5 describes the architecture of CEP, the prototype OPS system developed in this project.

Chapter 6 examines the idea of planning with unsafe situations and presents the algorithm used in CEP for reasoning about the safety of an operating procedure.

Chapter 7 presents the valve sequencing algorithm used in CEP.

Chapter 8 presents a case study illustrating the use of CEP to solve a sequence of four related problems. The sequence of problems was chosen because it could not easily have been solved by any earlier OPS tool.

Chapter 9 summarises the results of this project and suggests future work.

Appendix A provides a user manual for the current version of the CEP modelling language.

Appendix B provides a user manual for the current version of the CEP OPS system.

Appendix C demonstrates the use of CEP to shutdown a compressor. This problem illustrates CEP’s ability to plan.

Appendix D demonstrates the use of CEP in solving the purge problem from Fusillo and Powers (1988a). This problem illustrates CEP’s ability to work with safety constraints.

Appendix E demonstrates the use of CEP to create a procedure for cleaning a filter. This problem illustrates the general way in which CEP is able to solve valve sequencing tasks.
Chapter 2

Introduction to Planning

In this thesis, the term planning describes the task of finding an ordered sequence of steps or actions which, when carried out from a given initial situation, will achieve a given objective. The sequence of actions formed by a planner is often called a plan or, in the context of OPS, an operating procedure.

OPS is more than planning. For example, an operating procedure must not only achieve its objectives, it must do so in a safe way. Planning is however a major part of OPS. To some extent, an understanding of planning is required in order to understand procedure synthesis.

There are many ways of solving planning problems. This chapter describes a set of planning techniques developed in the AI community. These techniques are ultimately developed from the STanford Research Institute Problem Solver (STRIPS) planner described in Fikes and Nilsson (1971). Other planning techniques will be discussed in the review of the OPS literature in chapter 4.

The role of this chapter is to introduce much of the terminology which will be used later in the thesis. The chapter also introduces the planning technology which forms the basis for the prototype OPS system developed in this project. This chapter is based on the following planning review papers as well as the sources cited in the text: Vere (1992), Steel (1987), Georgeff (1987) and Tate et al. (1990).
2.1 Concepts

There are three ideas which are fundamental to planning: the concepts of a *domain*, a *state* and an *action*.

A *domain* defines the area in which planning will take place. To create a plan, a planning agent must have knowledge of the actions that it is possible to perform and the objects that can be acted upon. Obviously it is infeasible to tell the planner about the whole world and so we use the concept of a *domain* which is the portion of the world which the planner can reason about. The planning domains considered in the OPS literature are usually sections of process plants. The domain knowledge needed by the planner is dependent on the knowledge of the agent that will carry out the plan. For example, consider the operation of the double block and bleed valve arrangement shown in figure 2.1. If the valve set was to be operated by hand then the planner would have to model the state of each of the three valves and understand the correct order in which the valves were to be opened and closed. If a machine were provided to open and close the three valves as a single unit then the planner would represent much less detail. In effect, knowledge about operating the valves must exist somewhere, but not necessarily within the planner.

A *state* or *situation* is a complete description of the planning domain at a single instance in time. For example, consider a very simple plant containing only a single valve. We could model this valve as having two distinct states, one described by the statement ‘open(valve) is true’, and the other state describe by the statement ‘open(valve) is false’.

In many current planning systems, states are assumed not to change unaided. For example, if a plant contains a single valve, then the valve will not spontaneously open or shut by itself. The only way to move between states in a domain is by acting
An action is a method for transforming a domain from one state to another. A class of planners known as deductive planners assume that actions provide the only method for moving the domain to a new state, i.e. the state of the domain is assumed not to change unaided.

### 2.1.1 A state

In order to plan, an agent must have some representation for the state of a planning domain. The STRIPS planner (see Fikes & Nilsson, 1971) and some later planners describe the world in terms of atomic sentences. An atomic sentence has the format `predicate(name, name, ...)` with the optional prefix `¬` (read as 'not'). The meaning of an atomic sentence is defined by the convention chosen by the author of that sentence. For example, an atomic sentence `flow(a, b)` may mean that there is a flow of chemical from a to b or it may mean that there is a flow of chemical from b to a or it may mean something different. Whatever the intended meaning of an atomic sentence, prefixing the symbol `¬` to the sentence creates a new sentence with an opposite meaning. For example, if `flow(a, b)` means there is a flow from a to b then `¬flow(a, b)` means that there is no such flow. Table 2.1 lists some atomic sentences and their intended meanings. In the table a name of the form 'heater-104' should be read as 'the particular heater in the plant which is numbered 104'.

Atomic sentences can be represented as functions that return either true or false. For example, `open(valve-1)` can be represented as 'open(valve-1) is true' and `¬open(valve-1)` can be represented as 'open(valve-1) is false'.

The function based representation can be made more elaborate by allowing each

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>open(valve-1)</td>
<td>valve-1 is open.</td>
</tr>
<tr>
<td>¬active(heater-104)</td>
<td>heater-104 is not active.</td>
</tr>
<tr>
<td>contents(pipe-5, methane)</td>
<td>pipe-5 contains methane.</td>
</tr>
<tr>
<td>pressure(vessel-1, medium)</td>
<td>vessel-1 is at medium pressure.</td>
</tr>
</tbody>
</table>

Table 2.1: Example statements in the first order predicate calculus on the domain.
Predicate Meaning

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>aperture(valve-1) is open</td>
<td>valve-1 is open</td>
</tr>
<tr>
<td>state(heater-104) is off</td>
<td>heater-104 is not active</td>
</tr>
<tr>
<td>contents(pipe-5, methane) is true</td>
<td>pipe-5 contains methane</td>
</tr>
<tr>
<td>pressure(vessel-1) is medium</td>
<td>vessel-1 is at medium pressure.</td>
</tr>
</tbody>
</table>

Table 2.2: Example statements in the function literal representation

*predicate* (function/value pair) to have an arbitrary set of possible values. For example, the predicate pressure(vessel-1)' may have the set of values { high, medium, low }. This more elaborate representation is the *Function Literals* representation described in Vere (1983). Table 2.2 lists the sentences in table 2.1 rewritten as function literals.

The functional literal representation is superior to the atomic sentence representation at describing predicates that take their value from a set of two or more possible values. Consider the predicate pressure(vessel-1) and the set of possible values { high, medium, low }. In the functional literal representation, pressure(vessel-1) is high states that the vessel is at high pressure. This statement implies that the vessel is not at medium pressure because each predicate (function) can have only a single value. In the atomic sentence representation, a similar statement is pressure(vessel-1, high) (is true). Logically this statement does not contradict pressure(vessel-1, medium) (is true) because different predicates (functions) are independent of one another.

Some predicates do not have an obvious value. For example, a pipe may contain a number of different chemicals at the same time and so it is not clear how the value for the contents of a pipe should be chosen. In cases like this, the relation between a predicate and a possible value is usually described by a boolean function literal. For example contents(pipe-5, methane) is true and contents(pipe-5, chlorine) is true specifies that pipe-5 contains both methane and chlorine.

A statement represented by a function literal or an atomic sentence is called a *literal* or alternatively a *condition*. A state of a domain can be described by a collection of literals. Hence table 2.2 is a partial description of a state. A state description is said to be *complete* if it assigns a value to every predicate expressible in the domain.

So why use literals to describe the state of a domain rather than some more natural
representation? First, a logical sentence provides a standard way to express a given fact and so it avoids the ambiguity of a natural language representation. In English there are many ways to represent the same ideas for example "valve-1 is closed" and "the drainage tap on the white reactor vessel has been turned off" may both describe the logical statement aperture(valve-1) is closed. Second, a logical representation provides a framework for generalising ideas using variables. For example, the effect of closing an arbitrary valve, call it ?x, can be represented as aperture(?x) is closed. This ability to generalise ideas will become important when we come to represent actions in planning.

2.1.2 The representation of an action

An action is a method for moving from one state of the domain to another. For example, consider the operation of the forced reboiler shown in figure 2.2. Starting the pump of the reboiler, pump-201, will transform the state of the domain into a situation where pump-201 is on. Figure 2.3 depicts this effect of starting the pump in a particular state of the domain by showing the state of the plant before and after the use of the action.
In the STRIPS planner (Fikes & Nilsson, 1971), actions are described by STRIPS operators. A STRIPS operator is essentially two sets of literals; a set of effects which describe the changes that the action will make to the domain and a set of preconditions which describe the facts which must be true before the action can be used\(^1\). For example, we might define the operation ‘start pump ?p’ as having the precondition set \{state(?p) is off\} and the effect set \{state(?p) is on\}.

The distinction between an operator and an action is not always clear. We will use the term action to refer to a step in a plan. We will use the term operator to refer to a template for defining such a step. For example, an operator might describe how to start any given pump, whereas an action would describe how to start a particular pump at a particular time in a particular procedure.

### 2.2 Planning

This section introduces an AI planning methodology for constructing a plan and demonstrates the methodology using a simple problem.

A plan has three parts: an initial state which completely describes the state of the domain immediately before the plan is to be carried out; a goal state which describes some of the literals that must be true immediately after the plan is carried out; and an ordered sequence of actions to change the domain from the initial state to the goal state. The state transition shown in figure 2.3 can be seen as a very simple plan consisting of only one action.

The role of a planner is to find an appropriate sequence of actions when given a set of goals and an initial state and a set of operators which can be applied in the domain.

\(^1\)In STRIPS, the set of effects of an action is divided into two parts, an add-list and a delete-list. STRIPS represents facts as atomic sentences and only stores true facts about the world. Using this representation, the effect of an action is to delete some old facts and to assert some new ones, hence the two lists.

More recent planners describes the domain using function literals. Using this representation, each action has the effect of changing the values of some of the functions describing a domain. For example, turning on a heater will change the function representing the state of the heater from the value off to the value on. There is no need to subdivide the effects of an action into an add-list and a delete-list.
Figure 2.4: A plan to start pump-201

Part of the brilliance of STRIPS is to view a plan in terms of actions and their effects rather than viewing a plan in terms of states bridged by actions. For example, the state transition from figure 2.3 can be represented in terms of actions and their effects as shown in figure 2.4. In figure 2.4, an action is represented by a lollipop with the name of that action. Two special names, ‘<’ and ‘>’, are used to represent the start and end of planning respectively. The preconditions of an action are written on the left of its lollipop and the effects are written on the right. The start of planning has only effects, the initial state. The end of planning has only preconditions, the goal state.

Figure 2.4 shows how the preconditions of the goal state of the plan are achieved by the action in the plan and ultimately by the start state. The task given to an AI planner is to find some way to satisfy each precondition of the goal state. This is a backward chaining approach to planning.

There are two methods by which an unsolved precondition, or goal, can be satisfied. Either the goal can be matched with the effect of an action already in the plan, as with ‘state(heater-201) is off’ in figure 2.4, or a new action can be added explicitly to solve the goal, as with ‘state(pump-201) is on’ in the figure.

The basic planning cycle in any AI planner is a process of choosing an outstanding precondition, or goal, and finding a way to solve that chosen goal. One version of this cycle is shown in figure 2.5. If there is no way to solve a particular goal or the solution to a goal cannot be added to the plan then the planner will backtrack.

There are many ways to implement the planning cycle in figure 2.5. We will
demonstrate one possible implementation using the forced reboiler shown in figure 2.2. The problem involves starting up both the heater and the pump. The initial state of the problem is the same as in figure 2.4, that is \{state(pump-201) is off, pump_of(heater-201) is pump-201, state(heater-201) is off\}. The goal state is \{state(pump-201) is on, state(heater-201) is on\}. The problem requires two operators, ‘start heater ?h’ and ‘start pump ?p’ which are shown in figure 2.6 and 2.7 respectively.

At the start of planning, the procedure contains just the start action and end actions (see figure 2.8). The goal agenda contains two goals, state(pump-201) is on and state(heater-201) is on.

In the first planning cycle, the goal ‘state(pump-201) is on’ is selected from the agenda. The only operator which solves this goal is ‘start pump ?p’. The action ‘start pump pump-201’ from this operator is added to the plan. The planner protects the goal between the new action and the end of the plan by adding a causal link. A causal link represents a decision that a certain condition must be true in a certain region of the plan. The new plan is shown in figure 2.9.

The action ‘start pump pump-201’ has the precondition state(pump-201) is off. This precondition is added to the goal agenda. In the next planning cycle the precondition is chosen from the agenda and matched to an effect of the start state. A causal link is added to protect the condition between the start action and the action.

Figure 2.5: An AI planning loop
Figure 2.6: The start heater operator

Figure 2.7: The start pump operator

Figure 2.8: All goals are unsolved

Figure 2.9: The plan at the end of the first cycle
‘start pump pump-201’.

The next goal on the agenda is ‘state(heater-201) is on’, a precondition of the end of the plan. An action from the operator ‘start heater ?h’ is used to solve this goal. Initially, the action is unordered relative to the action ‘start pump pump-201’. The resulting plan so far is shown in figure 2.10.

There is a problem with the plan. The precondition state(?p) is on of the action ‘start heater heater-201’ conflicts with the causal link protecting state(pump-201) is off between the start of the plan and ‘start pump pump-201’. There are two possible ways to resolve this conflict, either ‘start pump pump-201’ can be ordered to occur before ‘start heater heater-201’ or the variable ?p can be prevented from binding to pump-201. If the latter option is chosen, then the planner will eventually have to backtrack because the goal pump_of(heater-201) is ?p will not be solvable. Ultimately then, the planner will resolve the conflict by ordering the action ‘start pump pump-201’ before the ‘start heater heater-201’.

The preconditions of the action ‘start heater heater-201’ are added to the agenda and solved in later planning cycles. The finished plan is shown in figure 2.11.

This example has demonstrated three important aspects of AI planning. First,
a goal can be solved by adding a new action or by relating the goal to an existing action. Second, a planner may add new goals to the agenda when solving a goal. This is sometimes called subgoaling in the OPS literature. Third, casual links are used by the planner to protect the solution of each goal. If there is ever a conflict between a causal link and an action in the plan then the planner actively resolves this conflict.

2.3 Planning theory

The last section introduced planning and demonstrated one implementation of the planning methodology.

Planning is only of interest in this thesis as a method for fulfilling the planning requirements of an OPS system. In order to use planning in OPS, the algorithms must be augmented to address OPS issues like the need to create ‘safe’ operating procedures and the need to solve valve operation tasks. This section examines the theoretical constraints on modifying the planning algorithm.
2.3.1 Correctness and completeness

Ideally, a planning algorithm should be algorithmically correct and complete. There should not be a problem that the planner solves incorrectly, nor should there be a problem that the planner can represent but not solve. Formally, correctness and completeness can be defined as given below.

Correctness If the modelling language of a planner is able to express a problem \( t \) and if the planner produces a set of procedures when given \( t \) as input then each procedure in the set should actually be a solution to the problem \( t \) according to the assumptions of the OPS system.

Completeness If the modelling language of an OPS system is able to express a problem \( t \) and if there is a procedure which is a solution to \( t \) then the planner should never produce the empty set of solutions when \( t \) is given as the problem to be solved.

The need for correctness and completeness is a significant constraint on the way that the planning algorithm can be modified. It is usually not acceptable to create a planner which solves easy problems very quickly but fails on the harder, more interesting problems.

2.3.2 The frame problem

The frame problem is the problem of representing the effects of an action on a given domain state. The frame problem also includes the reverse problem, the task of representing the literals that will not be affected by an action.

The simplest solution to the frame problem, in other words the simplest action representation, is to create a table with every possible action on one axis and every possible predicate on the other axis. The cells in the table then either provide the value of the predicate asserted by the operator or mark that predicate as unchanged. Essentially this is the idea of frame axioms presented in the planning literature. This solution is thought to be infeasible because the size of the table will be too large in most practical planning domains.
The solution to the frame problem used in STRIPS planning is the STRIPS operator representation. Accompanying this representation is the *STRIPS assumption*. The assumption is that predicates which are not mentioned in the effects list of an operator are not affected by the use of that operator. In other words, STRIPS assumes that the effects list of each operator is complete.

Every possible set of actions, that is every possible set of translations from one state of a domain to another, can be represented by a set of STRIPS operators. As a result, operators can be viewed as a solution to the frame problem. However, operators are an inefficient means of describing some domains.

Consider the task of operating the valves on the plant shown in figure 2.12. If valve x is closed then opening valve y will have very little effect on the plant. If valve x is already open then opening valve y will create a flow of hydrogen to outlet-1. To satisfy the STRIPS assumption, a model of the plant must have at least two operators to open valve y, one for when valve x is open and the other for when valve x is closed. To describe all the possible valve operations in the simple plant, many more operators will be required. The more similar operators available to the planner, the more likely the planner will choose the wrong operator and so have to backtrack and so waste effort when creating a plan.

The operator representation is clumsy at describing some domains because some predicates in those domain are difficult to reason about. For example, it is difficult to model the opening of a valve because it is difficult to reason about flow. To simplify the representation of the domains, dedicated algorithms can be written to reason about the ‘difficult’ literals in the domain. It is then not necessary to describe the effects that each operator has on these literals. In effect, this is a process of breaking the STRIPS assumption and then creating some algorithm to repair the hole that
this creates in the planner’s reasoning ability.

We identify three classes of predicates that are difficult to reason about using the STRIPS operator representation:

Safety: In one view of the safety of an operating procedure, a procedure is said to be safe if it does not move the plan through some unsafe state of the world. The idea of safety is hard to reason about. Assume that the mixing of hydrogen and oxygen in a single pipe is one marker that a state is unsafe. If an action achieves `contains(?p, oxygen) is true’ then it will also achieve ¬safe if ?p already contains hydrogen. If the pipe does not already contain hydrogen then the action will not assert ¬safe. Chapter 7 provides a discussion on planning with the idea of unsafe situations.

Valve operations: Valve operations are generally difficult to model. It is difficult to decide when opening a valve will create a flow of chemical. Any flow will move chemical around the plant and so there are problems reasoning about the contents of the pipes in a plant. A flow might also cause a chemical to be removed from a pipe, or purged, causing additional modelling problems. Chapter 6 suggests one possible solution to the valve operation problem.

Hierarchical Planning: A planner is often required to work at different levels of abstraction. For example, the planner may have to reason about shutting down a plant as well as reasoning about operating the equipment that will eventually lead to that shutdown. High level concepts are difficult to model because they are only asserted after a set of lower level conditions have been achieved. For example, shutdown occurs only after the machinery in the plant has been switched off and isolated etc. In a plan, the action which achieves shutdown is the action which asserts the last of the lower level conditions which together define the plant as being shutdown. The action to negate the shutdown predicate is the first action to negate one of the lower level conditions. This is difficult to represent using the STRIPS operator representation. A discussion of Hierarchical Planning is given in section 2.5.
2.3.3 Time complexity

Time complexity theory is a way of analysing the quality of an algorithm based on how that algorithm scales up to larger problems. Time complexity is not part of planning but has been used in the literature to evaluate OPS systems and planning algorithms. This section will provide an informal introduction to time complexity theory and will then discuss the time complexity of planning. The introduction is based on work in Garey and Johnson (1979) and on work in Azmoodeh (1990).

For any algorithm, a scatter graph can be drawn relating processing time for a problem against problem size for some implementation of that algorithm (see figure 2.13). Each problem size may contain more than one point because there may be more than one problem of that size. Informally, the worst case time complexity for an algorithm is the function of problem size that describes the upper bound on the points in the scatter graph for some implementation of that algorithm. It is common to refer to the worst case time complexity of an algorithm simple as the time complexity of that algorithm.

The shape of the time complexity curve is of particular interest. If the curve can be bounded above by a polynomial function then the algorithm is said to have polynomial time complexity. Polynomial time complexity is considered to be good. Doubling the speed of the computer used will increase by some percentage (dependent on the polynomial) the size of problem that can be solved in a fixed period of time. At the moment, the fastest computer on the market is roughly twice as fast as the
If an algorithm does not have polynomial time complexity but the time complexity can be bounded above by an exponential function then the algorithm is said to have exponential time complexity. With exponential time complexity, doubling the speed of computer used will only allow a fixed increase in the size of problems that can be solved in a given period of time. With exponential time complexity it becomes important to think about the biggest problem that can be solved using current technology. In the next few years the biggest solvable problem will not increase in size that much. As a result, exponential time algorithms are considered to be bad.

Up to this point, the time complexity of an algorithm has been considered. The time complexity of a problem, e.g. synthesising an arbitrary operating procedure, can also be studied. Some problems are known to be solvable by polynomial time algorithms. These problems are said to be *tractable*. Some problems are believed to be solvable only in exponential time. These problems are said to be *intractable*. Some problems are known to be unsolvable and these problems are said to be *undecidable*.

It is hard to prove that a problem is intractable. However, a problem can often be shown to be at least as time consuming as some other seemingly intractable problem. The class of *NP-complete* problems have two important properties: (1) if any *NP-complete* problem can be shown to be tractable then all *NP-complete* problems are tractable and (2) there is currently no known polynomial time algorithm to solve any problem in the class. If a problem can be shown to be at least as difficult as a member of the class of *NP-complete* problems then the problem is probably intractable and is said to be *NP-hard*.

In general, planning with STRIPS operators is an NP-hard task (see Bylander, 1994). It is tractable to plan with some restricted classes of STRIPS operators but these restricted languages are simply not powerful enough to describe many interesting OPS problems.

It would be wrong to think that general planning could be made tractable by choosing some new action representation scheme. From the experience in the OPS
literature, any new representation scheme will have one of two problems. The new scheme may be able to represent some intractable subclass of the STRIPS operators, in which case planning with the new scheme is NP-hard. Alternatively, the new scheme will describe only some tractable subclass of STRIPS operators, in which case the new scheme will be very limited in the planning problems that it can represent.

Although planning is NP-hard, practical planning systems have been developed. O-Plan and SIPE are good examples (see Tate et al., 1992; Wilkins & Desimone, 1994, respectively). By using techniques both to improve search efficiency and to reduce the search space, these planners are able to solve practical problems without running into time complexity problems.

Planners that do not use powerful search heuristics do have time complexity problems. Hence, when selecting a planner for an OPS system, it is necessary to select an architecture which will support intelligent search techniques.

### 2.4 Least commitment search

Least commitment search is an important heuristic for reducing computation time in planning. The least commitment heuristic is admissible, i.e. it does not remove valid plans from the planning space. In contrast, the planning algorithms used in the OPS literature often rely on action selection heuristics and these heuristics tend to be inadmissible.

In this section we will describe the least commitment search heuristic and its use. To understand the heuristic it is important to understand how time can be wasted during planning.

Planning requires making choices about how to solve each goal and where in the plan to order each action. Sometimes the correct decision for the planner to make is obvious. For example, if a new action will only fit into one place in a plan then the action should be ordered at that place. Often, however, the planner has to make a decision by arbitrarily choosing between some alternatives. The reasoning process is something like, ‘assuming I make this choice, what other choices do I have to make to form a valid plan’. If the planner makes the wrong decision it will find it impossible
to create a valid plan and so will have to backtrack and select another alternative.

The least commitment approach is to make a selection by representing decisions explicitly in the planning structure and then refining the alternatives for that decision. One way this can be done is by representing decisions as a variable. For example, a plan to remove small amounts of condensation from a pipe might contain an action 'purge with chemical ?c' where ?c is constrained to be either hot nitrogen or hot oxygen. In effect ?c represents a delayed decision on which purgative to use. If the plan is modified to achieve some new goal requiring that the pipe does not contain nitrogen, then ?c will be bound to oxygen and the decision will be made.

Least commitment planning can be thought of as a method of working with a large number of plans at the same time. That is, if a plan contains a variable with n alternative values then the plan actually represents n plans, one for each binding of the variable. We will call a plan that represents a set of many possible plans a partial plan. Conversely, a plan that only represents one possible plan will be called a complete plan. The idea of partial plans and complete plans should not be confused with the idea of finished plans and unfinished plans. A finished plan is a possibly partial plan in which the preconditions of every action are satisfied. Hence a plan may be complete, i.e. contain no unmade decisions, but at the same time it may be unfinished, i.e. it may contain unsolved preconditions.

A planner that is able to work with partial plans can do exponentially more work with each step than a planner that only works with complete plans. For example, if a plan represents m decisions that could each be made in n ways then the plan has \(n^m\) completions. One unit of work on this partial plan is equivalent to performing \(n^m\) units of work, one for each of the completions of this plan.

Least commitment planning is a trade off. On one hand, each unit of work done by the planner on a partial plan translates into an exponential number of units of work done on the exponentially large set of completions of a of plan. On the other hand, at each step the planner must reason about an exponentially large set of completions and this makes the planner harder to write and adapt.

In the development of planning, the assumption has been made that least com-
mitment search is almost always a good thing. However, recently this view has been challenged by planners like Prodigy (Blythe & Veloso, 1992). It appears that sometimes the use of least commitment search can complicate planning enough to prevent the use of more valuable search heuristics.

In conclusion, least commitment search is a powerful technique for reducing planning time by reducing backtracking. There is a limit to the kinds of decision that can be made in a least commitment way because the cost of working with a partial plan must not approach the cost of working with each completion of that plan individually. In general least commitment search has been found to be a good idea in planning. Most current planners use the following three least commitment search strategies.

### 2.4.1 Partial order planning

Partial ordering was first described in Sacerdoti (1985). In partial order planning, decisions about the order of the actions in a plan are handled using a least commitment approach. Rather than arranging the actions in the plan as a fully ordered sequence, the planner records the constraints on the order of each action. That is, each action's order is defined by a list of necessarily earlier actions and a list of necessarily later actions.

In the process of solving goals in the plan and in resolving conflict in the plan, the order of the plan will be refined. Plan order is refined by adding *temporal constraints* to the plan. A temporal constraint has the format $x \prec y$ meaning 'action $x$ comes before action $y$'.

An example of partial order planning is given in section 2.2. The example looked at the starting of a heater and a pump. At first there is no obvious reason for the pump to be turned on before the heater or after the heater and so the decision is left unmade. Later a conflict is found between a precondition of the start pump action and a causal link in the plan and because of this conflict the two actions are ordered.

With more complex problems, many ordering decisions are left open for long periods of time and it is clear that the use of partial ordering reduces the amount of backtracking required during planning.
2.4.2 Intelligent variables

Variables allow a least commitment approach to the selection of a specific action from a set of possible actions. For example, in figure 2.10 the action 'start heater-201' has a variable ?p to represent which pump is associated with heater-201. If we assume that there are n different pumps in the domain then there are n completions of this action, each with a different selection of ?p.

One possible handling of variables in planning is to have each variable assigned to a set of values. During planning, the set of values is constrained by relating that variable to other variables and values in the plan. If the variable is ever found to have only one possible value then that variable is bound to that value.

There are two relations which are used to constrain the possible values of a variable. The codesignation relation, written ‘symbol ≈ symbol’, relates two plan symbols with the same value. For example, ‘?x ≈ pump-1’ should be read as 'the value of ?x is pump-1'. The noncodesignation relation, written ‘symbol ≠ symbol’, relates two plan symbols that cannot have the same value. For example, ‘?x ≠ ?y should be read as 'the value of ?x is not the value of ?y'.

Most constraints have the effect of constraining the set of values for a variable. The exceptions are constraints of the format ?x ≠ ?y. These constraints are delayed until the binding of one of the two variables.

In some rare cases, these delayed constraints can over constrain the plan. As an example of an over constrained plan, consider the case where ?x, ?y and ?z are all members of the set {true, false} and where ?x ≠ ?y, ?y ≠ ?z and ?z ≠ ?x. The plan is over constrained because the three variables must share two possible values. If the planner cannot detect this, effort will be wasted trying to finish the plan. In some cases, the task of finding an acceptable assignment of values for the set of variables in the plan is NP-hard (see Chapman, 1987, section 3.2.3). In practice, the very occasional time cost of working with over-constrained plans is more than made up for by the regular time saving achieved by allowing variables to be non-codesignated.

The handling of variables described here is similar to the handling of variables in
the TWEAK³ planner described in Chapman (1987). The major difference is that TWEAK variables are not associated with sets of possible values. By not associating variables with possible values, TWEAK hopes to avoid problems with delayed constraints. It is not clear that this approach is at all successful. If a variable ?s is defined to be the current state of a switch and the planner imposes constraints on ?s such that '？s ≠ true and '？s ≠ false' then ?s is over-constrained whether the planner understands this or not.

2.4.3 Reasoning about partial plans

Reasoning about a partial plan is significantly more difficult than reasoning about a complete plan. In a partial plan, a literal is no longer simply true at a point in the plan. Instead, the literal is true at that point in a subset of the completions of that plan. If the subset is non-empty then the literal is said to be possibly true at the given point. If the subset is non-empty and includes all the completions of the plan then the literal is said to be necessarily true at the point.

As an example of possible and necessary truth, consider figure 2.14. The figure shows a plan which is constrained so that two actions, labelled 1 and 2, are ordered strictly before an action \( T \). In the figure, the literal \( p \) is necessarily true at \( T \) because \( p \) is made true by 2 and not possibly negated by 1. The literal \( q \) is only possibly true at \( T \) as is the literal \( \neg q \). In one completion of the plan, action 2 will come before action 1 and \( q \) will be true at \( T \). In the other completion, 1 will come before 2 and \( \neg q \) will be true at \( T \).

³It is not clear from Chapman (1987) exactly what TWEAK is an abbreviation for.
A modal truth criterion is an expression for evaluating the necessary and possible truth of a literal at a given point in a partial plan. Every planner that works with partial plans must embody some modal truth criterion. The first attempt to formalise the modal truth criterion is given in Chapman (1987). This formalisation is corrected and partly reinterpreted in work by Fox and Long (1994) and by Kambhampati and Nau (1996).

2.5 Hierarchical planning

A planning domain can often be described and reasoned about at many different levels of abstraction. For example, the statement “the suction, discharge and bleed valves of the compressor are all closed” is equivalent to the high level statement “the compressor is isolated”.

In the description of planning thus far in this Chapter, there has been no provision in the planner for translating between different levels of abstraction. For example, consider the problem of achieving the goal “isolate the compressor”. It is not clear how this goal can be reformulated or achieved so that the suction, discharge and bleed valves in the compressor all end up closed at the right point in time.

The process of planning at different levels of abstraction when solving a single problem is called Hierarchical Planning. In the past, people have looked to hierarchical planning to solve three problems: the translation of high level goals to lower level goals, the solution of a goal by a sub-plan and the reduction of planning time. This section will examine each of these three problems individually and propose different but compatible solutions to each.

2.5.1 Goal expansion

People often describe procedure synthesis tasks at a high level using ideas like shutdown and startup and isolate. However, the procedures generated to perform these tasks are usually written at a lower level, involving the precise operation of specific plant items. To further complicate matters, high level concepts are often ambigu-
ous. For example, the term shutdown could imply emergency shutdown or normal shutdown. In the case of ambiguous goals, the planner must choose the appropriate interpretation of the goal to meet all the other requirements on the plan or procedure.

This subsection looks at the problem of translating a high level goal into a set of low level goals. This translation is difficult to achieve using operators alone because of the ramification problem, an aspect of the frame problem. The ramification problem is the observation that if the planner has two ways to represent the same idea, e.g., a high level representation and a low level representation, then it is difficult to keep these two representations of the idea up to date.

As an example of the ramification problem, consider a domain which contains the high level literal isolated(compressor) is true and this literal is equivalent to the set of lower level literals aperture(suction) is closed, aperture(bleed) is closed and aperture(discharge) is closed. A specific plan in this domain is shown in figure 2.15. In the plan, the precondition of the action depressure conflicts with the protection of the condition isolate(compressor) is true because the bleed valve must be closed when the compressor is isolated. The planner does not understand the relationship between the isolation of the compressor and the aperture of the bleed valve and so will not find this conflict.

The ramification problem can be avoided if the literals in the domain are arranged in a hierarchy. For example, the literals in the simple domain described above can be arranged in the hierarchy shown in figure 2.16.
isolate(compressor) is true

aperture(suction) is open    aperture(bleed) is open    aperture(discharge) is open

Figure 2.16: A hierarchy of literals

Figure 2.17: The use of goal expansion

The literals at the bottom layer of the literal hierarchy are called primitive literals. If the operators in a domain only contain primitive literals then there is no problem with ramification because every literal is independent of every other literal. If operators contain high level literals then rewrite rules can be used to translate these high level literals into primitive literals. With the use of these rewrite rules the planner only needs to reason in terms of primitive literals. The process of translating a goal that is a high level literal into a set of lower level goals is called goal expansion.

Consider again the plan in figure 2.15, goal expansion can be used to allow the detection of the conflict in this plan. A rewrite rule is used to replace the goal isolate(compressor) is true by three lower level goals aperture(suction) is closed, aperture(bleed) is closed and aperture(discharge) is closed. Figure 2.17 shows the plan after the application of this rule. In figure 2.17, the action isolate is replaced by the equivalent action isolate2.

High level literals may be ambiguous in that there may be a number of low level interpretations of a high level condition. For example, a high level condition such as...
'the car is stopped' may be achieved by an action achieving the lower level condition
‘the hand-brake is applied’ or the low level condition ‘the foot-brake is depressed’. If
the high level literal ‘stop the car’ is a precondition of an action then it is clear that
this precondition can be satisfied by achieving either of the two lower level conditions.
However, if a high level literal is the effect of an action then it is not clear which lower
level literal is equivalent to this effect. For example, an action that causes a car to
stop will not necessarily stop the car by using the hand-brake. To avoid ambiguity,
the effects of each action must be written in terms of primitive literals.

The use of goal expansion requires two assumptions about a planning domain.
First is the assumption that the predicates in the domain form a hierarchy. For
example, there cannot be a rule to rewrite closed(bleed) is true into open(bleed) is
false if there is also a rule to rewrite open(bleed) is false as closed(bleed) is true.
Second is the assumption that no operator in the domain has a non-primitive literal
as an effect.

2.5.2 Prioritised goals

The heart of a planner is a loop in which a goal is selected and then the goal is solved
(see figure 2.5). In this section we will look at the task of goal selection, an idea that
has been so far neglected in this chapter.

Most recent planners do not rely on goal selection for either correctness or com-
pleteness. These planners tend to be based on the goal achievement strategy described
in Waldinger (1977) which states “In order to achieve a goal of the form P and Q, we
construct a plan F that achieves P, and then modify F so that it achieves Q while
still achieving P”. We will consider only these planners throughout the remainder of
this discussion on goal selection.

Although the formal properties of most planners are not affected by goal selec-
tion, the choice of goal selection strategy can improve the speed at which plans are
generated.

One goal selection heuristic, developed by Sacerdoti (1974), reasons that some
goals are more easily achieved than others. In this strategy, each literal in the domain
is assigned a weight depending on how easy that predicate is to achieve. In effect, this weighting process arranges the literals in the domain into a ranked hierarchy with the hardest to achieve predicates taking the highest weightings. The planner is then instructed to choose the goal with the highest weight.

Using this strategy, the planner is directed to create a high level plan and then to go back and fill in progressive levels of detail into this plan. The high level plan that is first created must be self-consistent and so some of the potential errors in the high level plan will be resolved before the next level of detail is considered. Each successive layer will be checked in the same way. In this way, the use of prioritised goals helps in the earlier detection of mistakes in the plan. This in turn helps to reduce backtracking and so improve planning speed.

For example, consider the problem of creating a flow of natural gas and a flow of air in the plant shown in figure 2.18. In solving this problem we choose to rank the predicates to create a flow higher than the predicates to open and close individual valves. The first stage in solving this problem is to choose a flow for natural gas. The route \( (a, d, g, e, c) \) is the first to be discovered (Chapter 6 details how such a route may be found in planning). The next high level goal on the agenda involves creating a flow of air. No flow route is found because valves \( g \) and \( d \) are currently reserved for the flow of natural gas. The planner backtracks and reroutes the flow of natural gas. The new flow path is \( (a, b, c) \). With this new flow path for natural gas, the flow of air can be routed along \( (f, g, h) \). This completes the high level planning process. Only at this point does the planner consider the details of opening and closing the valves necessary to create these two flows.

In many planners, goals must be weighted manually. However, recent work by Knoblock (1994) provides a method for weighting goals automatically.
2.5.3 Macro actions

Rather than solving a goal by proposing a single action, it is sometimes necessary to solve the goal by proposing a collection of actions. For example, to turn off a particular type of compressor it is necessary to switch the compressor off and then wait for the motor to spin down. Until the motor stops spinning, the compressor should be modelled as being active.

To represent the process of solving goals by a set of actions, we use the idea of a *macro action*. A macro action is simply an action that can be rewritten into a sub plan. The macro action representation of turning off a compressor is shown in figure 2.19.

A macro action is not strictly required to solve the problem of turning off a compressor. A clumsy solution can be formed by using goal expansion as shown in figure 2.20. Macro actions only become necessary when specific conditions must be protected across a sub plan. For example, consider a sub plan to repair a compressor by turning the compressor off, performing some maintenance actions and then turning the compressor back on. While the maintenance tasks are performed, the compressor...
must remain off. Without the use of macro actions, there is no way to force the planner to keep the compressor off. Goals can be used to ensure that the compressor is off before specific actions in the maintenance of the compressor but this does not prevent the planner from turning the compressor on then off between the actions with these goals.

The macro action representation is very powerful. For example, the idea of goal expansion can be implemented by using macro actions. Prioritised goals can also be represented as macro actions if the planner is made to solve action expansion goals after it solves normal goals. However, planning with macro actions is not well understood at a theoretical level.

In the planning literature, the idea of macro actions first appeared in Fikes et al. (1972). In Sacerdoti (1985) and many later planning systems, macro actions and the action expansion operators are in part used to replace the goal achievement procedures of the more traditional AI planning systems.

2.6 Conclusion

This chapter has provided a detailed introduction to some of the fundamental ideas of planning. The basic AI planning framework was introduced in the first two sections. In the third section, the frame problem and the time complexity problems were introduced as issues that must be addressed when moving away from the AI planning methodology. These issues will become important in the later sections which describe the adaptation of AI planning to OPS. The fourth section examined the least commitment planning heuristic, a heuristic allowing the efficient evaluation or larger planning spaces. The use of least commitment search is one of the major differences between AI planning the planning techniques traditionally used in OPS. In the fifth and final section, the idea of hierarchical planning was introduced. This idea becomes very important when we come to think about OPS problems involving valve operations.
Chapter 3

An Introduction to OPS

This chapter provides an introduction to OPS research. It gives an overall view of the issues that this thesis seeks to address and the contexts in which these issues will be resolved later in the thesis.

The chapter is divided into two sections. The first section examines three aspects of OPS. The second section looks at the simplifying assumptions which are made in the creation of most existing OPS systems.

3.1 Three aspects of OPS

As a field of research, OPS is very broad. To create a general OPS system, issues about the correct presentation of a procedure and about the choice of actions in a procedure, about user interface design, etc., must all be considered.

This thesis is concerned with the part of OPS that addresses the question "What steps should I perform to achieve a particular objective in a safe way?". Specifically, the thesis focuses on three key aspects of the OPS task; planning, valve sequencing and safety. The planning task is the task of choosing the actions (steps) needed to achieve an objective. Valve sequencing is the task of creating a flow of chemical from one part of a process plant to another. The safety task involves preventing an OPS system from creating procedures that might be considered unsafe.
Figure 3.1: A forced reboiler

Table 3.1: The initial and goal state for the forced reboiler problem

<table>
<thead>
<tr>
<th>Initial State</th>
<th>Goal State</th>
</tr>
</thead>
<tbody>
<tr>
<td>state of heater-201 is off</td>
<td>state of heater-201 is on</td>
</tr>
<tr>
<td>state of pump-201 is off</td>
<td>state of pump-201 is on</td>
</tr>
</tbody>
</table>

Figure 3.2: The initial and goal state for the forced reboiler problem

3.1.1 Planning

Planning involves choosing the sequence of actions needed to achieve a set of objectives. The sequence of actions formed by a planner is often called a plan or, in the context of OPS, an operating procedure.

A planning problem is usually defined by a plant model and by two situations called the initial state and the goal state. The plant model describes the equipment in a plant as well as the actions that can be performed with this equipment and the safety constraints on these actions. The initial state describes the state of the plant immediately before the operating procedure is to be carried out. The goal state describes the facts which must be true after the procedure has been completed.

As a simple example of planning, consider a procedure for the forced reboiler shown in figure 3.1. The objective of the procedure is to start up a heater and a pump (see figure 3.2). To solve this problem, the model for the forced reboiler should define concepts like heater, pump, on and off as well as describing the actions that model starting and stopping a unit.

Using this model, a planner simply has to combine two actions in order to solve the problem: start pump-201 and start heater-201. These two actions can take any
order relative to one another and so “start pump-201 then start heater-201” and
“start heater-201 then start pump-201” are both valid plans.

The planning task can be divided into two closely related subtasks. One subtask
involves finding the actions needed to solve each objective of the procedure. The other
subtask involves resolving any conflicts between the solution and other objectives. To
demonstrate these ideas more clearly, consider a slightly more complex problem of
taking milk from a refrigerator (see figure 3.3).

The achievement of the objective ‘I have the milk’ demonstrates some of the
complexity of selecting actions. The objective cannot be met by selecting one action
to bridge the gap between the start state and the goal state. In this case the objective
must be achieved by two steps, opening the refrigerator and then taking the milk. In
general, any number of steps may be required to achieve a single objective.

The achievement of the second objective ‘the refrigerator door is closed’ demon-
strates some of the complexity of conflict resolution. One way to achieve this objective
is to decide to keep the refrigerator door closed throughout the procedure. However,
this strategy conflicts with the sub-plan to take the milk. The planner must resolve
this conflict by adding a new action to close the refrigerator door.

One of the objectives of this thesis is to allow the planning tasks in OPS problems
to be solved using AI planning techniques.

3.1.2 Safety

In many AI planning domains, a procedure is acceptable if it achieves its objective. In
OPS, an operating procedure must also be safe. The safety task involves preventing
the generation of unsafe operating procedures.

There are many ways in which an operating procedure can be unsafe. For example,
the procedure may be hard to follow, or out of date, or less inherently safe compared

Figure 3.3: A plan to take milk from a refrigerator
Figure 3.4: A procedure to add methane to a vessel to an equally valid procedure. Many of these ideas of safety cannot be easily evaluated during procedure synthesis.

Existing OPS systems use a simplified view of safety. A procedure is viewed as safe if it does not cause an unsafe situation to occur. An unsafe situation can be represented by a list of facts which should not all be true at the same time. For example, it might be unsafe for a heater to be on at the same time as a pump is off.

The safety task is made up of two closely related subproblems. One subtask is to decide whether a procedure is safe. The other is to modify an unsafe procedure to make it safe.

To demonstrate these two subtasks, consider the problem of adding methane to a vessel containing chlorine (see figure 3.4). The particular difficulty of this problem is due to the safety constraint that methane and chlorine should not mix unless the system temperature is high. This example comes from Fusillo and Powers (1987).

To create the procedure in figure 3.4, planning is used to solve the goal contains(methane) = true by adding the step `flow methane'. Safety checking shows the new step to be unsafe because it allows methane and chlorine to mix while the system temperature is low. The procedure would be made safe if the system was heated or the chlorine was removed before the step `flow methane'. Taking the first of these options, the planner adds a new action `start heater'.

Many current OPS systems do not resolve safety problems in an algorithmically complete way. A common assumption involves the proximity of the action which threatens to cause a safety problem, the threat, and the action which is added to prevent the problem, the guard. In the methane and chlorine example, flow methane is an example of a threat and start heater is an example of a guard. In the example, the guard and the threat come one after the other. OPS systems often mistakenly assumed that there will never be the need to have an action between the guard and
the threat. In reality, safety problems sometimes have to be resolved a long way in advance. For example, consider purging a vessel as a response to a safety problem. Late in the procedure, the pipes around this vessel may be committed to carrying specific chemicals. As a result, it may be necessary to purge the vessel nearer the start of the procedure before the pipe work is committed to other flow tasks.

One of the achievements of this project has been to develop a safety strategy which avoids this assumption and is complete. The safety maintenance algorithm is described in chapter 7.

### 3.1.3 Valve sequencing

Valve sequencing is the task of creating or blocking a flow of chemical by using a sequence of pump and valve operations.

The term *valve sequencing* was used in Lees (1980) to describe the creation of operating procedures involving just valve operations. Arguably the task should be called flow analysis, or some other name, because valves need not be the only plant items to be operated by a valve sequencing tool. For example, in Foulkes et al. (1988) it is acknowledged that pumping is often required to create a flow of chemical.

The effect of opening or closing a valve is dependent on the state of the plant. For example, consider opening valve *a* in the plant section shown in figure 3.5. The state of the plant after opening valve *a* is dependent on the state of valves *b* and *c* before *a* is opened. The possible effects of opening valve *a* are listed in table 3.1.

An important problem in OPS is the conflict between valve sequencing and safety maintenance. Valve sequencing seems to require strong commitments about the state of the plant when valve operations are to take place. Safety maintenance seems to
require the ability to add new actions into the middle of an existing procedure in order to guarantee the safety of that procedure. Adding a new action in this way can change the state of the plant and so can affect the effect of the valve operations already in the procedure.

Chapter 6 presents a new valve sequencing algorithm. The algorithm does not require complete state information. As a result it overcomes the safety checking problems of earlier work. The algorithm is designed for use in an OPS system based on AI planning techniques.

### 3.2 Simplifying assumptions

All previous OPS systems make similar assumptions about the behaviour of a plant. The objective of this project is to improve upon the techniques used for OPS but to work within the assumptions that have previously been considered reasonable. Therefore, it is important to understand the assumptions that are commonly made.

1. An operating procedure can be modelled as a sequence of actions. Examples of actions that might be found in a procedure include closing a valve, starting a pump and telling a controller to achieve a particular set point.

2. All the effects of an action are assumed to appear at the same time. For example, opening a valve cannot be modelled as causing a tank to overfill some minutes after the valve is opened.

3. There is no specific model for carrying out two actions in parallel. Hence it is assumed that the steps in a procedure will either be carried out in sequence or

<table>
<thead>
<tr>
<th>Plant State</th>
<th>Resulting propagation of Hydrogen through the plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valve b</td>
<td>Valve c</td>
</tr>
<tr>
<td>closed</td>
<td>closed</td>
</tr>
<tr>
<td>open</td>
<td>closed</td>
</tr>
<tr>
<td>closed</td>
<td>open</td>
</tr>
<tr>
<td>open</td>
<td>open</td>
</tr>
</tbody>
</table>

Table 3.1: The effect of opening valve-a in a section of plant
that the effect of carrying out a set of steps in parallel will be the same as the effect of carrying out those steps in sequence.

4. It is assumed that a plant can be modelled qualitatively. For example, valves are usually modelled as just being open or shut. Similarly, the temperature of a vessel is often limited to a few qualitative values.

5. An operating procedure is assumed to be safe if it is not unsafe when the plant behaves as expected and the procedure is carried out correctly. Current OPS systems do not attempt to create procedures that are particularly immune to human error or plant failure.

6. Safety is defined in terms of unsafe situations. It is assumed that all relevant unsafe situations are expressed in the plant model.

7. One type of unsafe situation involves the formation of a dangerous mixture of chemicals, for example a mixture of hydrogen and oxygen. It is assumed that a mixture of chemicals is dangerous if it contains a particular combination of species. Concentrations and proportions are ignored.

By allowing the system to ignore the concentrations and proportions of the components a mixture, it is assumed that the system will err on the side of caution. However, with this approach it is not possible to represent a purge operation that only reduces the concentration of a certain species. Instead, all purge operations are modelled as completely effective. This may lead to the acceptance of some situations which are unsafe in practice.

8. The idea of safety is independent of time. An archetypal safety constraint is 'hydrogen and oxygen should not mix'. Current systems cannot represent safety constraints of the form 'food should not be left out of a refrigerator for a long time'.

9. The formatting of an operating procedure is not important. It is the steps in the procedure which matter.
10. The model assumes that a flow of a chemical can be created by opening the valves between some source and some sink of that chemical. The valves can be opened in any order and pumping is not required.
Chapter 4

Literature Review

This chapter provides an extensive review of the OPS literature. The review is guided by the analysis of OPS problems given in Chapter 3. The review is primarily concerned with three questions. How have the planning, valve sequencing and safety aspects of OPS been addressed in the past? What are the limitations of the techniques that have been used? What assumptions are implicit in these techniques?

The review divides existing work into five paradigms. The paradigms represent the different methods for selecting and ordering the actions which make up an operating procedure. This classification provides new insights by relating seemingly different work.

This chapter is divided into seven sections. The first section provides a brief history of OPS research. The next five sections describe respectively the state graph paradigm, the forward chaining paradigm, the action synergy paradigm, the action ordering paradigm and the scheduling paradigm. The chapter then ends with a short conclusion.

The sections on forward chaining and action synergy are most relevant to the research described in this thesis. The other paradigms are examined in order to provide a complete picture of the relationship between the thesis and existing work.
Figure 4.1: The history of OPS research
4.1 History of OPS research

The earliest work on operating procedure synthesis (OPS) is presented in Rivas and Rudd (1974). Computer based analysis, rather than synthesis, of operating procedures was considered six years earlier by Peck (1968).

The work by Rivas and Rudd (1974) is based on early Artificial Intelligence research, particularly the development of the GPS (General Problem Solver) system described in Ernst and Newell (1969). The work is also based on automatic plant design work as described in Sirola et al. (1971).

After the publication of the seminal paper by Rivas & Rudd very little work was done on OPS for about twelve years. Only five papers were published in this time (O'Shima, 1978; Ivanov et al., 1980b, 1980a; Kinoshita et al., 1981, 1982). These are short conference papers which outlined ideas for future work.

There was new interest in OPS around 1987. At the time, computers were thought to be cheap and powerful enough to make OPS practical. In 1988, a special issue of Computers in Chemical Engineering was published with OPS as one theme. This burst of interest died down around 1991 when it was no longer clear how the then current OPS paradigms could be advanced.

In the last five years, OPS work has moved away from the procedure synthesis towards procedure analysis and procedure optimisation. Recent OPS systems focus on areas where the choice of steps for a procedure is heavily constrained by design of the plant.

4.2 State graph paradigm

State graph planning is the simplest form of procedure synthesis. In this approach, a plant is modelled by a graph describing all its possible states. Nodes in the graph represent plant states and directed arcs in the graph represent the use of an action to move from one state to another. An example of a state graph is shown in figure 4.2.

The state graph model of plant operation has been used both for the synthesis and the analysis of operating procedures. This section is divided into two parts. The
4.2.1 Procedure synthesis

If a plant is represented by a state graph then an operating procedure for the plant
can be represented by a path through the graph. The path starts at the node which
represents the initial state of the procedure. The path traverses a sequence of arcs
corresponding to the sequence of steps in the procedure. The path finishes at a node
where the objectives of the procedure are satisfied. Operating procedure synthesis is
the task of finding a path which satisfies a given set of requirements.

One acceptable path searching algorithm is shown in figure 4.3. This algorithm
assumes that all modelling problems and safety issues are resolved when a state graph
is created, e.g. every path through a graph represents a valid operating procedure
for some objective. Optimal operating procedures can be created using a similar
algorithm if the arcs in the state graph are weighted with their cost.

The limitation of state graph approach is the difficulty involved in creating and
maintaining the state graph model of a plant. The size of a state graph is related exponentially to the number of variables in the planning domain. For instance, the plant used as an example in Rivas, Rudd, and Kelly (1974) contains 17 valves. Even if each valve may only be open or closed, there will be \(2^{17}\) or 131,072 states. Though some of these states are impossible and can be removed, extra variables are needed to model other features like contents of the pipes in the plant. A practical model of the plant will contain at least a hundred thousand states. It is not realistic to expect a team of people to enter this huge graph by hand or to expect the resulting graph to be free from errors.

The state graph approach to procedure synthesis was introduced independently by Ivanov et al. (1980b, 1980a) and by Kinoshita et al. (1981, 1982).

4.2.2 Procedure analysis

State graph planning is largely unsuitable for procedure synthesis because of the difficulty involved in creating and modifying a large state graph by hand. The state graph representation is usable for procedure optimisation and procedure analysis because the existence of a known procedure drastically reduces the set of interesting nodes in a graph.

Peck (1968) describes a method to analyse the probability of success of an operating procedure. The procedure is represented by a state graph. Nodes and arcs are added to the graph to represent the events corresponding to the failure of each action in the procedure. Subgraphs are added to represent the methods of recovery from failure. Finally, the arcs in the graph are weighted with probabilities so that the sum of the weights of all arcs leaving each node in the graph is always 1.0. The probability assigned to an arc leading from a node represents the chance that the event described by the arc will occur at the node.

This model is examined using Monte Carlo simulation. The simulation approximates the probability of failure for the procedure and reveals the most likely failure points. From this work, it is also possible to calculate the average time a procedure will take to execute given that steps in the procedure may fail.
Lakshmanan (1992) describes a method for optimising an operating procedure. The state graph of all the possible methods for performing a procedure task is constructed from the state graph of each step in a procedure using ideas presented in Lakshmanan and Stephanopoulos (1988a). Rules are then applied to the state graph to prune down the set of possible operating procedures to a set of efficient procedures.

4.3 Forward chaining paradigm

The forward chaining paradigm is in some ways very similar to the state graph paradigm. The two paradigms consider the same search space, the space represented by a state graph. They also use the same basic search algorithm involving choosing an action, finding the effects of that action and then planning on from the resulting state.

The difference in the forward chaining approach is that the state graph is not represented explicitly during planning. Instead, an OPS system is given enough knowledge to implicitly generate and explore the state graph during synthesis.

The modelling language used in the forward chaining paradigm is based on the idea of an operator. Operators in the forward chaining paradigm are similar to the STRIPS operators used in AI planning. A small set of operators together with an initial plant state can describe a very large state graph.

The operator model used in forward chaining consists of three components: a model defining when each operator is applicable, a model of the effects of each operator and a model to decide when the state of the plant is safe. The basic forward chaining algorithm is shown in figure 4.4. In this algorithm, the three models are used in steps 2, 5 and 7 respectively.

The forward chaining algorithm of figure 4.4 has three important properties.

1. New actions are always added to the end of a procedure. In other words, the $n^{th}$ action added to the procedure will also be the $n^{th}$ action in the finished procedure.

To simulate the effect of any operating procedure one can simply simulate the
ForwardChainingPlan\( (\text{start, goal, operators, seen, procedure}) \)

1. if \( \text{goal} \subseteq \text{start} \) return success.
2. \( \text{options} = \text{set of all } \text{op} \text{ such that } \text{op} \text{ is an instantiation of some operator in } \text{operators} \text{ and all the preconditions of } \text{op} \text{ are true in } \text{start} \).
3. if \( \text{options} = \emptyset \) then backtrack.
4. \( \text{action} = \text{choose an action from } \text{options} \).
5. \( \text{new} = \text{simulate the effects of } \text{action} \text{ in the state } \text{start} \).
6. if \( \text{new} \notin \text{seen} \) then backtrack.
7. if \( \neg \text{safe(new)} \) then backtrack.
8. call ForwardChainingPlan\( (\text{new, goal, operators, seen} \cup \{\text{new}\}, \text{procedure} + \text{action}) \)

Figure 4.4: The forward chaining planning algorithm

The forward chaining algorithm considers the space of all valid procedures which have the required initial state. From this space, the algorithm chooses a procedure which has the desired effects. The forward chaining algorithm is algorithmically correct and complete because all possible procedures are considered. Further, difficulties that occur in more complex OPS systems are avoided, e.g. the need to resolve conflict between objectives and the need to resolve safety problems.

3. Step 6 in the algorithm is necessary to prevent looping. If it were not for this step then plans of the form 'open valve-a, close valve-a, open valve-a etc.' would be explored.

All current forward chaining systems use operator selection heuristics to improve on the performance of the basic forward chaining algorithm shown in figure 4.4. These
heuristics attempt to avoid considering the actions which will not help achieve the
given objectives.

When 'intelligent' operator selection is used in a system, the system loses the sec-
ond property of the basic forward chaining algorithm. The system does not consider
all possible procedures and so is not guaranteed to be algorithmically complete.

All current systems are incomplete for two reasons. First, they fail to select the
actions needed to solve some problems where the solutions to two goals conflict in a
particular way\(^1\). Second, the actions needed to resolve some safety problems are not
selected or are selected too late. As discussed in chapter 3 (section 3.1.2), the repair
of an unsafe procedure may require adding new actions into the body of the existing
procedure. The evaluation of the safety in forward chaining OPS is dependent on the
assumption that actions will only be added to the end of the procedure. The forward
chaining algorithm would have to be changed significantly to allow new actions to
be added to the middle of a procedure. No current forward chaining OPS systems
attempt this.

In conclusion, the ideal forward chaining algorithm is correct and complete but too
slow to be practical. Current forward chaining OPS systems reduce the search space
by using intelligent operator selection algorithms. The operator selection algorithms
used by current systems are incomplete.

4.3.1 Rivas and Rudd

Rivas and Rudd (1974)\(^2\) is the earliest forward chaining OPS system. The system
is designed specifically for valve sequencing tasks. The procedures produced by the
system only involve opening and closing valves. The goals given to the system describe
the need for a flow or a purge or the need to isolate a unit or to trap a chemical.

As an example of the problem solving ability of the system, consider the plant
in figure 4.5 and the operation of moving dough into the mixer. In the initial state

---

\(^1\)As a point of interest, McDermott (1996) describes a forward chaining algorithm that uses
intelligent action selection and is complete. The algorithm was developed to solve planning problems
and has not been tried in OPS domains.

\(^2\)This work is also reported in Rivas and Rudd (1975).
RivasRuddPlan(start, goal, operators, procedure)
1. differences = goals - start.
2. if differences = ø then success.
3. options = set of all op such that op is an instantiation of some operator in operators and all the preconditions of op are true in start and (∃g ∈ differences)(related(op, g) and safe(simulate(op, current))).
4. if options = ø then repair plan safety.
5. action = select any action from options. This choice is not remade through backtracking.
6. new = simulate(action, start).
7. call RivasRuddPlan(new, goal, operators, procedure + action )

Figure 4.6: Rivas and Rudd’s OPS algorithm

the mixer is full of cleaning water. The water should not come into contact with the dough.

The system considered here can produce the three step procedure to open the drain valve in order to remove the water, close the drain valve to avoid losing the dough and then open the inlet valve to add the dough. This procedure is created in response to the single goal ‘trap the dough in the mixer’.

The high level synthesis algorithm used in the system is given in figure 4.6. The implementation of this algorithm is essentially the implementation of the three forward chaining models (operator selection, action effects and safety). The objective of this section is to describe the interesting features of these three models.
Operator selection

The operator selection model finds actions which are possible, safe and positively related to some outstanding goal. Safety is considered during operator selection, rather than later in the cycle, because of a need to constrain the set of possible actions. For implementation reasons the system will not backtrack and so the set of possible actions must be as small as possible.

An action is considered to be ‘positively related’ to a goal if it satisfies a rule specific to the type of goal. For example, an action is positively related to a goal to create flow of a chemical through a specific point if the action creates a flow into that point or if the action creates a flow out of that point. Consider the goal to create a flow through the vessel in figure 4.7. Opening valve 3 is positively related to the goal because this action creates a flow out of the vessel. However, opening valve 1 or valve 2 is not positively related to the goal because opening either of these two valves on their own will not create a flow. In general, the system cannot solve flow goals which require two or more valves to be operated. The system is incomplete.

Operator selection requires an agenda of the goals which are left to be solved. In Rivas and Rudd (1974) the agenda is defined as the set of goals not satisfied in the current state. In figure 4.6 this agenda is called differences. Synthesis stops when the agenda is empty and so all goals have been solved. As a result, the system only generates procedures which satisfy their goals and so the system is algorithmically correct.

A goal may reappear on the agenda after it has already been solved once. This will happen if some action negates the original solution to a goal. In some cases, procedures will gain redundant actions as a result. In rare cases the system will fail to terminate. As a worst case example, consider the plant in figure 4.8 and the three objectives to (1) isolate the section of pipe between valves 1 and 2, (2) create a flow
Figure 4.8: A case where looping may occur

from the inlet and (3) create a flow to the outlet. The correct solution to this problem is to open valves 3 and 4. The system may start by opening valves 1 and 2 to create the flow. The system will then close valves 1 and 2 to isolate the pipe section. The actions to open valves 1 and 2 are now redundant because the valves have been closed again. The system does not perform loop checking and so there is no reason why the cycle of opening and closing valves 1 and 2 should not continue indefinitely.

Action effects

The model of action effects takes the form of a hard coded function. The function essentially analyses a plant state to find where each chemical species is flowing and where it is trapped. This function is described in Rivas et al. (1974).

It is significant that this action effect model can deduce that closing a valve will block an existing flow. Later OPS systems have difficulty protecting a flow because they cannot easily reason about when a flow becomes blocked.

Safety

As in all forward chaining systems, a procedure is considered safe unless some unsafe situation is caused by the procedure. The safety of a procedure is evaluated through the simulation of the effects of each action.

In the system considered here, unsafe situations are represented by logical statements. There are five different types of situation that can be defined as unsafe: (1) the mixing of two or more chemicals, (2) a flow between two units, (3) the blocking of a particular flow, (4) a particular chemical reaching a particular outlet, (5) a chemical entering a particular unit. Most later OPS systems cannot represent all these five types of safety constraint; the exception is Strimaitis (1987).

When needed, the system will add a step to drain a unit in order to prevent
the creation of an unsafe mixture of chemicals. Draining is modelled as completely removing a chemical from a unit. A draining step is added if all actions that are positively related to a goal are unsafe because they cause an unsafe mixture to form.

The system will not act to avoid unsafe situations that do not involve mixing. For example the system will not reroute a protected flow of a chemical in order to prevent the flow from becoming blocked.

### 4.3.2 Fusillo and Powers

Fusillo and Powers (1987, 1988a, 1988b) describe an attempt to solve general OPS problems using an adaptation of the procedure synthesis algorithm given in Rivas and Rudd (1974). In this review, each of the three papers by Fusillo and Powers will be considered individually.

An example from Fusillo and Powers (1987) demonstrates the kind of problems this work can solve. The example is based around the plant in figure 4.9. The four goals are to start the heater, compressor, methane inlet and chlorine inlet. There are three significant safety constraints: (1) the heater may not be started while the compressor is off, (2) chlorine and methane should not mix while the system is cold and (3) methane should not be added to chlorine.

When run on this problem, the system proposes to start the compressor then the
FP Plan(start, goals, operators, procedure)
1. actions = FP.SelectActions(goals, operators)
2. return FP.OrderActions(start, actions, procedure)

FP.SelectActions(goals, operators)
1. if goals = 0 return 0.
2. g = select any g from goals.
3. if g is true in start goto step 7.
4. options = set of all op such that op is an instantiation of some operator in operators and op achieves g.
5. if options = 0 then fail.
6. action = choose an action from options.
7. return {action} U FP.SelectActions(goals - g, operators).

FP.OrderActions(start, actions, procedure)
1. if actions = 0 then succeed.
2. act = choose an action from actions such that all the preconditions of act are true in start.
3. new = simulate the effects of act in the state start.
4. if ¬safe(new) then backtrack.
5. call FP.OrderActions(new, actions - act, procedure + act)

Figure 4.10: Fusillo and Powers OPS algorithm

heater and then methane feed and finally the chlorine feed.

Fusillo and Powers (1987)

The algorithm used in Fusillo and Powers (1987) is shown in figure 4.10. The most striking difference between this algorithm and the work in Rivas and Rudd (1974) is that action selection and action ordering are separated. This separation limits the system's ability to select the actions which will make up a procedure.

The operator selection model used in Fusillo and Powers (1987) selects exactly one action for each goal that is not true in the start state. This prevents looping because there is a limit on the number of steps in a procedure.

By performing operator selection before synthesis, the system is less able to re-
solve conflict between actions. Consider the procedure shown in figure 4.11. The procedure involves compressing a chemical but finishing with the chemical's temperature unchanged. The compressor heats up any chemical it operates on and so some form of cooling will be required during the procedure. However, because the chemical is cool at the start of the procedure, the operator selection algorithm used by Fusillo and Powers' system will not propose a needed cooling step. Instead, the system will generate an incorrect procedure containing the single action 'start compressor'.

The operator selection algorithm is also incomplete. If an OPS problem has exactly one goal then the problem cannot be solved if the goal is two steps from the start state, e.g. the procedure in figure 4.12 cannot be generated. This is similar to the limitation which prevents Rivas and Rudd (1974) from opening two or more valves to create a flow.

The model of safety used in Fusillo and Powers (1987) is similar to the model of safety used in Rivas and Rudd (1974). In Fusillo and Powers (1987), unsafe situations are represented by lisp equations. These equations are called *global constraints*. A state is unsafe if any global constraint evaluates to true in any state produced by the procedure.

If adding a new action to a procedure causes an unsafe state to be formed, the system simply backtracks. This approach is incomplete. For example, consider the plant in figure 4.9 and the goal to start the heater. The heater cannot be started if the compressor is off but the system will not automatically propose the action to start the compressor because starting the compressor does not achieve a goal directly.
Fusillo and Powers (1988a)

Fusillo and Powers (1988a) describe an extension to the action selection strategy in the earlier OPS system. The extended system can predict some of the actions that will be needed to resolve safety problems during synthesis.

The strategy is based around creating a hypothetical state for an OPS problem. In this hypothetical state the goals of the problem are true and any literal in the initial state which is not contradicted by the goal state is also true. An example is given in figure 4.13. Note that adding new goals to a problem changes the hypothetical state for that problem. For example, in figure 4.13 if a goal was added to require the compressor to be on in the goal state then the compressor would be on in the hypothetical state.

The system described in Fusillo and Powers (1987) will select an action for every difference between the hypothetical state and the initial state (every goal). Hence the hypothetical state can be viewed as the state that the system intends to achieve. Synthesis will usually fail if the hypothetical state is unsafe because the system is prevented from achieving an unsafe state.

The strategy used in Fusillo and Powers (1988a) is to examine the hypothetical state and add new goals to make the state safe. For example, if a heater cannot be turned on without a compressor being on and if the hypothetical state violates this safety constraint then the system will add a new goal to turn the compressor on given that the compressor is off in the initial state.

---

3 If no action in a procedure has a side effect then carrying out the procedure will result in the hypothetical state. Hence, if the hypothetical state is unsafe then synthesis will fail given that no actions have side effects. However, if the actions in a procedure do have side effects then carrying out the procedure will produce some alternative state, rather than the hypothetical state. In some rare cases, this alternative state is safe, and so achievable, even though the hypothetical state is unsafe, and so unachievable.

4 The work in Fusillo and Powers (1988a) considers specifically the problem of adding purge
The strategy represents a significant attempt to allow a forward chaining OPS system to work with safety constraints. The strategy has two limitations.

1. The system assumes that each action will have only one effect. The safety problems caused by the side effects of actions are not considered. For example, consider where a step is added to a procedure to purge a particular vessel with nitrogen. If nitrogen is not in the vessel at the start of the procedure then the system will not evaluate whether it is safe for nitrogen to be in the vessel at the end of the procedure.

2. In some situations the system will fail to propose all the actions needed in the creation of a safe procedure even though the hypothetical state is safe and each action has only one effect and a safe procedure exists.

Consider a simple domain involving a hopper full of gravel and two trucks. In the initial state, the first truck is being filled with gravel and the second truck is away. The goal is to have the second truck being filled with gravel and the first truck away. The safety constraints prevent both trucks being in the loading bay at the same time and also prevents the gravel hopper from being open while no truck is in the loading bay. The state space for this problem is depicted by the Karnaugh map (see Karnaugh, 1953) in figure 4.14. Adjacent cells in this map represents states joined by a single action. Crossed out cells represent unsafe states.

The only safe solution to the truck and hopper problem is to close the gravel operations to prevent dangerous mixtures of chemicals forming. The heater and compressor problem cannot be solved using the system described in the paper but only due to the implementation of the methodology used in the system rather than the methodology itself.
hopper, drive the first truck away, park the second truck and then open the hopper again. The difficulty for the system considered here is the need to operate the gravel hopper. It is not apparent from the start and goal states of the procedure that the hopper will need to be operated.

Fusillo and Powers (1988b)


If an unsafe procedure is created then the procedure is saved. If no safe procedure is found during synthesis then one of the saved procedures is recalled. The safety of this procedure is analysed and sufficient actions are selected to resolve the safety problems that the procedure contains. The action ordering phase is then restarted using the newly selected actions together with the original action set.

This approach, as opposed to the implementation of the approach, is apparently algorithmically correct and complete. However the approach may also be very slow. The ordering function requires at worst exponential time with respect to the number of actions in a procedure. The ordering function will have to be called often in order to create a complex procedure.

The implementation of the approach is limited so that only one action can be selected to change the state of any one literal. This limitation may be imposed to prevent a procedure containing redundant actions. Due to this limitation, the trucks and the hopper problem cannot be solved because the state of the hopper must change twice during the procedure, it must be opened then closed.

The paper also considers a more complex operator model. In this model, the effect of each step is dependent on the state of the plant when the step is carried out. The relation between the effects of a step and the state of the plant is defined by an arbitrary function. The system has little understanding of the functions that model each step. As a result, the side effects of later steps may be allowed by the system to negate the important effects asserted by earlier steps. This is demonstrated by the example in Fusillo and Powers (1988b).
It has often been assumed that the forward chaining paradigm allows arbitrarily complex action models. The work considered here demonstrates that arbitrarily complex actions are not feasible in practice. If an operator is represented by an arbitrarily complex model then the system has no way to prevent a step in a procedure from having a particular undesirable side effect.

4.3.3 Tomita, Hwang, O’Shima and McGreavy

The OPS system described in Tomita et al. (1989b), Hwang et al. (1991)\(^5\) takes a novel approach to plant modelling in order to create a more powerful OPS system. On the surface this system appears to be very different to the work discussed so far. In practice the system is similar to Fusillo and Powers (1987) in that it involves an action selection phase and an action ordering phase. However, this work seems to have been developed independently of Fusillo & Powers work.

As an example of the problem solving ability of the system, consider the startup of the column in figure 4.15. Safety constraints dictate that the heater cannot be started if the pump is off. Physical constraints prevent the column from being started before the heater is started. Given the single goal to start up the column, the system considered here can generate the procedure in figure 4.16.

\(^5\)The work described in Hwang et al. (1991) is based on a Japanese Language paper (Hwang, Tomita, & O’Shima, 1988). The work is also reported in Tomita, Hwang, and O’Shima (1989a). There is an early paper which I have not been able to obtain, Tomita, Nagata, O’Shima, and McGreavy (1986).
1. Start the flow of oil to pump-201.
2. Start the flow of cooling water to pump-201.
3. Open v2.
4. Open v3.
5. Start pump-201.
6. Start the flow of fuel to heater-201.
7. Start heater-201.
8. Start column-201.

Figure 4.16: The procedure to start the reboiler

The review of this system is broken into three parts. The first part will consider plant modelling. The second part will examine the operation selection strategy used. The final part will examine the two operator ordering strategies that have been proposed.

**Plant modelling**

The work is centred around the operations of units in a plant. In the work, a unit is called a *node* (Tomita et al., 1989b) or a *PRIMITIVE-SCOPE* (Hwang et al., 1991). Examples of nodes are reactors, columns and pumps. Valves and pipes are handled separately and are not counted as nodes.

Each node has only two states, on or off. However, a node may be used to achieve a number of different effects. The effect of a node is dependent on the state of the other units in the plant and on valves connecting to those units. The effects of a node are described by a set of *functional rules* which are if-then rules. Only one functional rule is applicable in any given plant state. The functional rules are used both for the action selection model and the action effects model.

Safety is modelled separately by a set of what are apparently preconditions for the operation of individual unit. These preconditions are called *OPEN-PATH-CONDITIONs* in Tomita et al. (1989b). It is not clear how a unit specific model of safety can represent safety constraints involving a section of plant, e.g. ‘there should always exist a flow path between units X and Y'.
The operation of a single node is represented by a sequence of steps in an operating procedure. The sequence involves performing preparatory work, such as obtaining permission to work, and then opening the necessary valves and finally operating the node itself. The sequence is dependent on the desired effect of the unit and is generated by simply formatting stored knowledge about that unit.

The algorithm for translating the operation of a node to a sequence of steps is possibly too simplistic. It assumes that the route for the flow of chemical into a unit (node) will be fixed depending on the role of that unit. It is also assumed that the flow route will be clear of possible contaminants. The large amount of valve sequencing work in the OPS literature suggests that finding a safe route for a flow is the hard part of some practical problems.

It is also not clear how this model of a unit can be used to represent batch processes involving a number of steps in one vessel. For example, consider modelling a process in a single vessel to make strawberry yogurt from yogurt culture by first making plain yogurt from the culture. In this process, the vessel has one of at least three states depending on its contents.

Operator selection

The synthesis algorithm in the series of papers considered here can be thought of as a development of the synthesis algorithm in Fusillo and Powers (1987). As in the earlier work, operator selection and action ordering are performed in two separate phases. Operator selection involves a prediction of the steps that will be needed in a procedure and action ordering involves a forward chaining search so that the procedure can be simulated.

The operator selection algorithm is based on a backward chaining search. The basic algorithm involves selecting an action to achieve each goal and also each precondition of each action already selected. The algorithm used in the paper is more sophisticated than this basic algorithm. The basic algorithm may solve a goal in a way that prevents the solution to some other goal. The system described in the paper prevents conflict between actions by developing the idea of a subgoal state.
BACKWARD(goals, subgoal_state, functional_rules)

1. if goals - subgoal_state = ∅ then succeed returning subgoal_state.
2. g = select any g from (goal - subgoal_state).
3. rule = choose a rule from functional_rules such that the g ∈ effects(rule). The effects of a rule are described by the ‘then’ part of the rule.
4. new = effects(rule) ∪ preconditions(rule). The preconditions of a rule are described by the ‘if’ part of that rule.
5. if new is inconsistent with subgoal_state, e.g. if x ∈ new and ¬x ∈ subgoal_state, then backtrack.
6. if new is inconsistent with goals then backtrack.
7. return BACKWARD(goals ∪ preconditions(rule), subgoal_state ∪ new, functional_rules).

call as BACKWARD(goals, goals, functional_rules).

Figure 4.17: Operator Selection used in work by Tomita and others

The subgoal state is a consistent state in which the preconditions and effects of all currently selected actions are true. Actions are only selected if they are consistent with the subgoal state.

A simplified version of the operator selection algorithm is shown in figure 4.17. The full version of this algorithm is given in a preprint of Tomita et al. (1989b). The algorithm is called BACKWARD in Tomita et al. (1989b) and SUBGOAL-MAKER in Hwang et al. (1991).

The advantage of the operator selection strategy used here is an ability to solve problems where a goal is more than one step away from the start state. Figure 4.12 demonstrates a problem that can be solved using this approach but cannot be solved using earlier work.

The limitation of this operator selection algorithm is that each unit can change state only once during a procedure. For example, the system cannot model a procedure where a vessel is unclean in the initial state and then cleaned during the procedure and finally used as the mixing vessel for a batch. The limitation is inherent in the way the subgoal state is used to develop a consistent strategy for operating the plant.
Action ordering

The system described in Tomita et al. (1989b) provides three different action ordering strategies.

1. A strategy that involves cycling through the agenda of nodes to be operated. On each pass, every action is selected that has its OPEN-PATH-CONDITIONS satisfied.

2. A strategy that is guided by the OPEN-PATH-CONDITIONS for the operation of each node. If a node cannot be operated because its OPEN-PATH-CONDITIONS are not satisfied, then a new node will be chosen to help satisfy these conditions.

3. A strategy using expert system rules to select operators. Two expert systems are described, a backward chaining system presented in Tomita et al. (1989b) and a forward chaining system in Hwang et al. (1991).

The relative advantages of the different ordering strategies is not clear from the work. The impression is that more complex strategies produce procedures with a more logical ordering.

4.3.4 Aelion and Powers

Aelion and Powers (1991) continue on the work started by Fusillo and Powers (1987). The synthesis algorithm used in this work is shown in figure 4.18. The algorithm is based on the STRIPS planning algorithm described in Fikes and Nilsson (1971).

The operator selection strategy used in the system considered here can be viewed as a development of the operator selection strategy used in Tomita et al. (1989b). As in the earlier work, the system can solve problems where a goal needs to be solved by more than one action. However, the system can also solve problems where a unit must change state more than once during a procedure.

Line (c) of Figure 4.19 shows a procedure that can be generated by the system considered here but not by any earlier forward chaining OPS system. The procedure
AelionPowersPlan(start, goals, operators, procedure)

1. if goals \( \subseteq \) start then succeed.
2. \( g = \) choose a \( g \) from goals that is not in start.
3. \( options = \) set of all \( op \) such that \( op \) is an instantiation of some operator in operators and \( op \) achieves \( g \).
4. \( action = \) choose an action from \( options \).
5. \( added = \) set of all preconditions from \( action \) which are not true in start.
6. \( subplan = \) AelionPowersPlan(start, added, operators, procedure)
7. \( mid = \) simulate the effects of \( subplan \) in the state start.
8. \( new = \) simulate the effects of \( action \) in the state \( mid \).
9. if \(~\text{safe}(new)\) then apply rules to repair plan safety.
10. call AelionPowersPlan(new, goal, operators, procedure + subplan + action)

Figure 4.18: Aelion and Powers OPS algorithm

<table>
<thead>
<tr>
<th>INITIAL STATE</th>
<th>FINAL STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) pressure = high</td>
<td>pressure = low</td>
</tr>
<tr>
<td>open bleed</td>
<td>wait</td>
</tr>
<tr>
<td>(b) bleed = closed</td>
<td>bleed = closed</td>
</tr>
<tr>
<td>(c) pressure = high</td>
<td>pressure = low</td>
</tr>
<tr>
<td>bleed = closed</td>
<td>bleed = closed</td>
</tr>
<tr>
<td>open bleed</td>
<td>wait</td>
</tr>
<tr>
<td>close bleed</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.19: The procedure for bleeding a unit involves depressurising a unit. The procedure is required to achieve two conditions: that the pressure of the unit is atmospheric and that the bleed valve is closed. The procedure is difficult to generate for two reasons: (1) the solution to the two goals together requires more steps than the combined solution of the two goals individually (see figure 4.19) and (2) the bleed valve changes state twice during the procedure. The features which make this problem difficult appear frequently in OPS domains.

The system will solve the problem in figure 4.19 if the goal to depressure is chosen before the goal to close the bleed valve. The system tries all possible orderings of the goals of a problem and so the correct solution to this problem will be found. However, the system will also generate an incorrect solution. The system, as described, does not check that its goals are met by all the plans it suggests. The incorrect solution will occur when the goal to close the bleed valve is considered first and will be similar.
While the approach considered here is an improvement on earlier work, it is not complete. There is a well known problem with the STRIPS planning algorithm that the system considered here will fail to solve. The problem is known in the AI Planning literature as the Sussman anomaly (see Waldinger, 1977). A chemical engineering version of the Sussman anomaly is exhibited by the plant in figure 4.20. A difficult problem for this plant involves achieving a situation where the store is clean and a reaction is taking place in the bio-reactor. In the initial state for this problem, the store contains bacteria culture and the reaction vessel is cleaned. The reaction is created by placing the culture in the reactor and then continuously adding bio-food to the reactor. It is intended that the flow of bio-food should continue past the end of the procedure.

The correct operation of the plant in figure 4.20 involves moving the culture to the reactor, cleaning the store with cleaning-water and then starting the flow of bio-food. Any other order of instructions would either destroy the culture when the store was cleaned or would interrupt the reaction in order to clean the store. This procedure is difficult to generate because the two goals are interleaved. The system gets part way to starting the reaction, then cleans the store, then finishes starting the reaction. The system considered here assumes that goals will not have to be interleaved in this way and so cannot solve this problem.\(^6\)

The model of safety used in Aelion and Powers (1991) is similar to the model of safety in Fusillo and Powers (1987). However, the system will add new precondi-

---

\(^6\)As a point of interest, the system described in Tomita et al. (1989b) allows goals to be interleaved. Hence this earlier system is able to solve the version of the Sussman anomaly presented here.
tions rather than backtracking if an unsafe state is generated during synthesis. This handling of safety is similar to the handling of mixing constraint violations in Rivas and Rudd (1974). It is not complete because the actions to resolve a safety problem cannot be inserted into the portion of the operating procedure that has already been generated.

Later work by the group (Aelion, Kalagnanam, & Powers, 1992; Aelion & Powers, 1993) moves away from the procedure synthesis problem to look at methods for evaluating the safety of operating procedures.

4.3.5 Conclusions on the forward chaining paradigm

Recent forward chaining systems use backward chaining search for operator selection and forward chaining search for action ordering. Backward chaining search is required in operator selection to ensure that the preconditions of each action are satisfied. Forward chaining search is required in action ordering so that the system can simulate the effects of each action to make sure that the action is safe and its preconditions are satisfied.

The use of forward chaining search for action ordering limits the ability of the planner to select the actions needed in a procedure. Similar problems were faced in the development of AI planning. Current AI planning systems avoid the limitations of the forward chaining work reviewed here by using more sophisticated action ordering techniques. This foundation for these techniques is developed in Waldinger (1977).

There are two likely reasons why the action ordering strategy used by the AI planning community has not been used in the development of the forward chaining paradigm. First is the difficulty involved in checking the safety of a procedure. Safety checking algorithms used in current OPS systems rely on the use of forward chaining search to allow the procedure to be simulated so that the safety of the procedure can be checked. Second is the belief that forward chaining search allows the use of complex action effects models. It is not clear that this belief is justified. Systems that have used complex models have either been unable to propose the actions needed to achieve some effect that can be represented (Rivas & Rudd, 1974) or have been
unable to understand the interaction between the steps in a procedure (Fusillo & Powers, 1988b).

### 4.4 Action synergy

The action synergy paradigm is concerned specifically with valve sequencing problems. The approach is not suitable for more general OPS problems.

Forward chaining OPS systems expect an action to have a fixed set of effects or, in the case of Tomita et al. (1989b), a small number of different effects based on the state of the plant. Valve operations do not fit into this way of thinking. Opening a valve will often cause one of a large number of different effects to occur, depending on the state of the plant. As a result, current forward chaining systems have a limited ability to reason about valve operations.

Rather than reasoning about a valve as having a particular effect in a particular state, it is more constructive to reason about the effect of a set of valve operations. As an example, consider the plant in figure 4.21 and note that all valves in the plant are initially closed. Opening any one valve in this plant will have very little effect. Opening the sequence of valves \( \{a, b, c\} \) will create a flow of oxygen. This flow will also have the side effect of contaminating some of the pipes in the plant with oxygen. All these effects can be seen to result from synergy (interaction) between the three valve operations chosen.

Action synergy systems reason about the set of valve operations necessary to achieve some desired effect in a safe way. Action synergy systems usually have the following four components.

1. An algorithm to search for the sequence of valve operations needed to achieve...
some desired effect. For example, to create a flow of oxygen in figure 4.21, a system must select and open either the valves \{a, b, c\} or the valves \{a, d, g, e, c\}. It may be necessary to backtrack this step. For example, if the system happens to choose the flow route \{a, d, g, e, c\} and if a flow of hydrogen is also required then at some time the system will have to backtrack to choose the route \{a, b, c\}.

2. An algorithm to decide how to limit the side effect of a sequence of operations. For example, if a flow of oxygen is created by opening the valves \{a, b, c\} then the valves \{d, e\} should be closed so as to limit the flow of oxygen.

3. An algorithm to order the selected operations. This algorithm can be based on simple rules. For example, in creating a single flow, all valve closing operations should be performed before any valves are opened.

4. An algorithm to simulate the effects of a sequence of operations and to make sure that the sequence is safe.

Action synergy system are limited in their ability to plan. These system can be compared to a forward chaining system by considering the sequence of steps needed to create each flow or purge as a single forward chaining 'action'. Under this mapping, most systems operate like the forward chaining system described in Fusillo and Powers (1987). For example, action synergy systems have difficulty when a goal is more than one 'action' away from the start state. Also most, but not all, action synergy systems simply backtrack if a procedure is unsafe.

Figure 4.22 describes the start state for a procedure that no action synergy system can create. The goal is to achieve a state in which the tank is full of oxygen and hydrogen is flowing. To solve this problem, a system must address the goal to fill the
tank before the goal to restore the flow of hydrogen. Action synergy systems assume that all goals can be solved in parallel or that a goal ordering will be provided by the user.

4.4.1 O'Shima

O'Shima (1978) describes the first Action Synergy system. The only goal considered by the system is the task of creating a flow of chemical. A flow goal is solved by searching for a path from the given source to the given sink possibly via some specified intermediate components, perhaps a reactor. The search algorithm is similar to an algorithm for searching for a path through a maze. Actions are proposed to open the valves along this path and to close the valves around the path, so that the species does not stray.

When a flow is created, chemicals are moved into new locations in a plant. The system is able to predict the affected plant units without resorting to the complex simulation used in Rivas and Rudd (1974). Essentially, the only units affected by a flow are the units along the flow path.

4.4.2 Roach and Strimaitis

Strimaitis (1987) and later Roach et al. (1990) use action synergy techniques to solve the case study from Rivas and Rudd (1974).

An interesting idea from this work is that valves should have a default state. Most valves in the plant default to closed. Procedure synthesis involves a thought process in which a flow is created then all the valves in the plant are set to their default state (closed usually) and then a second flow is created. In this imaginary procedure, some valves are closed and then opened immediately. These redundant valve operations are removed in the real procedure so the valves remain open between the two flows. A procedure involving a sequencing of flow operations can be created using this methodology. The procedure will be efficient and will minimise the propagation of chemicals around the plant.

When the system must maintain a flow of a chemical, the valves along the initial
route are kept open. They can only be closed if some alternative flow route is found. Using this mechanism, the system is able to maintain a flow route during a procedure.

4.4.3 Foulkes, Walton, Andow and Galluzzo

Foulkes et al. (1988) provide a tool which is designed primarily to aid the plant operator to create a correct operating procedure rather than creating such a procedure for the operator. At each stage in a two stage cycle, the user asks the computer to move a chemical from one part of a process plant to another. The computer then generates for the user all the safe routes along which this chemical can flow. When a route has been chosen by the user, the computer updates its model of the plant and a new cycle starts.

As with O'Shima (1978), the system creates a flow of a chemical by finding a path through the plant from the given source point to the given sink. In contrast to earlier work, the system requires that a flow route contains at least one pump. The system will turn on this pump as part of creating the flow. The system will also order the actions involved in a flow of chemical so that, for example, all the valves in a flow are opened before any pump along the flow route is started.

Like Rivas and Rudd (1974), the system allows safety considerations to be specified. Safety is defined in terms of unsafe situations. To monitor the safety of a procedure, the system maintains the information about the current state of the plant so that safety constraints can be evaluated. No attempt is made to resolve a situation where a goal cannot be achieved because of the safety constraints on the system. Instead the user is told why a particular goal cannot be achieved. In effect, it is the responsibility of the user to find some modification that will later allow that goal to be solved.

Safety is evaluated in terms of units and their contents but not in terms of flow. It is not clear that the system can be asked to maintain a flow of chemical. This may cause problems with the use of pumping operations. Although the system will turn on an appropriate pump in order to create a flow of chemical, it is not clear that the appropriate pumps will be turned off when a flow path becomes blocked.
The final point of interest is the idea of a *normal* path. There are often many flow routes which are possible in theory but undesirable in practice. For example, a flow route that uses a reactor as if it were a simple pipe. Also, plants are often designed with the intention that the flow of a certain chemical should travel along a certain route and the plant operator is usually aware of this intention. The idea of a normal flow captures knowledge about the intended use of the plant and search is only used in situations where the plant cannot be operated in its intended way.

### 4.4.4 Lakshmanan and Stephanopoulos

Lakshmanan and Stephanopoulos (1990) provide a method for adding purge operations to prevent unsafe mixtures of chemicals from forming. The allowable mixing constraints are described as being *binary* and *qualitative*. The term *binary* is used to mean that if three chemicals cannot mix then two of these three chemicals also cannot mix. The term *qualitative* is used to mean that two chemicals cannot mix in any proportions. To put this into context, most valve sequencing systems consider arbitrary qualitative mixing constraints but don't consider purge.

The need for purge operations is found by examining a predicted *worst case* state for a procedure. The *worst case* state is defined as the state in which all new flows have been created and no existing flows have been blocked. Purge operations are added to clean the pipes that contain unsafe mixtures of chemical in the worst case state.

If a purge operation is found to be required, the system uses action synergy techniques to route the flow of purgative through the area to be cleaned. After this purge has taken place, the system will go on to try to achieve the objectives of the final procedure.

Many algorithms used in the system are shown to have polynomial time complexity. However, the time complexity of the system as a whole is left undecided. A simple example based on the plant in figure 4.23 demonstrates that the system will work forever when attempting to solve some problems. In effect, the time complexity of the system is infinite.
Consider the operation of the plant in figure 4.23. The plant may contain four chemicals: Chem-a, Chem-b, Chem-x and Chem-y. The pairs of chemicals which cannot be mixed are (Chem-a, Chem-x), (Chem-a, Chem-y) and (Chem-b, Chem-y). At the start of a particular OPS problem, Chem-a is flowing to the 'Waste Stream Out' as shown in the diagram. The objective is to create a flow of Chem-y to the 'Product Out'. One solution is to purge the common pipe first with Chem-b and then with Chem-x and finally to flow Chem-y. The difficult with this problem involves choosing a suitable purgative. The need to purge with Chem-b in the first operation is apparent but the system can then purge with Chem-x or Chem-a. The system always chooses the first available purgative (Lakshmanan & Stephanopoulos, 1990, section 6.2.7) and so can be made to choose Chem-a rather than Chem-x. The cycle of purging with Chem-b then Chem-a will be repeated indefinitely.

The time complexity of the algorithm is interesting because the paper appears to argue for polynomial time OPS. It is not yet clear that an OPS system can be designed to solve practical problems in polynomial time.

### 4.4.5 Conclusions

The development of the action synergy paradigm is less structured than the development of the forward chaining paradigm. This is partly because action synergy systems are seen to be easier to create.

In general action synergy systems have difficulty looking beyond a single flow operation. For example, no system is able to maintain a flow over a number of operations unless that flow should always exist.

It is not clear why Forward Chaining OPS systems have not included Action Synergy components to solve flow goals.
4.5 Action ordering systems

The OPS systems considered so far view action selection as perhaps the hardest part of synthesis. Action ordering systems focus on problems where the steps in a procedure are determined by the design of the plant. The philosophy is that a plant is designed to be operated in a particular way. Hence, the design of a plant should determine the equipment used for any particular operation.

These systems work by selecting an action to bridge each difference between the start state and the goal state. There is only one action for each kind of difference and so no backtracking is needed. The focus of the work is to order the action so as to satisfy the preconditions of each operation.

The Action Ordering paradigm is limited in that each unit can have only two basic states, an idle state and an active state. The objective of synthesis is either to start up the unit or shut down the unit or to leave the unit alone. Some situations cannot be modelled using this approach. For example, consider a procedure in which a vessel starts off containing air and then has to be purged with nitrogen and finally filled with methane and chlorine. This procedure cannot be modelled as a single action ordering task because the vessel must go through three states rather than two.

The main influences on action ordering work are the forward chaining systems described in Fusillo and Powers (1987) and in Tomita et al. (1989b).

4.5.1 Lakshmanan and Stephanopoulos

Lakshmanan and Stephanopoulos (1988b) describe an action ordering system. The role of the system is to create ordered sequences of valve operations. The system does not consider the mixing of chemicals and so valve operations never conflict with each other. Hence, without any safety constraints, the system could choose any order for a set of valve operations. However, the system allows safety constraints in the form of temporal constraints. A temporal constraint has constraints of the form ‘action x must come before action y’. The synthesis problem involves generating a sequence of operations that satisfies these constraints.

The system is based around the use of an intelligent user interface. The role
of the user interface is to elicit the start state, goal state and temporal constraints involved in a problem. The user interface is intelligent in that information can be specified at any level of abstraction and is translated automatically into the level of valve operations.

Actions are proposed for each difference between the start state and the goal state. For example, if valve-1 is open at the start and closed at the end then an action 'close valve-1' would be proposed. The set of actions is then ordered through constraint posting using a partial order plan representation. The system never backtracks and never needs to.

Difficulties arise because high level ideas are translated down to low level ideas before planning begins. Sometimes the translation process will be ambiguous. For example, there may be many different sets of valve operations that will achieve a given flow goal. It is the responsibility of the user to choose the correct interpretation for any ambiguous request, e.g. to decide the correct flow path. The decision is not backtracked by the system.

The system is also limited because in each problem each valve may change state only once. To work around this problem the system encourages the user to represent complex problems by a sequence of simpler subproblems. However, it is not clear that the work done by the user to subdivide a problem is at all trivial.

4.5.2 Alsop and Macchietto

Alsop and Macchietto (1995) describe an extension of the approach used in Lakshmanan and Stephanopoulos (1988b). The extension provides a new ordering algorithm that is able to deal with more general safety constraints. The extended work also looks at actions to operate machinery as well as to open and close valves. These actions are implicitly assumed not to have side effects that would conflict with the desired effect of earlier actions. However, the actions may have preconditions that must be satisfied before they can be operated.

Action ordering, the focus of the paper, works on a two stage cycle. All actions that have satisfied preconditions are selected in the first stage. These actions are
then simulated. The system then starts the cycle again by identifying and selecting the actions that are applicable in the new state.

This approach is similar to Fusillo and Powers (1987) without backtracking. Backtracking is not required because of the simplicity of the actions used.

### 4.5.3 Naka and others

Naka and McGreavy (1994) and Batres et al. (1995) continue the tradition of Tomita et al. (1989b) for creating complex procedures by using novel plant representation techniques.

In the work considered here, a plant is divided into sections called control group units or CGUs. Physically, a CGU represents a number of units surrounded by control valves. An example of a CGU is a tank with its surrounding valves.

CGUs are chosen to be independent of each other in the same way that actions are independent in Alsop and Macchietto (1995). Starting up one CGU may help satisfy the preconditions for starting up some other CGUs. However, the startup of a CGU will not interfere with the operation of any CGU that is already active.

Startup is modelled as the activation of each CGU in a plant. Synthesis involves finding the order in which CGUs can be started and using synthesis to find the detailed instructions for each CGU startup.

This approach, which is not fully described here, is not as trivial as it may seem and has been used in the startup of at least two real plants.

The limitation of this work is that each CGU is allowed only one objective during an operation. It is not clear that a similar approach could be used for synthesising batch operating procedures or for synthesising maintenance tasks. During these tasks, units move from their usual state to an intermediate state (i.e. where a batch is produced) and then back to their usual state. In effect, the units have two objectives, (1) to produce a batch or fix a component and (2) to return to the usual state.
4.6 Scheduling

The Scheduling paradigm is not too dissimilar to the action ordering paradigm. Most scheduling work has looked at the operation of batch plants. It is assumed that there are a fixed set of methods for creating each batch. The role of the scheduling system is to find the most cost effective method for producing a set of batches while satisfying the physical constraints on the use of the plant.

The constraints considered in scheduling usually involve the sharing of resources or the ordering of actions. For example, two batches cannot be produced in the same unit at the same time. Also, if a chemical YZ is built from chemicals Y and Z, then the production of Y and the production of Z must come before the production of YZ.

It is not clear that scheduling systems can represent the idea of an unsafe situation which is used in other systems. In general it is assumed that the production of two different batches may come into conflict only if they require the same resources, e.g. the production of a batch is assumed not to have side effects which may cause conflict. For example, in the system described in Crooks and Macchietto (1992) the processing of one batch may be delayed while a shared vessel is used by another batch however the vessels in a plant never have to be cleaned out between batches of different types.

Scheduling is a large and mature area of research. It may be possible to treat some OPS problems as scheduling problems. However, traditionally scheduling and planning are separate fields of research. Scheduling is useful when the actions needed in a procedure can be easily identified and planning is useful when it is difficult to choose the actions that will make up a valid procedure. It is doubtful that scheduling can be used to solve all practical OPS problems partly because of the difficulty involved in satisfying the safety constraints on some problems.

4.6.1 Crooks and Macchietto

Crooks and Macchietto (1992) describe the first attempt to apply scheduling techniques to the creation of a general OPS system. The procedures considered by this system require the production of a set of batches using a batch processing plant.

Synthesis takes place in two steps. The first step schedules the operation of the
The scheduling phase is powerful. It is able to schedule a number of batches in order to produce the required tonnage for a chemical. Also, when more than one type of chemical is to be produced, the system will choose the optimal way to produce the two chemicals in parallel rather than the two chemicals individually.

The detailed procedure synthesis is not as powerful. It is assumed that the unit operations required to produced different batches will not interact. For example, when the system finds flow paths to move chemicals into units it does not check that the pipes used to carry these flows are not already in use.

4.6.2 Other scheduling work

The work started in Crooks and Macchietto (1992) is extended in Rotstein et al. (1994a, 1994b, 1996).

A scheduling approach is also used in Liu and McGreavy (1995) for the operation of pipeless plants. Pipeless plants work by physically moving chemicals in their reaction vessels rather than by using pipe-work to transfer chemicals. Pipeless plants do not have the flow routing problems encountered by Crooks and Macchietto (1992).

4.7 Conclusion

This review has examined the five OPS paradigms in detail. In the process more than 35 publications have been cited.

The oldest three paradigms primarily examine the selection and ordering (planning) of actions to make up an operating procedure. The work is relatively immature and no systems from these three paradigms have been taken up by industry. The state graph paradigm has been abandoned because it is slow and hard to use. Forward chaining systems are limited in their ability to select the actions needed to solve some problems. The forward chaining paradigm also has difficulty with safety and
with valve sequencing. Action Synergy systems are limited to solving valve sequencing problems. Action Synergy systems have difficulty with valve sequencing problems that require some planning.

The remaining two paradigms look at tasks where it is apparent which actions could make up a procedure. These paradigms are interested in how to order actions and how to choose the details of actions in order to solve complex objectives. Systems in these paradigms are good at solving problems in specific areas of OPS. These areas are chosen so that the plan operator has a good understanding of the units that will be operated in the solution to a given task. The areas are also chosen to limit the way that units can interact. However, these paradigms have difficulties with safety and the need to propose new actions to resolve safety problems.
Chapter 5

The Design of CEP

CEP (Chemical Engineering Planner) is the prototype OPS system that has been developed at Loughborough over the last three years. The aim in creating CEP was to develop an OPS paradigm which could solve tasks involving planning, safety and valve sequencing.

CEP contains two procedure synthesis engines, each based on a different OPS paradigm. The first synthesis engine has the same role as a forward chaining system but does not work in the same way. This engine selects the steps which will make up a procedure and at the same time finds a valid ordering for these steps. We call this the planning engine. The second synthesis engine has the role of an action synergy system. The engine is used to solve valve sequencing problems.

The two synthesis engines in CEP are fully integrated so that the user can view the system as a single function. The planning engine automatically delegates flow goals to the valve sequencing engine. The valve sequencing engine returns a set of lower level goals involving the operation of valves and pumps. These goals are solved by the planning engine. CEP is the first OPS system that is able to solve tasks that require both planning and valve sequencing.

CEP contains a safety maintenance engine as well as the two synthesis engines. The safety maintenance engine is based around a novel algorithm described in chapter 7. The algorithm makes CEP the first OPS system to be able to maintain an arbitrary set of safety constraints in a correct and complete way.
CEP is more than just a hybrid OPS system. It is the first system to utilise current AI planning techniques. There are two reasons why current AI planning techniques should be used in OPS.

First, AI planning research is better understood than research in the forward chaining paradigm. All forward chaining systems have been shown to be either incorrect or incomplete in chapter 4. Many AI planning techniques and some AI planning systems have been shown to be correct and complete. UCPOP (Penberthy & Weld, 1992) is an example of a correct and complete planner. The CEP planning engine is reasonably similar to UCPOP. An OPS system could be based around UCPOP using the algorithms described in this thesis.

Second, AI planning research is more developed than forward chaining research. Currently more is published on AI planning in a year than has ever been published on OPS. Many techniques have been developed in the planning literature which could be used to extend current OPS systems. For example, there is a large body of work on planning with time and deadlines. Planning systems are relatively fast because of the development of general but powerful search heuristics.

In summary, CEP is an OPS system that overcomes many of the limitations of earlier work. Furthermore, CEP can be extended to solve some new classes of OPS problems by using existing technology.

5.1 The CEP architecture

CEP is based around three basic components: a planning engine, a valve sequencing engine and a safety maintenance engine. A separate safety maintenance engine is required because current planning techniques provide little support for the model of safety used in OPS.

The basic architecture of CEP (shown in figure 5.1) is based around the use of a planner to control the process of synthesis.

During synthesis each goal is solved using the planning loop shown in figure 5.2. The loop starts by the planner choosing a new goal from the agenda. The goal is either solved by the planner or is delegated to the valve sequencing engine. The
Goal
Agenda

Plan

Safety
Maintainer

[call or redo]

[success or failure]

Planner

Valve
Sequencer

[call or redo]

[goal +]

[success or failure]

Safety
Problems

Plan
Conflicts

Figure 5.1: The structure of CEP

Select a goal from the agenda

Solve with an existing action

Add a new action to the plan

Use valve sequencing

Ensure the solution to the current goal does not negate the solution to some earlier goal

Ensure that the procedure is still safe

Solve the goal

Figure 5.2: The planning loop used in CEP
solution to a goal may add to any, or all, of the data structures shown in figure 5.1. After a goal is solved, the conflict resolution algorithm is called to clear the conflict agenda. This algorithm makes sure that one action does not negate the important effects of another action. After conflict resolution, the safety maintenance algorithm is called to resolve any safety problems with the unfinished plan. The planning loop is then repeated.

The planner reports success if the goal agenda is empty at the start of any planning cycle. However, any point in the planning cycle may fail except for the goal selection point. The failure of a point causes backtracking. If all points in all planning cycles fail then the planner returns an unsuccessful code.

The valve sequencing engine and the safety maintenance engine are described in detail in chapters 6 and 7 respectively. The planning engine used in CEP employs established AI planning techniques (see chapter 2) and is not discussed in detail here. The reader is invited to consult Weld (1994) for a general introduction to planning. Section 5.3 describes the main differences between CEP and the planning algorithm described in Weld (1994).

5.2 A Comparison with earlier work

The paradigm used in the development of CEP is closest to the forward chaining paradigm. The operation of a plant is represented by a set of planning operators. The role of the system is to choose and order actions from these operators so as to achieve a given set of objectives while satisfying a given set of safety constraints.

CEP differs from earlier work because procedures in CEP are not simulated as they are created. Forward chaining systems require a forward chaining search so that the steps in a procedure can be simulated. This simulation is used to reason about safety and to order the steps in a procedure. CEP is able to reason directly about the steps in a procedure. It is able to find a valid procedure without explicit simulation.

The algorithm used by CEP to reason about a plan is based on well understood AI planning ideas. These ideas come from Sacerdoti (1985) (partial order planning), Stefik (1981) (constraint posting), and Tate (1976), Kambhampati and Nau (1996)
This section describes the advantages and disadvantages of the complex action ordering techniques used in CEP.

5.2.1 Disadvantages

The CEP paradigm has three disadvantages when compared with the forward chaining paradigm.

First, CEP is more complex than most earlier OPS systems. The current implementation of CEP is written mostly in ANSI C; the parser for the plant model is written in lex and yacc (see Levine et al., 1992). The system consists of more than thirty thousand lines of source code.

Second, CEP cannot use the traditional algorithm for evaluating the safety of a procedure. In the forward chaining paradigm, safety constraints are evaluated through the simulation of the procedure. CEP uses least commitment planning techniques to reason about more than one procedure at the same time. It is not practical in CEP to simulate the effect of the set of procedures being considered at any one time. Instead CEP contains a more efficient but more complex safety analysis algorithm (see chapter 7).

Third, the operator model used in CEP is more limited than the models used in some forward chaining systems.

Operations are modelled in CEP using a STRIPS (see Fikes & Nilsson, 1971) like modelling language. CEP operators are represented by a list of preconditions and a list of effects. The preconditions must be true before any action from the operator can be used. The effects are caused to be true by carrying out the action.

In CEP it is assumed that each operation will have a fixed set of effects. It is also assumed that only a small number of different operations will achieve any given effect.

For example, consider the operation of turning on a heater. The operation can be modelled to cause the plant to become warm. However, the operation cannot be modelled as causing the plant to reach a specific temperature depending on the
initial temperature of the plant. Further, it is not always practical to get around the problem by defining a large set of operators with one for each interesting starting temperature. This ad hoc approach can make CEP very slow.

Some forward chaining systems allow less constrained operations. For example, valve operations in Rivas and Rudd (1974) can sometimes cause a flow and sometimes not cause a flow. Operations in Fusillo and Powers (1988b) can be modelled by arbitrary functions and so may have an infinite range of possible effects. However, neither of these systems are correct and complete. The system described in Rivas and Rudd (1974) cannot select the operations needed to create an arbitrary flow and so is algorithmically incomplete. The system described in Fusillo and Powers (1988b) cannot reason about the possible side effects of an operation and so is algorithmically incorrect. It is not clear that a correct and complete forward chaining system can be created if complex action models are used.

It may be possible to extend CEP to work with numeric variables. Numeric variables provide a powerful tool for describing some operations even if these variables can only be related using simple equations. Current research is examining how CEP can be extended to use numerical variables.

5.2.2 Advantages

There are two advantages of the AI planning based action ordering algorithm used in CEP.

First, CEP can insert new actions into the middle of an existing procedure. Forward chaining systems are more limited and can only add new actions at the end of a procedure. The advantage of the CEP approach is that safety constraints can be resolved in a correct and complete way. To avoid mixing two chemicals, CEP is able to add a purge operation to the procedure and this operation can appear much earlier than the point at which the chemicals would otherwise mix. Forward chaining systems have to resolve safety constraint violations at the point just before the violation would otherwise occur. This may be too late because the pipes needed to carry the purgative may be committed to some other task at that point.
Second, CEP uses a least commitment strategy to select the ordering of the steps in a procedure and the variable bindings for each step. As a result, CEP is often able to create quite complex procedures using very little search.

5.3 Planning in CEP

CEP is designed around a purpose built AI Planner. At the start of the project no recent planners were publicly available. This has since changed with the publication of UCPOP.

This section describes the differences between CEP and the UCPOP planning algorithm as described in Weld (1994). UCPOP is chosen here simply as a common point of reference. An objective of this project was to use standard planning techniques in the creation of an OPS system.

Features not in CEP

CEP does not reason about conditional effects, disjunctive preconditions or quantification. These features may have been useful but were not needed in the problems examined during the project.

In CEP, variable constraints are applied when an operator is instantiated into an action. In UCPOP variable constraints are treated as preconditions. This difference is subtle. The difference is only important to the representational power of disjunctive preconditions.

Features only in CEP

In CEP an operator can be defined to expand a goal into a set of lower level goals. For example, an operator may be defined to rewrite a goal like 'shutdown the compressor' into three goals 'turn off the compressor', 'depressurize the compressor' and 'isolate the compressor'. In UCPOP, goals cannot be expanded in this way.

Some operators are designed to achieve a particular effect but also have side effects. For example, starting a pump is designed to add pressure to a system but
may also heat the system. Often it is undesirable to use an operator just to achieve its side effects. In CEP the useful effects of each operator, as opposed to the side effects, can be marked and the operator will only be used to achieve these useful effects.

Variables are constrained to sets of possible values. If the possible bindings for a variable are ever constrained to a single value then that variable is assigned to the value. As a result, in CEP if a variable is not false then it is true (given the set of possible values \{true, false\}). In UCPOP a variable is not associated with a set of possible values. A variable that is not false may be unbound.

CEP uses the notion of a *fragile* causal link. Fragile causal links are part of a heuristic for reducing the search space in planning. To understand the heuristic requires some understanding of how CEP solves goals. In CEP and in UCPOP, a goal can be solved by relating the goal with the effect of some action earlier in the plan. This will involve adding a causal link to the plan to remember and protect the association between the goal and the selected action. It may also involve constraining the plan either to match the goal and the effects of the action or to move the action earlier than the goal or to resolve conflict between other actions and the causal link. If no constraints are imposed, then the planner has effectively solved the goal without any work. After choosing a method for solving a goal the planner may eventually fail and backtrack. The planner should not choose an alternative method for solving the goal if the chosen method did not involve constraining the plant and if the causal link protecting the goal was never involved in conflict during planning.

In the domains considered during the development of CEP, goals are regularly solved without having to constrain the plan. In CEP, the causal links for these goals are marked as fragile. During backtracking, CEP will not look for an alternative solution to a goal if the causal link for the goal has been marked as fragile. However, if there is a conflict between a fragile causal link and some other action in the plan then the causal link may be removed and the original goal placed back on the goal agenda.

CEP also uses the 'shadowing heuristic' taken from Drabble (1994). Consider a
goal \( g \) which is required to be true at a point \( P \) in the plan. \( g \) can be solved by an effect \( q \) of some earlier action if constraints can be applied so that \( g = q \). Consider the case when both \( X \) and \( Y \) achieve the same \( q \) such that \( X \) is before \( Y \) and \( Y \) is before \( P \). Given that planning fails when the effect of \( Y \) is used to solve \( g \), then planning will fail if the effect of \( X \) is used to solve \( g \). Hence the effect of \( X \) need not be considered as a method for solving the goal.

**Goal selection**

CEP has its own set of heuristics for choosing goals from the goal agenda. Goal ordering is important for the performance of the planner.

In CEP, each goal is assigned with a priority value. The goal with the highest priority is selected at each opportunity. The priority of each goal is based on the factors given below. The weight of each factor can be changed by the user.

- CEP prefers goals that come later in planning. This means that that CEP will tend to find a complete solution for one goal before solving the next goal. This is a common heuristic.

- CEP prefers goals that are added by the algorithm in chapter 7 to protect the safety of the plan.

  The main reason for adding priority to a goal is to discover more quickly whether that goal can be solved. There are often many ways that a plan can be made safe. Simple empirical evidence suggests that adding priority to goals added to make the plan safe can reduce the amount of time wasted in backtracking.

- CEP prefers goals that have in the past caused the planner to backtrack. The idea is that if the planner starts to backtrack because \( g \) cannot be solved then the planner should not stop backtracking until a point at which \( g \) is solvable or a point at which the action requiring \( g \) has been removed from the plan.

- CEP avoids goals that contain variables. Again, this is a common heuristic.
• if a goal $g$ can be solved by relating $g$ to an effect $q$ of an action $X$ and if $q$
contains variables that appear in other goals, then CEP avoids solving the goal
$g$. The idea is that the solution to one goal should not restrict the planners
ability to solve another goal. For a time there was a bug in CEP which applied
this heuristic in reverse. Since this bug has been fixed, CEP runs ten times
faster on some problems. This shows the power of the heuristic.

• CEP can be instructed to add or remove priority from any goal matching a
given pattern.

5.4 Conclusions

CEP, developed in this project, represents a new approach to OPS. CEP overcomes
many of the limitations that are found in current forward chaining and action synergy
systems. At the same time, CEP does not suffer any new and significant limitations.
The CEP approach also provides a means for using the large corpus of relevant AI
planning research in the development of OPS systems.
Chapter 6

Valve Sequencing

Valve sequencing is the task of creating or blocking a flow of chemical by using a sequence of pump and valve operations. Valve sequencing is the oldest and possibly the most well studied form of procedure synthesis.

This section presents the approach to valve sequencing used in CEP.

6.1 Valve sequencing and AI planning

It would be impractical for CEP to solve valve sequencing problems using current OPS algorithms (see Rivas & Rudd, 1974; Strimaitis, 1987; Foulkes et al., 1988; Lakshmanan & Stephanopoulos, 1990). Current algorithms require complete knowledge about the state of a plant in order to create a flow of chemical. CEP has a least commitment representation of each plant state. It is impractical for CEP to arbitrarily choose a complete plant state for a point in a procedure so that a flow can be created. At most points in a procedure there is huge set of plans which satisfy CEP's representation of that point. If CEP must choose a particular state at a point, then through backtracking, CEP will have to examine each of the huge set of possible states. This could make synthesis very slow.

CEP can solve a few simple valve sequencing problems by modelling valve operations in the STRIPS operator language supported by CEP's planning engine (see Appendix D). However, this representation is impractical for many valve sequencing
This chapter describes a new mechanism for solving valve sequencing problems within an AI planning framework. The approach is based on the use of action synergy techniques to find the steps needed to create a flow or purge or the isolation of a unit. The action synergy system developed here requires a minimal amount of state information and so is compatible with least commitment AI planning techniques. The

Figure 6.1: A divider shown with neighbouring valves
approach also avoids many of the issues discussed in Lakshmanan and Stephanopoulos (1988a). The work reasons about creating or blocking a flow rather than reasoning about opening an arbitrary valve in an arbitrary plant state.

### 6.2 Overview of valve sequencing in CEP

The valve sequencing algorithm in CEP is responsible for handling goals related to the flow of chemical in a plant. The algorithm has two duties. First, it must find the valve operations needed to create a flow of chemical. Second, it must predict the effect of these operations. In the development of CEP, the second of these two requirements was found to be the more difficult to satisfy.

It is difficult to model an arbitrary valve operation but the valves in a plant are usually not operated arbitrarily. Instead, valves are operated because of a need to create or block a flow of chemical. CEP reasons about the effect of creating and blocking each flow rather than trying to model individual valve operations.

To simplify modelling, CEP makes use of the idea of a contained flow. A contained flow path is a flow path that is isolated from the rest of the plant by a barrier of closed valves. Figure 6.2 shows two contained flow paths that link the ports in and out of a simple plant. Note that in the first flow route, the bottom section of the plant is not affected by the flow because the flow has been contained by the valves d and e. Part of the idea of a contained flow is to leave as much of the plant as possible unaffected when a flow is created or blocked.

The idea of a contained flow was first used in the valve sequencing algorithm described in O’Shima (1978). The algorithm works by finding a path or route between the source and the sink of a flow. A contained flow is then created along this route.
"Once the desired route is found, all the valves connecting to the connector [pipes in the route] but not lying on the route should be [closed] and all the valves on the route should be opened in order to maintain the flow only through that route". Foulkes et al. (1988) complete this work with the additional rule "all valves that must be closed in the final set of conditions must be closed before the flow is started". This additional rule prevents the escape of chemical when the flow is set up.

Figure 6.3 shows an unfinished plan to create and model the contained flow in figure 6.2a. The format of this plan is similar to the format of any plan to create a contained flow. The action ‘close point’ guarantees that all the valves around the flow route are closed before any of the valves along the flow route are open. The protections on the plan ensure that these surrounding valves remain closed until the point at which the flow is used. The valves along the flow route are required to be open at the point at which the flow is required. To ensure that the valves along the flow route are open no earlier than the close point, the valves are required to be closed before that point.

In figure 6.3, the contamination of the units along the flow route by the chemical being moved is modelled as occurring immediately after the close point. However, when the procedure is actually carried out, the contamination will occur as the valves along the flow route are opened. This discrepancy is not a problem because knowledge of the contamination caused by a flow of chemical is only needed to decide whether the units along the flow route must be cleaned out (purged) before the flow is created. A purge step cannot occur between the close point and the point at which the flow
is required because the valves around the flow route are protected between these two points. As a result, contamination can be modelled as occurring any time between these two points and for simplicity the point immediately after the close action is chosen.

The basis of valve sequencing in CEP is to tailor outline plans, like the one shown in figure 6.3, to solve specific flow goals in specific unfinished plans. This tailoring is done by code within the planner rather than by some user defined operator or function.

Fikes (1977) first describes the idea of using domain specific code within a planner to translate particular high level goals into macro actions. These translators are called *procedural subplanners*. Procedural subplanning is like the process of translating high level goals into low level goals by using goal expansion (see section 2.5.1). Both ideas share the same limitations and for the same reasons. For example, a high level term like flow may only appear as a precondition and not as an effect or a protected fact.

CEP has three procedural subplanners: one to solve flow goals, another to solve purge goals and a third to isolate a component. The role of the flow subplanner is to expand a flow goal using the form of expansion shown in figure 6.3. Similar templates can be created for the expansion of purge goals and isolate goals. The three subplanners are described in section 6.4.

These three subplanners are based around a plant connectivity model to describe the flow paths between the units of a plant. The plant connectivity model is described to the system using the QUEEN qualitative modelling language (Chung, 1993). This language is presented in the next section.

### 6.3 Plant modelling

The procedural subplanners used for valve sequencing require a model of the plant's connectivity. The model is written in the QUEEN qualitative modelling language for compatibility with other tools being developed at Loughborough at the present time. The QUEEN modelling language was designed primarily for an automatic HAZOP application (see Jefferson, Chung, & Rushton, 1995). As a result, the language best
supports the generation of complex static models of a plant. CEP uses only a fraction of the representational power of the language.

A plant connectivity model in QUEEN has two parts. A model of each type of unit and a model of the connectivity between the units in the plant. CEP considers only the portion of the QUEEN models that describes the propagation of chemical flow through the plant. In this restricted view, the models can be drawn as simple directed graphs. The nodes in the graph represent the ports (identifiable locations) in a unit and the arcs represent flow paths between these ports. The direction of an arc shows the intended direction of flow between the ports. Two ports may be connected by arcs in both directions to indicate that bi-directional flow is supported between the two ports.

A unit model relates the flow of a chemical entering a unit with the chemical leaving that unit. The unit model for an open non-return valve is shown in row a of table 6.1. The model shows that any chemical reaching the ‘in’ port of the valve will be transported to the corresponding ‘out’ port. The unit model for a divider is shown
in row b of table 6.1. In a divider, any chemical entering the ‘in’ port is transported to both out-ports: ‘out1’ and ‘out2’.

The names of the ports in the unit models can be chosen arbitrarily. These unit models are stored in a frame hierarchy so that any specific plant item will inherit the appropriate model from the class of unit that the item belongs to.

A model of a plant is formed by relating the out ports of specific units to the in ports of other units. These connections are always modelled as being bi-directional. Row c of table 6.1 shows the model of a plant section containing only a valve and a divider. In the diagram, dotted lines are used for inter unit connections. They are there simply to highlight the boundaries between the units. In the plant model, each port is named by a pair formed by combining the name of the unit with the name of the port within the model for that unit.

The QUEEN modelling language provides a more complex description of plant connectivity than has been used in previous OPS systems.

Previous valve sequencing tools use connectivity models in which many of the units in the plant are not represented. For example, the connectivity model in Foulkes et al. (1988) represents just the valves in a plant and the flow routes connecting them. There are two reasons not to use a simplified connectivity model. First, information about flow direction is lost when a model is simplified. As a result, there may be a point in a plant that is not equivalent to any point in the simplified model of that plant. Second, the user may not be able to operate a system properly without understanding the simplifications that are made in the model used by that system. For these two reasons, the connectivity model used in CEP is not simplified.

6.3.1 Dynamics

A QUEEN model depicts the plant in one particular state. CEP uses this model to deduce the movement of chemical between units in the plant. However, CEP reasons about acting on the plant and these actions often change the plant model. The most common way for the plant model to be changed is that a valve is opened or closed. CEP handles these changes by assigning special meaning to some types of units. In
the current CEP implementation, only valves are handled specially but in a practical system it would be necessary to handle pumps and perhaps other units in the same way.

In the QUEEN language there are no requirements on the way that a valve is modelled. It is possible, if not common, for a number of valve models to be used on the same plant (see figure 6.4). To support planning, CEP must make some assumptions about the way a valve will be modelled and how this model can be manipulated, e.g. how the valve can be closed. The challenge has been to make these assumptions general enough so that all reasonable plant models will be supported.

CEP makes the following assumptions:

1. Every valve in the plant is a descendent of a frame called value in the unit model hierarchy.

   This assumption is used to distinguish valves from other units in the plant. The implication is that any unit which acts like a valve, i.e. can be used to enable or disable flow, must be modelled as a descendent of the 'valve' frame. Some units, for example a hose that can be plugged into the plant, must be described as a kind of 'valve', although these units would not normally be called valves. A single plant model may contain more than one model for a valve so long as all these valve models are descendents of the frame 'valve'.

2. All valve models in the unit model library describe open valves.

3. The unit model of a particular type of valve when it is closed can be formed by cutting all the links between the inlets and outlets in the unit model for the
Figure 6.5: The difference between an open and a closed valve in CEP

open version of that valve. For example, figure 6.5 shows the difference between the open and the closed models of a valve.

4. Each port in the valve model is connected to at most one unit external to the valve.

It is physically impossible for two units to be connected directly to the same port in a valve. However, a header can be placed immediately upstream of a valve and enable two units to be connected to the one port of the valve. There is a temptation when modelling a plant with this layout to omit the header from the model. This simplification is not permitted in CEP.

This assumption is used to simplify the route finding algorithm.

### 6.3.2 Flow direction

A unit or pipe can support flow in three different ways: the unit may physically support flow in only one direction; the unit may be intended for flow in a single direction but fail to prevent back-flow; or the unit may be intended for bi-directional flow. Few units physically support only uni-directional flow. Examples are non-return valves and perhaps pipes on a steep slope. Most units support bi-directional flow. However, sometimes these units have an intended flow direction. Consider the case where many vessels share a common drainage line. It may be physically possible to move material between the vessels using this drainage line but this is not the intended use of the line.

There is significant difference between uni-directional flow and intended flow direction. The difference is reverse flow. In a bi-directional unit with an intended flow
direction chemical can be washed backwards through the unit as a side effect of creating a flow. If the unit is uni-directional then reverse flow should not be considered.

The current version of the QUEEN modelling language supports only bi-directional flow and intended flow direction. The modelling languages used in most OPS systems support bi-directional flow and uni-directional flow but not intended flow direction.

6.4 The procedural subplanners

This section looks at the detailed implementation of the three procedural subplanners used in CEP. One subplanner looks at creation of a flow, another looks at the creation of a purge and the third looks at the isolation of a unit.

The role of the procedural subplanners is to translate a high level goal requesting a flow operation into a set of lower level goals and effects. The lower level goals that are generated require the operation of valves and pumps. Earlier valve sequencing systems generate procedure steps instead of generating these goals. In these earlier systems, all valves and pumps are assumed to be operated in the same way. By generating goals rather than procedure steps, CEP's valve sequencing system allows ball valves to be operated differently from butterfly valves. The user is able to define separate planning operators for opening and closing each different type of valve.

Each subplanner provides an ability to write planning operators which interact with the plant at a pipe and valve level. It is the responsibility of the user when writing the planning operators describing a domain to use the subplanners correctly. For example, it is possible to use the subplanner for purging to move chemical into a tank. However, the effect of filling the tank would be better described by the subplanner for creating flow.

This section provides an overview of the architecture of the three subplanners and then looks at each subplanner individually.
6.4.1 Overview

Each procedural subplanner is called by the planner to solve a specific type of goal. For example, the flow subplanner is only called upon to solve flow goals. A subplanner will refuse to solve a goal if it has the wrong type or if the goal is not bound sufficiently. For example, the flow subplanner will not create a flow if the source point for the flow is not known. Figure 6.6 shows the point during the planning loop where the subplanners are called.

The flow and purge subplanners are very similar because they both involve creating a flow route. These two subplanners are based around four steps:

1. The current procedure is analysed to find the valves that are necessarily open or necessarily closed during the flow. These valves are treated as being locked open or locked closed accordingly.

2. A flow route is found by searching through the plant connectivity model. Search starts from the source of the flow and works towards the sink. In the case of purge, the flow path must come within a fixed distance of the point that is to be purged.

3. A second search is made to find the units affected by the flow and the valves that must be closed to contain the flow.
4. A macro action is created based around a standard template. This macro action is added to the procedure. The goal agenda, the safety problem agenda and the conflict agenda are also updated. The subplanner then returns a success token. The new conflicts, safety problems and goals are then solved by the planner.

The subplanner to block a flow is based along similar lines. However, only one search is needed to find the valves that need to be closed and the effects of closing these valves.

Each subplanner can be asked to backtrack. In the case of the flow and purge subplanners, backtracking will have the system try an alternative flow route if one is available.

6.4.2 Flow

The flow subplanner is designed to create a flow of a particular chemical between two ports in a plant. The subplanner is invoked by using a goal of either of the two formats shown below.

\[
\text{flow}(\text{start-unit}, \text{start-port}, \text{dest-unit}, \text{dest-port}, \text{chemical}) \text{ is true}
\]

\[
\text{flow}(\text{start-unit}, \text{start-port}, \text{dest-unit}, \text{dest-port}, \text{chemical}, \text{close-upstream}, \text{close-downstream}) \text{ is true}
\]

The first format of the flow goal is to create a flow of \textit{chemical} between \textit{(start-unit, start-port)} and \textit{(dest-unit, dest-port)}. The second format is provided to work around a subtle problem in the idea of using a macro action to solve a given goal. The problem is that a flow goal expands into two actions: a point at which the flow route will be isolated from the rest of the plant and the later point at which flow along this route will be achieved. Sometimes additional conditions must be achieved at the same time at which the flow is achieved. For example, an operating procedure might require both that methane was flowing through a pipe and that the filter in the pipe was clean. These extra goals can be stated in the same preconditions list as the flow goal and will become preconditions of the action which achieves the flow.
However, occasionally conditions must be satisfied strictly before a flow is created. For example, to fill a bath the plug should be in the tub before the water is run. The goal to put the plug in the tub should be modelled as a precondition of the close point in the flow macro-action. There is currently no general mechanism in CEP to specify additional preconditions for the close point of a flow. Instead, two specific preconditions have been identified and coded into CEP, the idea of closing the ports downstream of a flow destination and the idea of closing the ports upstream of a flow source. Either or both of these two preconditions can be activated by putting the keyword `true` in the appropriate slot of the second form of the flow goal.

Both formats of the flow goal are processed in exactly the same way. A search is first made through the plant model to find a route between \((\text{start-unit, start-port})\) and \((\text{dest-unit, dest-port})\). The search algorithm adheres to the idea of intended flow direction used in the plant model. If a route is found then the subplanner searches back along the flow route to find the effects of the flow in terms of the valves that will have to be opened and closed as well as the points in the plant that will be contaminated by the chemical to be transported. This search for effects assumes that chemical can flow against the intended flow direction. Figure 6.7 shows the portion of a simple plant that is examined when looking for a flow route. Figure 6.8 shows the portion of the same plant which is examined when looking for the effects of that route.

If the second format of the flow goal is used then the isolate subplanner is called to provide the extra valves which need to be shut. This list of valves is put together with the valves which must already be closed to isolate the flow path from the rest.
Figure 6.8: Area of the plant contaminated by the flowing chemical species of the plant.

Similar search algorithms are used in all the procedural subplanners. In each case, the search is based on the idea of dividing the plant into containment zones. Formally, a containment zone is a maximally large section of a plant that does not contain a valve but can be completely isolated by closing a set of surrounding valves. The premise is that a valve is the only way the planner has to control the spread of chemical and so once a species has been introduced into a containment zone, the planner has little control of the spread of the species through the zone although it can stop the chemical spreading outside the zone. The term containment zones comes from Roach et al. (1990) but the idea is very similar to the idea of a connector used in O'Shima (1978). We use the term containment zone because the idea of a connector implies that all points in that area of plant are equivalent. In a plant model that allows uni-directional flow it may be possible to have a chemical species in one part of a containment zone but not in all parts of that zone.

Figure 6.9 gives the details of the search algorithm used for finding a flow route in the plant model. A flow chart for this algorithm is given in figure 6.10\(^1\). Essentially, the algorithm performs a depth first search through the plant structure. Earlier planners have often used breadth first rather than depth first search, so as to find the shortest route for each flow individually. However, depth first search is simpler than breadth first search and neither algorithm guarantees to find the optimal solutions to the problem of routing a number of flow paths.

The flow routing algorithm can be demonstrated with figure 6.11. In frame (a)

\(^1\)The numbers besides the step boxes in figure 6.10 correspond to the step numbers in figure 6.9.
1. Define a data-type path consisting of a stack of pairs. Each pair consists of a (unit, port) pair as one element and a set of (valve, port) pairs as the other. Call the (unit, port) element of the pair the point and call the set element the options of that point. Create a path with a single item pair that consists of the starting unit and an empty options set. Identify this path by its head item. Name the head item 'current'. Now go to step 3.

2. Choose and remove one (valve, port) pair from the set options of the current point. Create a new head item for the current path. The new item takes the selected pair as its point. Note that the name current now refers to this new item.

3. If the current point is a valve then assume that valve is open. Search downstream from the current point to find the units that would be reached by the flow if chemical was to arrive at this point. Do not search past any valve except the unit of the current point, if it is a valve. Do not search past any unit that has been seen already either in this search or in the search downstream from any other point in the current path.

4. If the target unit was found in step 3 then return the current path as a possible flow route.

5. Collect the set of valves found in step 3. Make this set the options of the current point.

6. If the current options set is not empty then go to step 2.

7. If the current path has only one item then return failure.

8. Remove the head item from the current path. Note that current now refers to the next item in the path. Go to step 6.

Figure 6.9: The flow route searching algorithm
Figure 6.10: Flow routing — a standard depth first search algorithm

Figure 6.11: The search to find a flow route
the effect of introducing a chemical from 'In' has been found and the subplanner is waiting to decide which valve to open first. In frame (b), valve-a has been chosen and the effects of opening this valve have been simulated. The target has not been found and there are no valves downstream of valve-a and so the subplanner is about to start the backtracking loop. In frame (c) the decision to choose valve-a has been unmade and valve-b has now been chosen. The effects of opening valve-b have been considered and the target unit has been found. The routing algorithm will now return the current path.

Figure 6.12 shows the template of the macro-action used to create a flow. This template is filled in based on the effects calculated for the current flow route. By filling in the template, a new macro action is created. This macro-action is inserted into the plan so that the step with no name in figure 6.12 is substituted for the action in the plan that has the current flow goal as its precondition.

Using this process of creating a macro-action based on the results of a search, the subplanner creates what amounts to a tailored solution to a flow goal. This solution is treated by the planner very much like any other goal solution in that it is evaluated by the planner for its safety and for its compatibility with other goal solutions already in the plan. In contrast to many earlier valve sequencing systems (Strimaitis, 1987; Lakshmanan & Stephanopoulos, 1990), CEP's flow subplanner does not have any knowledge about safety. Instead, the safety problem is left to the safety maintenance engine.

106
As a result of safety evaluation or for other reasons, the planner may backtrack and ask the subplanner for a new flow route. The subplanner records enough search information to make backtracking possible.

### 6.4.3 Purge

Sometimes a flow will push out the chemicals that previously contaminated the pipework along the flow route. This kind of flow is modelled in CEP by the purge subplanner.

The format for the purge goal is shown below. It is very similar to the first of the flow goal formats except that it describes three rather than two points on the flow path, the extra point is a unit which must be cleaned out.

\[
\text{purge}(\text{start-unit}, \text{start-port}, \text{purge-unit}, \text{purge-port}, \text{dest-unit}, \text{dest-port}, \text{chemical}) \text{ is true}
\]

The purge subplanner is very similar to the flow subplanner. A search is first made through the plant to find a flow route. If a flow route is found then a search is made to find the effects of this flow. These effects are used to fill in a template for a macro-action. The macro-action for a purge is shown in figure 6.13.

However, the algorithm to search the plant for a flow route is different from the algorithm used in the flow subplanner because the flow route for a purge must be chosen so that a specific unit will be cleaned by the flow. Purge occurs when one chemical physically pushes another chemical out of a unit. To allow purge, the unit
to be cleaned must be relatively close to the main flow path of the purgative. In CEP, the idea of *relatively close* is represented by the number of ports away from the main flow path that the purged unit can be. The number is represented by a variable which can be set as part of the description of a planning domain. The default value is 2. Figure 6.14 shows the area of a simple plant that will be purged by a flow from start to dest. Figure 6.15 shows the area of this same plant that is contaminated with the purgative as a result of the purge.

### 6.4.4 Isolation

One task not usually associated with valve sequencing is the task of isolating a vessel or other component. However, the CEP planner itself has no direct access to the plant model and so would have difficulty finding the valves to be closed in order to isolate any particular component. As a result, a procedural subplanner is needed to translate a goal for isolating the plant into a set of valves to be closed. There are two
goals that this subplanner solves:

\[ \text{noFlowUpstream}(\text{unit, port}) \text{ is true} \]

\[ \text{noFlowDownstream}(\text{unit, port}) \text{ is true} \]

The isolate subplanner is much simpler than either the flow subplanner or the purge subplanner. A search is performed to find the set of all valves that can be reached by travelling upstream or downstream of a point without crossing a valve. A set of goals are then written to close these valves. These goals are used directly as the expansion of the isolate goal; there is no need for a macro-action.

6.5 The Limitations of CEP’s valve sequencer

The assumptions and limitations in the current version of CEP’s valve sequencing engine are comparable to the assumptions and limitations of the tools which are the current state of the art in valve sequencing.

In this section, some of the significant limitations with CEP’s valve sequencing engine are discussed along with ideas on how these problems may be overcome.

6.5.1 Flow paths that share a common destination

A flow path in CEP is always isolated from the rest of the plant. As a result, mixtures of chemicals are not created by CEP even when creating these mixtures would seem appropriate. For example, in the plant section shown in figure 6.16, the planner cannot simply request that oxygen and acetylene should both flow to the outlet. This is because requesting a flow of oxygen results in the acetylene inlet valve being held closed by the planner. To solve this problem in CEP, one would have to request a flow of both chemicals to the header which is labelled ‘a’ in the figure, and then request a flow of the oxygen/acetylene mixture to the outlet.
What is required is to extend the flow subplanner so that it can create a flow that will co-exist with existing flow paths in some way.

It should be noted that this problem is not specific to CEP. It is present in all systems that try to isolate each flow path by closing the valves around that path (see O'Shima, 1978; Foulkes et al., 1988). One way to avoid this problem is to have the valves in the plant close when they are not held open by a flow goal rather than explicitly closing valves in order to enclose a flow (see Strimaitis, 1987). However, this solution does not seem to be workable in a backward chaining system. The other solution, proposed in the literature, is to require the user to choose the valves which will be used to separate two streams that cannot mix (see Rivas & Rudd, 1974; Lakshmanan & Stephanopoulos, 1990).

6.5.2 Protecting a flow

Section 2.5 of this thesis discussed the idea of hierarchical planning. In this discussion it was put forward that high level or abstract literals could not be protected by the planner, i.e. the planner was not able to maintain a high level literal across a period of time in the plan. The reason is that high level literals are sometimes ambiguous. For example, there is often a number of different sets of valves that can be opened to create a flow. We will call literals that are ambiguous disjunctive literals because they can be achieved by satisfying one or more interpretations of the literal.

As has been implied already, flow literals are often disjunctive. As a result, the concept of a flow cannot be protected directly by the planner. An obvious solution is to have a procedural subplanner choose one specific set of valve operations to achieve the flow goals and then to protect these valves states instead of protecting the flow condition itself. For a rather subtle reason, this solution is too simplistic.

The problem of simply protecting a particular flow route was first highlighted by Rivas and Rudd (1974). They give an example in which the safety of a plant is dependent on the existence of a flow path between a particular furnace and a particular chimney. A simplified version of this problem is shown in figure 6.17. Usually flow travels through valve-a in the figure but sometimes the waste gas must
be diverted to the reaction vessel and so valve-a must be closed. In the procedure given in Rivas and Rudd (1974), initially the waste gas flows through valve-a but during the procedure it is diverted to the vessel and later diverted back through valve-a.

If the safety constraints for the procedure are represented by protecting the original flow route then there are two problems with this example: how to allow the planner to divert the flow of waste gas to the reactor; and how to have the planner reestablish the flow through valve-a when the path to the reactor becomes blocked. These two problems are different. In the first case, the planner needs to understand that the purge breaks the current flow route and at the same time re-establishes the flow. In the second case, the planner must create a new flow route, by opening valve-a, to ensure the safety of the procedure.

At least two other problems can arise. The third problem is a combination of the first and second. Under a planning operator’s control the flow of a chemical is blocked by the creation of a new flow route which ends in the middle of the plant. The planner must extend the new flow route to reach the required outport. For example, in figure 6.17, if the planner decides to create a flow to the reaction vessel by opening valve-b and closing valve-a then the planner must also open valve-c to complete the flow path to the chimney.

The fourth problem is a loop checking issue that comes out of the second problem. In the second problem the planner selects a new route for the flow because some unit in the current route is required for other duties. If the new route is later contested then when should the planner decide to change its original choice of routes rather than rerouting the flow another time.

Rivas and Rudd (1974) and O’Shima (1978) both use the same technique for maintaining flow. In these systems, a flow is maintained at a point in the plan by
having the valve sequencer only select actions which can be shown to preserve that flow. This only addresses the first flow maintenance problem.

Strimaitis (1987) provides a better solution to the problem of protecting flow routes. The technique is to define a default flow route and to keep the valves along this route open unless some alternative flow path is active. This technique is a direct solution to the second flow protection problem. The first flow protection problem is avoided because the system models the consequence of each action in terms of its high level effects as well as its low level effects. However, Strimaitis (1987) does not consider the situation where there is no default flow route. Nor does the paper address the third flow routing problem.

Other papers in the OPS literature ignore the problem of maintaining flow.

6.5.3 Pump operation

To create a flow of chemical, it is often necessary for the chemical to be actively pumped through the pipe work. Foulkes et al. (1988) describes the first valve sequencing tool to work with pumps as well as valves. A contribution of this work is to impose constraints on the relative order in which pumps and valves should be operated.

“There may be operating constraints concerned with pumps and other items of equipment. For example, it would be normal to require suction valves to be open before a pump is started. From some pumps (e.g. reciprocating ones) we would also want to ensure that discharge valves were open whilst for others (e.g. centrifugal ones) it may be permissible to run up against a closed discharge valve—perhaps to pressurise a line.”
— Foulkes et al. (1988).

It is a trivial problem to have CEP start up pumps along flow routes and to have these pumps started after all the valves along the flow route have been opened. This is a natural extension to the search algorithms already used to create flow and purge. However, it is not a simple problem to have CEP shut off these pumps when the flow path is blocked.
One way to stop the pumps along a flow route when that flow route is blocked is to add an extra action to the macro-action used in the flow subplanner. The extra action and its relation to the subplan is shown in figure 6.18. What is interesting about this extra action is that it may have to occur after the end of the plan because the flow may not require that the flow ever stops. For example, the flow may be one of the goals of the operating procedure. There is no problem in principle in having actions occur after the end of the plan and for these actions to be edited out of the final plan. However, the full implementation of the idea requires further investigation.

The idea of valve sequencing with pump operations has not been properly examined in the literature. Only two papers have considered pumping. Foulkes et al. (1988) propose a system that can turn pumps on to create a flow but will not turn those pumps off again when the flow becomes blocked. Crooks and Macchietto (1992) propose a system that is based on the idea of transfer rather than the idea of flow. A transfer involves creating a flow route and then later blocking that route. Pumping is handled correctly but a flow is not permitted to continue after the end of a procedure.

6.5.4 Mixing constraints and trapped chemicals

One way that a flow of chemical can be unsafe is when that flow causes a dangerous mixture of chemicals to form. In CEP, mixing constraints are represented as goals of prevention (see chapter 7). The goal of prevention used in this way might state that no unit should contain both hydrogen and oxygen. In CEP, this is written as shown in figure 6.19.
restrictions { 
    unit ?unit; 
    port ?port; 
    prevent 
        contains(?unit, ?port, hydrogen) is true 
        contains(?unit, ?port, oxygen) is true 
    end 
}

Figure 6.19: The goal of prevention “hydrogen and oxygen should not appear in the same pipe”

A flow may also be unsafe if it causes a particular chemical to arrive at a particular part of a plant. For example, wet air may not be allowed in a steel pipe to avoid rusting. Similarly, an inert gas may not be allowed into the flare stack. These situations can also be represented as goals of prevention.

In the CEP model of the world, when a flow is created the chemical being transported contaminates all the pipes along the flow path. A chemical species previously trapped in the pipes along the flow path is modelled as not being disturbed by the flow. Hence if a quantity of hydrogen is trapped in a particular pipe, pipe-a, and pipe-a becomes part of a flow path for oxygen then a mixture of hydrogen and oxygen will be considered to have formed in pipe-a as a result of the flow.

The problem with the CEP model is that a trapped chemical species freed by a flow, hydrogen in the above example, is not modelled as being transported along the flow path. As a result, some safety problems cannot be detected by CEP. Consider figure 6.20. If a flow of nitrogen is created then three possibly hazardous events will occur undetected by CEP: hydrogen and oxygen will be allowed to mix; oxygen will arrive in the steel tank; any new flow from the tank will be contaminated with oxygen and with hydrogen.

In practice, these problems do not show up nearly as often as one might expect. In
the example, the first problem occurs because the flow route contains two incompatible chemicals at the same time, two chemicals mentioned in the goals of prevention in the domain. In most of the problems that we have examined so far, a system is made to contain only one dangerous chemical at a time. For example, if a gas filter contains natural gas then the natural gas is completely removed with a purge of nitrogen before oxygen is added to the system. The second and third problems occur because a neutral chemical is added to a system containing a dangerous chemical. In most of the problems we have examined so far, neutral chemicals are only added to the system to purge a specific dangerous chemical. In this case the planner knows the dangerous chemical to be removed and so can check that the destination for the purge is compatible with the chemical. Section A.3.5 shows one OPS task where CEP is able to handle all safety problems correctly.

Goal of prevention regression

In order to solve the trapped chemical problem, we probably need the idea of goal regression. Goal regression is first described in Waldinger (1977).

“Suppose $P$ is a relation [literal] and $F$ is an action of program instruction; if $P$ is true and we execute $F$, then of course we have no guarantee that $P$ will still be true. However, given $P$, it is always possible to find some relation $P'$ such that achieving $P'$ and then executing $F$ guarantees that $P$ will be true afterwards.” — Waldinger (1977)

Goal regression can be applied to regress goals of prevention over flow goals. For example, if $P$ is a goal of prevention “chem-a and chem-b should not exist in the same pipe” and $F$ is a flow macro-action then $P'$ would be “chem-a and chem-b should not both exist in pipes along the flow path created by $F$ and also should not both exist in the same pipe”. There are at least three ways that goal of prevention regression can be applied to solve the trapped chemical problems in CEP.
restrictions {
pump ?pump;
flow-5-unit ?unit1; flow-5-unit ?unit2;
flow-5-port ?port1; flow-5-port ?port2;
prevent
contains(?unit1, ?port1, hydrogen) is true
contains(?unit2, ?port2, oxygen) is true
end
}

Figure 6.21: The specialised goal of prevention “hydrogen and oxygen should not appear in the units along flow-5”

Regression as part of creating a flow

The simplest idea is to apply goal of prevention regression as part of creating a flow. Whenever a flow is created then the goals of prevention in the domain are regressed over that flow to create specialised goals of prevention that apply only at the point that flow is created. For example, the goal of prevention shown in figure 6.19 regressed over a flow macro-action ‘flow-5’ could be rewritten as shown in figure 6.21 and then made specific to the close point of flow-5. For this regression to work correctly it must be possible to place constraints between pairs of variables. For example, in figure 6.21 setting a valuable for ?unit1 constrains the possible valves for ?port1 to the ports of ?unit1 that are involved in the flow.

The problem with this idea is that it does not address the third trapped chemical problem, the problem where chemicals are transported downstream by one flow and then becomes part of a second flow. In effect, use of this simple strategy assumes that once a chemical has become part of a flow then it is dilute enough to just be ignored.

Regression as part of safety checking

The most complex approach to goal of prevention regression is to perform regression at the same time that the safety of the plan is checked. For example, consider a goal of prevention stating that hydrogen and oxygen can’t be in the same pipe. If a new action contaminates a particular pipe with oxygen then the system must satisfy itself
that hydrogen was not already in the pipe. If this pipe was part of an earlier flow path then the system will regress the goal and check that hydrogen was not at or upstream of the pipe before the flow.

The advantage of this approach is that the safety checking algorithm gains an understanding of goal of prevention violations. It understands how the components of a dangerous mixture of chemicals all came to be in a particular unit at the same time. This is important when deciding how to resolve the goal of prevention violation. For example, imagine that a quantity starts off trapped within a unit and then is washed down into a vessel and then later washed down into a particular pipe where it causes a goal of prevention violation. There are two ways of preventing the violation: the chemical can be purged from the unit where it is initially trapped so long as this purge can take place before the first flow; alternatively the chemical can be purged from the vessel between the times when the first and second flows are created.

The disadvantage of this approach is complexity. Imagine again our trapped chemical. To move the chemical to the pipe where the safety problem will occur, the chemical must be washed into a vessel and then washed downstream from that vessel into the pipe. Consider the situation where the flow out of the vessel is created before the flow that washes the troublesome chemical into the vessel. This is possible while still creating a safety problem if the first flow is maintained until after the second flow is created. In this situation, consider the regression of the question “is the chemical in the pipe?”. Our intuition is to have regression work backwards through time and so the question is first regressed over the flow that washed the chemicals into the vessel. This regression has no effect on the question because this flow does not touch the pipe. The question is then regressed over the earlier flow into the pipe to achieve “is the chemical at or upstream of the pipe before the flow into the pipe?”. At the point in time when this flow is created, the troublesome chemical is not in the vessel and so the safety problem is not detected.
A hybrid regression strategy

The final approach to goal of prevention regression is a hybrid idea. Two flows of chemical cannot use the same pipe at the same time in the current implementation of CEP. As a result, all the uses of a pipe must be strictly ordered relative to one another. In theory, when a flow is created and the conflicts from adding the flow macro action into the plan have been resolved, that is the flow macro action has been ordered, it should be possible to simulate the plan to find the location of each chemical species at each point in time.

The difficulty with this approach is the difficulty of resolving a goal of prevention violation. When a mixing constraint violation occurs in a particular pipe, the planner has no knowledge of how the chemicals involved in that violation arrived in that pipe. In this case, the solution to the goal of prevention violation must be resolved over the flow actions in the procedure.

The handling of trapped chemicals in earlier valve sequencing tools

In the procedure synthesis literature we would expect the systems to detect trapped chemical problems but not to properly correct these problems. The problems could not be properly corrected because this requires inserting actions into the middle of an existing plan, something which has been found to be difficult in forward chaining OPS (See chapter 4). What we find is that early OPS work (see Rivas & Rudd, 1974) considers trapped chemicals but later work is less cautious. For example, Lakshmanan and Stephanopoulos (1990) only considers the mixing constraint violations that are caused by flowing chemicals.

6.6 Conclusion

In this chapter, the valve sequencing tool used in CEP has been described. This tool is novel because it is able to work in collaboration with a backward chaining, least commitment planning system. Previous valve sequencing tools have required forward chaining OPS systems.
The valve sequencing tool described here is not without limitations. Effort has been made to describe in detail all the problems that have been found. It has been shown that for each limitation of CEP’s valve sequencing tool, similar limitations can be found in the current state of the art of forward chaining valve sequencing work. As a result, we might expect to see these problems in a first attempt to solve valve sequencing in a backward chaining system.

As a valve sequencing tool, CEP improves on earlier work in a number of ways. By using AI planning together with valve sequencing, CEP is able to solve high level goals like “clean the filter in the gas line” while earlier systems have required ordered sequences of lower level goals like “stop the methane flow; purge with nitrogen; . . .”\(^2\). By using the goal of prevention algorithm described in chapter 7, CEP is able to resolve goal of prevention violations in a correct and complete manner while earlier OPS systems have been restricted on the type of violations that can be resolved, by the way these violations can be corrected and by the position in the plan when corrections can be included. By considering planning and valve sequencing separately, CEP allows the user to choose details about how to operate a valve and how to clean a unit. For example, a block and bleed valve may be opened in a different way from a ball valve; a vessel may be purged with a pressure-up blow-down while a contaminated pipe may require a simpler purge. By using a separate flow routing model and flow effects model, CEP is able to reason about back-flow and so work with the idea of intended flow direction.

\(^2\)Using the goal hierarchy given in Rivas and Rudd (1974) it can be said that CEP is able to solve goals at level 0 while Foulkes et al. (1988), Strimaitis (1987) require level 1 goals and Rivas and Rudd (1974) requires level 2 goals.
Chapter 7

Goals of prevention

In real world domains, a planner must avoid unsafe situations. For example, when planning to operate a chemical plant, it is necessary to avoid mixing of certain chemicals to avoid creating explosions.

These unsafe situations can be represented by goals of prevention. Informally, a 'goal of prevention' is a logical expression that is false only in states which should be avoided. As will be discussed in section 7.1, the idea of a goal of prevention has been proposed many times in the planning literature. However, little has been said about the practical aspects of planning with goals of prevention.

This chapter considers two methods of planning with goals of prevention. In the first, the goals are handled implicitly through the modification of operators in the planning domain. This technique can be, and is often, used on existing planning systems. The second method of planning, proposed in this chapter, involves representing the goals of prevention explicitly during planning. An implementation framework for planning explicitly with goals of prevention is also described.

This chapter is divided into seven sections. The first section examines the use of goals of prevention as a representation for the undesirable states in a planning domain. The second section considers the first method of handling goals of prevention implicitly through the operators in the planning domain. The third section describes a framework for handling goals of prevention explicitly in partial order planning. Section 4 provides a comparison between the implicit and explicit strategies for working
with goals of prevention. Section 5 examines the time complexity issues of planning with goals of prevention and concludes that practical OPS problems are in general NP-Hard. Section 6 provides proofs for the theorems used in this chapter. The chapter ends with a short conclusion.

### 7.1 Background

A goal of prevention describes a set of states that must not occur during a plan. Expressed in predicate calculus, a goal of prevention has the general form shown in statement (7.1).

\[
\forall (\forall i \in S_1, \ldots, \forall m \in S_m, \text{[var constraints]}). \neg (p_1 \land p_2 \land \ldots \land p_n) \quad (7.1)
\]

In statement (7.1), \(\forall i \ldots \forall m\) represent planning variables; \(S_1 \ldots S_m\) represent sets of possible values for each of the variables; and \(p_1 \ldots p_n\) represent literals. The only variables in \(p_1 \ldots p_n\) are \(\forall i \ldots \forall m\).

For example, the goal of prevention shown in statement (7.2) represents the blocks world constraint "a block cannot have more than one other block stacked on top of it". Note that \(\forall b_1 \neq \forall b_2\) should be read as "\(\forall b_1\) does not codesignate with \(\forall b_2\)."

\[
\forall (\forall b_1, \forall b_2, \forall b_3 \in \text{Blocks}, \forall b_1 \neq \forall b_2). \neg (\text{on}(\forall b_1, \forall b_3) \land \text{on}(\forall b_2, \forall b_3)) \quad (7.2)
\]

A goal of prevention is violated if it becomes false during a plan. A plan is considered safe if every goal of prevention in the domain is not violated at every point in the plan.

and ‘negative protracted goals’ in Vere (1992) and ‘dnt-disturb [sic] goals’ in Weld and Etzioni (1994).

Most of the research mentioned so far considers the concept of goals of prevention but not the process of planning with these goals. The exception is the work by Weld and Etzioni (1994) which examines the handling of goals of prevention in a complex partial order planner which supports both disjunctive preconditions and conditional operators. The description of implicit handling of goals of prevention in section 7.2 is very similar to their work. The main difference is that it repairs the incompleteness of their rules for rewriting operators.

To illustrate planning with goals of prevention, the simple problem of transporting a fox, a goose and a cabbage across a river by boat is used here. There is only one boat and the boat can only carry one passenger at a time. The difficulty is that if the fox and the goose are left alone in the absence of the ferry man then the fox will eat the goose. Similarly, if the goose and cabbage are left alone, the goose will eat the cabbage. The goals of prevention for the domain are shown in statements (7.3) and (7.4).

\[ \forall(?a, ?b \in Banks, ?a \neq ?b). \neg(position(\text{fox}, ?a) \land position(\text{goose}, ?a) \land position(\text{boat}, ?b)) \] (7.3)

\[ \forall(?l, ?r \in Banks, ?l \neq ?r). \neg(position(\text{goose}, ?a) \land position(\text{cabbage}, ?a) \land position(\text{boat}, ?b)) \] (7.4)

The start and goal states of the problem are shown in figure 7.1. The general planning operator for moving an object from one bank to another is shown in figure 7.2. The notation used is the language for our planner CEP (Chemical Engineering Planner). The achieve and using sections of the operator describe the operator’s effects and preconditions respectively. CEP describes the world in terms of function literals (function, value pairs). Therefore, operators have a single effects list rather than having both an add list and a delete list, as in the STRIPS notation.

We define the literals position(nothing) is leftBank and position(nothing) is right-
7.2 Implicit strategy

The general FerryCargo operator given in figure 7.2 can easily cause a goal of prevention violation. For example, given the start state in figure 7.1, if an operator is applied to move the cabbage to the right bank then statement (7.3) will be violated because the fox and the goose will be left alone together.

One method of planning with a set of goals of prevention is to handle the goals implicitly by rewriting the operators in a planning domain into a set of equivalent operators which are safe with respect to the goals. Goals of prevention, when handled in this way, are often called domain constraints in the planning literature.

Most earlier work on domain constraints is concerned with analysing the constraints implicit within a set of operators. Drummond and Currie (1988, 1989) pro-
vide an algorithm for pruning the planning space based on knowledge about the
domain constraints in a domain. Kelleher and Cohn (1992) provide an algorithm for
deducing the domain constraints implicit within a set of operator. In this section we
are interested in how new domain constraints can be added to the set of operators in
a domain.

We define an operator $X$ to be *necessarily protected* from a goal of prevention $g$
if and only if for all $g'$ where $g'$ is an instantiation of $g$, and for all actions $x$ from $X$,
$g'$ will not be violated after $X$ unless $g'$ was violated before $X$.

An action $x$ can be modelled by two situations: a situation in which the precondi-
tions of $x$ are asserted and a situation in which the effects of $x$ are asserted. In this
model, the concept of ‘before $x$’ is represented by the first situation and the concept
of ‘after $x$’ is represented by the second situation. Applying this model to Chapman’s
modal truth criterion (see Chapman, 1987), the concept of a necessarily protected
action can be redefined as follows:

**Necessarily protected:** An operator $X$ is *necessarily protected* from a goal of pre-
vention $\neg(p_1 \land p_2 \land \ldots \land p_n)$ if and only if whenever $X$ achieves some term $q$
such that $q$ codesignates with some $p_i \in \{p_1 \ldots p_n\}$ then for each binding of
$\{p_1 \ldots p_n\}$ in which $p_i$ and $q$ codesignate there is some $p_c \in \{p_1 \ldots p_n\}$ such
that either (1) $X$ requires $\neg p_c$ and $p_c$ is not possibly achieved by $X$ or (2) $X$
denies $p_c$.

As an example of a necessarily protected operator, consider a planning problem
involving an international criminal and a number of investigation teams. The criminal
must never be in the same country as any investigation team. In other words not
$(\text{in(criminal, ?c) is true } \land \text{ in(?t, ?c) is true})$ for all teams ?t and all countries ?c.
Consider an operation in which the criminal moves to a new country ?p. This operator
achieves $\text{in(criminal, ?p) is true}$. This operation is safe if there are no investigation
teams already in ?p. Hence the precondition “$\text{in(?t, ?p) is false}$ for all teams ?t” will
make the operation safe. In this precondition ?t is universally quantified but ?p is not
quantified. This is because a goal of prevention will be violated if any investigation
team ?t is in the particular country ?p that the criminal enters.
We define a planning domain to be \textit{necessarily protected} if all operators in the domain are necessarily protected from all goals of prevention in the domain.

As discussed, a plan is defined as \textit{safe} if no goal of prevention is violated at any point in the plan. Assuming that actions are indivisible, a plan is safe if the initial state of a plan is safe and if the state of the world immediately after each action in the plan is safe.

If a planning domain is necessarily protected then by definition any plan in the domain is safe if the initial state of the plan does not violate any of the goals of prevention in the domain.

One way to plan with goals of prevention in a particular planning domain is to create and plan in an equivalent and protected domain. We call this strategy the implicit handling of goals of prevention. Note that this strategy protects the safety of the plan only if the initial state and goal state are both safe. It is assumed that the planner will check the safety of the initial and goal states when a new domain description is loaded.

The rules needed to rewrite a domain to form a protected domain are beyond the scope of this thesis.

As an example of the implicit representation of goals of prevention, figure 7.3 provides a protected domain that is equivalent to the operator shown in figure 7.2 combined with the goals of prevention given in statements (7.3) and (7.4).

Each operator in the modified set given in figure 7.2 is necessarily safe, i.e. the application of any of the operators in a safe state will result in a new safe state. For example, consider a safe state where the fox is on the right bank and the goose, cabbage and boat are on the left bank — note that the ferry man is always with the boat — then it is safe to apply the operator FerryCargo1 to move the goose and the boat to the right bank or to apply the operator FerryCargo2 to move the cabbage and the boat to the right bank.

The task of rewriting general operators into safe operators can be seen as a process of deciding how potential goals of prevention violations will be resolved even before a single operator has been added to the plan. It could be described as a pre-
operator FerryCargo1 {
  bank ?a; bank ?b; ?a ≠ ?b;

  achieve
  position(goose) is ?b;
  position(boat) is ?b;

  using
  position(goose) is ?a;
  position(boat) is ?a;

  end }

operator FerryCargo2 {
  bank ?a; bank ?b; ?a ≠ ?b;

  achieve
  position(fox) is ?b;
  position(boat) is ?b;

  using
  position(fox) is ?a;
  position(boat) is ?a;
  position(cabbage) is ?b;

  end }

operator FerryCargo3 {
  bank ?a; bank ?b; ?a ≠ ?b;

  achieve
  position(cabbage) is ?b;
  position(boat) is ?b;

  using
  position(cabbage) is ?a;
  position(boat) is ?a;
  position(fox) is ?b;

  end }

operator FerryCargo4 {
  bank ?a; bank ?b; ?a ≠ ?b;

  achieve
  position(nothing) is ?b;
  position(boat) is ?b;

  using
  position(nothing) is ?a;
  position(boat) is ?a;
  position(fox) is ?b;
  position(cabbage) is ?b;

  end }

operator FerryCargo5 {
  bank ?a; bank ?b;
  passenger ?cargo;

  ?cargo ≠ goose;
  ?a ≠ ?b;

  achieve
  position(?cargo) is ?b;
  position(boat) is ?b;

  using
  position(?cargo) is ?a;
  position(boat) is ?a;
  position(goose) is ?b;

  end }

Figure 7.3: Equivalent operator set using implicit goal of prevention handling.
Figure 7.4: The explicit safety maintenance algorithm and the CEP planning loop compilation process. The advantage of this approach is both its simplicity and the ability to remove redundant strategies for resolving goal of prevention violations. The disadvantage of the approach is that it only considers the plan at an operator level and so work may be done to protect an action that would never have violated a goal of prevention anyway. Also, in this handling of goals of prevention, knowledge is lost about why certain actions were added to the plan because the planner never discovers the specific goal of prevention violations these actions were added to resolve.

7.3 Explicit strategy

This section describes an explicit strategy for planning with goals of prevention. The strategy is 'explicit' because the responsibility for plan safety is taken from the planner and given to an explicit safety maintenance algorithm. We present the strategy as an interesting and powerful method for working with goals of prevention.

The explicit strategy has been implemented in the CEP OPS system. In CEP, the safety maintenance algorithm is run at the end of each planning cycle (see figure 7.4). The role of the algorithm is to prevent the plan from possibly violating a given list of goals of prevention.

It would be possible to create a safety maintenance algorithm that prevented a
restrictions {
  bank ?a; bank ?b; ?a \neq ?b;

  prevent
  position of fox is ?a;
  position of goose is ?a;
  position of boat is ?b;
  end

  prevent
  position of goose is ?a;
  position of cabbage is ?a;
  position of boat is ?b;
  end }

Figure 7.5: The representation of goals of prevention in CEP

plan from necessarily violating a list of goals of prevention. CEP's safety maintenance algorithm looks for possible rather than necessary violations. From simple analysis it seems that a safety maintenance algorithm that detected only necessary violations would have to be run each time the plan was constrained. This seems too cumbersome to be practical.

CEP can solve the fox, goose and cabbage problem using both the implicit or explicit strategies. Similar planning times are achieved with the two strategies. When the explicit strategy is used to solve the fox, goose and cabbage problem, the goals of prevention are written explicitly in the model of the domain file as shown figure 7.5.

The section is divided into two parts. The first part looks at the algorithm for evaluating whether a new action possibly violates a goal of prevention. The second part provides an algorithm to resolve any goal of prevention violations that are found.

7.3.1 Detecting a violation

This subsection describes the algorithm for monitoring the safety of the plan, in other words to detect any goal of prevention violations. The algorithm works by examining the threat posed by each new action as that action is added to the plan.

As stated earlier, a goal of prevention can be represented by a statement of the
form \(- (p_1 \land p_2 \land \ldots \land p_n)\). The goal of prevention is necessarily violated in a partial plan if each \(p_1 \ldots p_n\) is necessarily true at some point in all the completions of that plan.

Consider a simple example from the fox, goose and cabbage problem. Let \(g\) be the goal of prevention preventing the fox and goose from being left alone together. Consider the situation where the fox, goose and cabbage are on the left bank and a single action is used to ferry the cabbage to the right bank (see figure 7.6). This plan necessarily violates \(g\) because in every (one) ordering of the plan and binding of the plan variables, the fox and the goose are left alone while the cabbage is ferried.

The simple plan can be modified by adding a new action to ferry the goose (see figure 7.7). The plan now has two actions which are unordered relative to one another. The new plan does not violate \(g\) if the goose is ferried before the cabbage. However, if the cabbage is moved first then the plan will violate \(g\) because the fox and goose will be left alone while the cabbage is transported. As a result, the new plan is said to possibly violate \(g\).

The safety of a plan can be evaluated incrementally by examining each action as it is added to the plan. This strategy was first suggested by Rivas and Rudd.
(1974). This evaluation is based on the idea of an action causing a goal of prevention violation. In CEP, a new action is said to necessarily cause the violation of a goal of prevention \( \neg(p_1 \land p_2 \land \ldots \land p_n) \) if the new action necessarily achieves \( p_j \in p_1 \ldots p_n \) at some point \( s \) and the goal of prevention is violated at \( s \). This is not quite the intuitive idea of causation because, by the definition, an action can cause a violation that already existed before that action was added to the plan.

A new action is said to possibly cause a goal of prevention violation if there is some completion of the plan in which the action necessarily causes a goal of prevention violation. This definition is sufficient for our needs. If the initial state of a plan is safe and each action does not possibly cause a goal of prevention violation then the plan as a whole is safe for two reasons. Firstly, if no action causes a violation then the plan cannot contain a violation to which any action contributes. Secondly, part of the STRIPS assumption is that the world state will not change except as the result of an action. If a goal of prevention is violated by a plan then at least one of the actions must contribute to this violation.

DetectViolation() is the procedure in CEP to find whether a new action possibly causes a goal of prevention violation. The heart of this procedure is a routine to examine a point \( s \) in a partial plan and decide whether a goal of prevention is violated at that point. This routine can be implemented using a simplified planner which cannot add new actions to the plan. The simplified planner is given the task of achieving all the terms in the goal of prevention at the point \( s \). Each term is treated as an end goal. In solving these end goals, conflicts are resolved as normal. We reason that the simplified planner can achieve its end goals if and only if the goal of prevention is violated at \( s \) in some completion of the plan.

At worst the simplified planner will have to consider every completion of a partial plan in order to achieve the goals that it has. In CEP each partial plan has a finite number of completions because all planning variables are constrained to sets of possible values and because a plan can be ordered only in a finite number of ways. Hence the simplified planner has a finite search space. The search space does not contain loops, mainly because solving a goal does not create new subgoals, and so
planning is deterministic.

The algorithm used in CEP to decide if a plan possibly violates a goal of prevention is shown in figure 7.8. This algorithm implements the simplified planner idea described above. For this implementation a set of sensible values for \( s \) was needed. The possible values were limited to the set of actions achieving literals in the goal of prevention. The reasoning is that, for each completion of the plan, one of the actions achieving a literal must come later than all of the other actions achieving a literal. During this final action, all terms in the goal of prevention must become true and so this action is a candidate value for \( s \). It is not clear how to predict which literal will be achieved last and so each action achieving a literal must be tried as a possible value of \( s \).

The implementation also required a strategy for handling the variables in the goal of prevention. A goal of prevention may contain variables and these variables will be implicitly universally quantified (a goal of prevention holds for any binding of its variables). CEP represents the variables in a goal of prevention by creating a set of plan variables to associate with the goal of prevention variables. The simplified planner is allowed to constrain these new variables as normal. In effect, the simplified planner is directed to find the violation of any instance of a goal of prevention violation.

### 7.3.2 Resolving a violation

When a goal of prevention violation is found in a plan, the planner must either resolve the violation or backtrack. The plan cannot ignore the violation and produce an unsafe plan. For completeness, all non-redundant methods of resolving each violation must be considered.

A goal of prevention violation occurs if for each term in the goal of prevention there is some action which makes that term true (the achiever of the term) and there is no action which possibly denies (or clobbers) the term between the point at which it is achieved and the point \( s \) where the goal of prevention is violated.

A goal of prevention violation is said to have been resolved if the DetectViolation()
Algorithm: DetectViolation(new-action, plan)

1. $c =$ choose a condition from the set of conditions achieved by new-action.
2. $p =$ choose a goal of prevention from the domain description.
3. $p_j =$ choose a term from $p$ such that $p_j$ possibly codesignates with $c$.
4. Create a new set of plan variables to represent the variables in $p$.
5. Constrain the plan variables so that $p_j$ codesignates with $c$.
6. $p_z =$ choose some term from the goal of prevention $p$.
7. If $z \neq j$ then $s =$ choose some action in plan such that possibly $s$ achieves $p_z$ and possibly $s$ comes before new-action.
8. If $z = j$ then $s =$ new-action.
9. Constrain the plan variables so that $s$ necessarily achieves $p_z$.
10. Order new-action at or before $s$.
11. Add a causal link to protect $c$ between new-action and $s$. Resolve any conflicts with this new causal link.
12. For each term $p_i$ in $p$ such that $i \neq j$ and $i \neq z$:
   (a) $X =$ choose an action in plan which possibly achieves $p_i$ and which is possibly before $s$.
   (b) $a =$ choose an effect of $X$ which possibly codesignates with $p_i$.
   (c) Constrain the plan variables so that $a$ codesignates with $p_i$.
   (d) Order $X$ at or before $s$.
   (e) Add a causal link to protect $p_i$ between $X$ and $s$. Resolve any conflicts with this new causal link.
13. Remove all the constraints and causal links that have been added during DetectViolation(). This will restore the original plan. Return that a violation was found.

Figure 7.8: The DetectViolation() Algorithm
algorithm will not signal the same violation\(^1\) again.

There are exactly two ways that a goal of prevention violation may be resolved. First, one of the literals in the goal of prevention can be denied between the point it is achieved and \(s\). This will prevent the success of step 11 or step 12e in the DetectViolation() algorithm. Second, variables can be constrained to prevent some achiever from matching the proper term in the goal of prevention, and thus prevent the success of step 3 or step 7 or step 12b in the algorithm. No other steps in the algorithm can be prevented from succeeding without erasing part of the current plan structure.

This section on resolving a violation will be divided into three parts. The first part will use an example to explain how violations are resolved. The second part will describe the procedure for preventing a violation by 'clobbering' some term in the violation. The third part will describe the procedure for preventing a violation by constraining some of the variables in the plan.

**Example of resolving a violation**

Consider the plan shown in figure 7.9 and the goal of prevention which protects the fox and goose from being left together without the boat. The plan violates the goal of prevention at move boat if \(?x\) codesignates with the left bank.

The action move boat is important to the possible violation because it is the latest achiever in the violation. We will use the symbol \(s\) to refer to the move boat in this violation because move boat relates to the \(s\) in the DetectViolation() algorithm of figure 7.8.

\(^1\)Intuitively, two goal of prevention violations are the same if they both involve the same achievers and the same goal of prevention.
There are three ways to repair the safety of the plan shown in figure 7.9. Firstly, \( ?x \) can be constrained so that it cannot bind to the value 'leftBank'. The result of this constraint is that the fox and goose are on different banks during \( s \). The two alternative methods for resolving the violation involve adding new actions. If the precondition 'goose at \( ?y \)' is added to \( s \) (MoveBoat) then the fox and goose will not possibly be left alone at \( ?x \) during the action \( s \). The precondition 'fox at \( ?y \)' can be added to \( s \) for similar reasons.

Clobbering a goal of prevention violation

A goal of prevention violation can be clobbered by using an action to negate the effect of one of the achievers involved in the violation. This is stated more formally by the theorem 1. In the theorem read \( x \prec y \) as “situation \( x \) comes strictly before situation \( y \)”.

**Theorem 1 Clobbering Theorem:** Let \( t_h \) and \( t_i \) (\( t_h \prec t_i \)) be the achievers for \( p_h \) and \( p_i \) respectively in the goal of prevention \( \neg(p_1 \land p_2 \land \ldots \land p_n) \). If \( c \) achieves or requires \( \neg p_h \) and \( t_h \prec c \preceq t_i \) then there is no goal of prevention violation containing both \( t_h \) and \( t_i \) as achievers for \( p_h \) and \( p_i \) respectively.

The clobbering Theorem can be interpreted as “one way to resolve a goal of prevention violation is to find a clobberer \( c \) for one of the terms in the violation \( p_h \) and to position this clobberer between the achiever of the term, \( t_h \), and a later term, \( t_i \)”.

A complete violation resolution algorithm requires that all non-redundant methods for applying theorem 1 are examined. An application of the theorem is a triplet \( \langle t_h, c, t_i \rangle \).

A goal of prevention violation is resolved if a suitable clobberer is found for any term in the goal of prevention. Hence the set of possible values of \( t_h \) is the set of all achievers involved in the violation.

A value for \( t_i \) is obviously redundant if there is some achiever \( t_j \) that comes strictly after \( t_i \). Hence the only non-redundant values for \( t_i \) are the achievers which possibly come last. We will call this set of achievers \( S \). Given that \( T \) is the set of all
1. \( t_h = \) choose an achiever from \( T \).
2. \( t_i = \) choose an achiever from \( S \).
3. Order \( t_h \) at or before \( t_i \).
4. Add the goal \( \neg p_h \) to \( t_i \) or constrain \( t_i \) to ensure that \( \neg p_h \) is achieved at this point.

Figure 7.10: The algorithm to clobber a goal of prevention violation

achievers involved in the goal of prevention violation, \( S \) can defined by the necessary and sufficient conditions given in equations 7.5 and 7.6.

\[
\forall (x \in S). \forall (y \in S). (x \neq y) \tag{7.5}
\]
\[
\forall (t \in T). \exists (s \in S). (t = s \lor t < s) \tag{7.6}
\]

The values of \( c \) should be chosen so that all possible clobberers are considered. This can be done efficiently by adding a new goal to \( t_i \) to achieve \( \neg p_h \). In this way, the heuristics used in goal achievement can also be used in clobberer selection.

Hence the algorithm used in figure 7.10 can be used to find all the possible clobberers for a goal of prevention violation.

Further efficiency issues

The set \( S \) can further be refined by considering some of the ordering constraints on the achievers involved in the goal of prevention violation. The achievers have a natural order which is determined by the ordering constraints in force before DetectViolation was called. The achievers also have a set of required orders, orders which will cause a goal of prevention violation.

The natural order differs from the required order because of step 11 and step 12e. These are the steps which reorder the plan so that there is no clobberer of any goal of prevention term \( p_j \) that occurs between its point of achievement \( t_j \) and the last term \( s \).

\( S \) is defined using the natural order of the actions in the plan. In other words, \( a < b \) in the definition of \( S \) implies that \( a \) is strictly before \( b \) in the natural order of
Let $S^\circ$ and $S^\circ_0$ be versions of $S$ applied to the required order of actions in the plan. $S^\circ$ is the version of $S$ where $a < b$ implies that $a$ is before $b$ in all cases when the set of achievers violate a goal of prevention. $S^\circ_0$ is the version of $S$ where $a < b$ implies that $a$ is before $b$ in all cases when the set of achievers violates a goal of prevention given that an arbitrary set of variable constraints have been applied.

What does it mean for an achiever $t_j$ to be in $S$ but not in $S^\circ_0$? It means that if $t_j$ occurs last then some term $p_k$ will be necessarily clobbered before $t_j$ and so the violation will not occur. One way this can happen is shown in figure 7.11a. In this case $t_j$ achieves $\neg p_k$ before it achieves $p_j$ and so the precondition of $t_j$ ensures that there will be a clobberer $c$ between $t_k$ and $t_j$ as per theorem 1.

What does it mean for an achiever $t_j$ to be in $S^\circ_0$ but not in $S^\circ_0$? It means that for some, but not all, bindings of the variables in the plan, $t_j$ will act as a clobberer if ordered last.

Note that figure 7.11a and figure 7.11b are different because their preconditions and effects are reversed. Figure 7.11b can appear as a last achiever as shown in figure figure 7.11c.

However, consider the case in figure 7.11c where the achiever of $p_k$ occurs before $t_j$ in the natural order of the plan. In this case, there is only one value of $S^\circ$ which is $t_j$. In other words, $t_j$ must come last.

The concepts $S^\circ$ and $S^\circ_0$ can be used to reduce the number of possible clobberers that must be tried. For example, if a possible value of $s$ is not a member of $S^\circ_0$ then simply ordering $s$ last is sufficient to make the plan safe.
Constraining the plan to prevent a violation

This section considers how the variables in a plan can be constrained so as to resolve a possible goal of prevention violation.

If CEP decides that two literals in a plan can be unified, it provides the list of variable constraints that will cause this unification. If the negation of one or more constraints in this list is applied to the plan then unification will not be possible. For example, \( f(?z, 1) \) unifies with \( f(?x, ?y) \) given that \((?z \approx ?x)\) and \((?y \approx 1)\). Applying either \((?y \neq 1)\) or \((?z \neq ?x)\) will prevent the unification.

It is slightly more difficult to prevent the unification of a term in a goal of prevention and a literal in the plan. Consider the situation where the unification between a literal in a plan and a term in a goal of prevention requires that a goal of prevention variable \( ?a \) is unified with a plan variable \( ?x \). Applying \(?a \neq ?x\) will not prevent the unification because \( ?a \) is implicitly universally quantified (the goal of prevention holds for all bindings of \( ?a \)).

Unifying a term in a goal of prevention with a literal in a plan may induce constraints on plan variables. These induced constraints can be negated to prevent a unification. For example, if a plan variable \( ?x \) in the domain \( \{1, 2, 3\} \) is matched with a goal variable \( ?a \) in the domain \( \{1, 2\} \) then the constraint \( ?x \neq 3 \) is imposed on the plan. To prevent this match between goal of prevention and the plan, the planner can impose the constraint \( \neg(\neg(\exists ?x \neq 3) \) or \( ?x \approx 3 \).

There are three different types of variable constraints that result from an attempt to unify a goal of prevention with the plan. The first two of these are the common unification (see figure 7.12) and separation (see figure 7.13) constraints used in least commitment planners. The third, which we call a 'constriction constraint', results when unification constricts the domain of a plan variable (see figure 7.14). The negation of an induced constraint from any of these three types will prevent the goal of prevention matching the plan.

To find the constraints induced on the plan variables when a plan is matched with a goal of prevention, CEP starts by finding the constraints needed to unify each term in the goal of prevention with each corresponding achiever in the plan. Define
C to be the set of all constraints found in this way. C will contain goal of prevention variables and may also contain redundancy. Redundancy occurs because unification is transitive and so given that ?x ≈?y, the two constraints ?a ≈?x and ?a ≈?y are equivalent.

The task facing CEP is to deduce a maximal, non-redundant set of constraints which does not contain goal of prevention variables but is implied by C. CEP starts by dividing the variables in C into equivalence classes. This can be done by parsing the unification constraints within C. The separation constraints in C are then used to mark classes as being different. At the same time, the constriction constraints from C are used to constrain the possible values of an equivalence class.

All goal of prevention variables are then removed from each equivalence class. Equivalence classes that become empty during this operation are removed.

A dominant symbol is selected to represent each equivalence class. This dominant symbol is chosen so that if the possible values for a class has been constricted to one value, the dominant symbol will be that value. Otherwise the dominant symbol is any plan variable from the class. Each dominant symbol is used to represent the whole class. A minimum set of imposed constraints is then formed by expressing all relations between equivalence classes in terms of the dominant symbols.

The goal of prevention violation can be resolved by negating and applying any one of the minimal set of imposed constraints formed in this way.
7.4 A comparison of the two strategies

In this section we examine the difference between the implicit and explicit strategies for handling goals of prevention. We find that the two strategies differ very little in the way that they make the plan safe. There are only a few ways to protect the safety of a plan and for completeness, the planner must use them all. The real difference between the two strategies is how and when the plan is modified in order to improve safety.

The explicit strategy is based on a safety maintenance algorithm that is run at regular intervals during planning. The maintenance algorithm has two stages. In the first stage, the safety of the plan is analysed. In the second stage, the plan is repaired to remove any and all safety problems that are found.

The second stage in the safety maintenance algorithm is similar to the algorithm used in the implicit strategy to make an action safe. Both routines make an action safe by either adding new preconditions (Protect) or by adding constraints to the variables in the action (Constrain).

The two strategies differ in the point in planning when an action is made safe. In the implicit strategy, each action is made safe before it is added to the plan. In the explicit strategy, an action is made safe if the action possibly violates a goal of prevention. The two strategies also differ in the possible violations that an action is made safe against. In the implicit strategy, each action is made safe against all violations that could occur. In the explicit strategy, each action is made safe against the violations that possibly occur in the current plan.

In effect, the explicit strategy can be viewed as the implicit strategy plus a filter. The implicit strategy reacts to every possible threat to the safety of the plan. The explicit strategy reacts only to those violations which possibly occur in the current partial plan.

There are two good reasons for using an explicit strategy for handling goals of prevention:

- The planner is better able to explain a plan if the explicit strategy is used.

   Sometimes the planner must modify a plan to make it safe. With the explicit
strategy, the planner knows which possible violation each modification is added to protect against. The ability to explain a plan is important if a person and a computer are to cooperate to create a plan.

- The explicit strategy is flexible and can support a more complicated handling of goals of prevention. For some OPS relevant problems, goals of prevention must be constrained to a region of time and the planner must allow new goals of prevention to be added during planning. The explicit strategy provides access to the algorithm which detects a goal of prevention violation. It is possible to change this algorithm so that some goals of prevention are only noticed in certain periods of time during a plan. It is also possible to have the algorithm look for all the violations of a new goal of prevention rather than just the violations resulting from the newest action.

To illustrate the need for flexibility, consider the operation of a chemical plant. In a particular plant, a chemical $h$ needs to be continuously added to a reaction vessel during a reaction lasting ten hours. There are six routes that $h$ can flow along from its supply tank to the reaction vessel. At all times during the reaction at least one route must be open to the flow of $h$. However, the route that is used during the first five hours may be different from the route that is used used in the last five hours. This may happen if one of the pipes in the original flow route is needed in the achievement of some other objective of the planner. Protecting the flow of $h$ during the reaction is the same as maintaining a goal of prevention which guards against closing all six flow routes during the reaction. If a planner is to model a reaction as a planning operator of some kind, the planner must (1) be able to plan with goals of prevention which are limited to a region of time and (2) be able to add new goals of prevention during planning when a reaction operator is added to the plan. There is nothing special about the protection of a flow. It is just an example of the protection of a disjunctive goal over a region of time.
7.5 Time complexity

Planning is in general an NP-Hard problem (see Bylander, 1994). Hence planning with goals of prevention is also NP-Hard, quite obviously.

Planning with some restricted modelling languages is a polynomial time problem. A suitable language can be formed by taking the STRIPS language and removing the ability to represent preconditions (see Bylander, 1994).

Theorem 2 Intractability of Goals of Prevention. Planning with goals of prevention is NP-Hard even if the following conditions are imposed on the planner and/or planning language: (1) every goal of prevention must be the conjunction of no more than three literals; (2) the start and end states of any planning problem must be safe with respect to the goals of prevention in the domain; (3) all goals of prevention must not contain variables; (4) the planning operators in the domain description cannot contain preconditions; (5) all planning operators must not contain variables; (6) the planning algorithm includes the simplifying assumption that for every planning problem, each end goal will be achieved by exactly one action in the final plan and every action in the final plan will achieve at least one end goal.

This theorem is applicable to a broad range of planning and OPS systems. For example, it implies that limited OPS algorithms described in Fusillo and Powers (1987) can represent and solve NP-Hard tasks.

Current thinking suggests that OPS systems should support the use of preconditions and should also support the use of three term goals of prevention. Hence, the OPS task appears to be NP-Hard because OPS systems seem to require a modelling language that can describe NP-Hard problems.

However, the synthesis of real operating procedures for real plants might not be NP-Hard. There may be some property of a real plant that greatly simplifies the operation of these plants. Similarly, there may be some property of common OPS tasks like startup that makes these tasks easier.

In summary, theorem 2 implies that a polynomial time OPS system either holds some new insight into procedure synthesis or the system has significant limitations.
7.6 Proofs

Theorem 1 Clobbering Theorem: Let $t_h$ and $t_i$ ($t_h < t_i$) be the achievers for $p_h$ and $p_i$ respectively. If $c$ achieves or requires $\neg p_h$ and $t_h < c \leq t_i$ then there is no goal of prevention violation containing both $t_h$ and $t_i$ as achievers for $p_h$ and $p_i$ respectively.

Proof $s$, the last term in the goal of prevention violation, must come at or after $t_i$. Hence $t_h$ cannot achieve $p_h$ at $s$ because $c$ denies $p_h$ between $t_h$ and $s$.

Theorem 2 Intractability of Goals of Prevention. Planning with goals of prevention is NP-Hard even if the following conditions are imposed on the planner and/or planning language: (1) every goal of prevention must be the conjunction of no more than three literals; (2) the start and end states of any planning problem must be safe with respect to the goals of prevention in the domain; (3) all goals of prevention must not contain variables; (4) the planning operators in the domain description cannot contain preconditions; (5) all planning operators must not contain variables; (6) the planning algorithm includes the simplifying assumption that for every planning problem, each end goal will be achieved by exactly one action in the final plan and every action in the final plan will achieve at least one end goal.

Proof The proof is by direct reduction to the Conjunctive Normal Form (CNF) Satisfiability Problem with three or more literals per clause. The Satisfiability Problem involves solving an arbitrary equation in product of sums form. It is known to be NP-Complete (see Horowitz & Sahni, 1978, pg. 545).

Let $l_1 \ldots l_n$ represent a set of literals. Let $A_1 \ldots A_n$, $B_1 \ldots B_n$ and $C_1 \ldots C_n$ each represent exactly one arbitrary literal from $l_1 \ldots l_n$. The Satisfiability Problem with exactly three literals is the problem of finding a truth assignment for $l_1 \ldots l_n$ such that an arbitrary formula in the format of equation 7.7 is valid.
DeMorgan's law can be applied to equation 7.7 in order to form equation 7.8. This transformation can be achieved by a simple rewriting process which roughly maintains the length of the equation. Hence the rewriting process can be completed in at worst polynomial time.

\[(A_1 \lor B_1 \lor C_1) \land \ldots \land (A_n \lor B_n \lor C_n) \quad (7.7)\]

\[\neg(\neg A_1 \land \neg B_1 \land \neg C_1) \land \ldots \land \neg(\neg A_n \land \neg B_n \land \neg C_n) \quad (7.8)\]

Define three new literals s, e, g which are independent of all l_1 \ldots l_n. Define h_1 \ldots h_n which are also all independent of all the previously defined literals. A new formula, equation 7.9, is formulated such that if g is true then equation 7.9 is satisfied if equation 7.7 is satisfied and some appropriate values are chosen for h_1 \ldots h_n. The variables s and e don’t appear in this equation but will become important shortly.

\[\neg(g \land \neg A_1 \land \neg h_1) \land \neg(\neg B_1 \land \neg C_1 \land h_1) \land \ldots \land \neg(g \land \neg A_n \land \neg h_n) \land \neg(\neg B_n \land \neg C_n \land h_n) \quad (7.9)\]

Define a planning domain which contains the actions L_1 \ldots L_n, H_1 \ldots H_n, S and E which are defined as follows. Each L_i has the single effect l_i and no preconditions. Each H_i has the single effect h_i and no preconditions. The action S has the effect s, g and no preconditions. E has the effect e and \neg g and no preconditions. Note that the operators in the domain do not contain variables nor do they contain preconditions. In a domain where no operators have preconditions or variables, any planning problem can be solved in polynomial time (see Bylander, 1994). We will now show that allowing goals of prevention in this domain will make planning NP-Hard.

Add to the domain a set of goals of prevention of the form shown in equation 7.10. This set of goals of prevention has been chosen to be safe if an only if equation 7.9 is satisfied. Note that the translation process from equation 7.8 to equation 7.9 to equation 7.10 requires only polynomial time. Note also that the goals of prevention
Define a planning problem containing an initial state in which \( g, s, e \) and all \( l_1 \ldots l_n \) and all \( h_1 \ldots h_n \) are false. In the final state of this planning problem, \( s, e \) and all \( l_1 \ldots l_n \) and all \( h_1 \ldots h_n \) are true, and \( g \) is false. Note that the above planning domain contains sufficient actions to solve this problem.

A planning problem may be considered unsolvable if the initial or final states of the plan violate any goals of prevention. Note that the initial and final states of the above plan do not violate any goals of prevention because \( g \) is false in all these states.

All correct solutions to this planning problem must contain a point in which \( g \) is true because \( S \) is the only action to achieve \( s \) for the end state, and \( S \) also asserts \( g \). Hence, to avoid violating any goals of prevention, equation 7.7 must be valid in the truth assignments given by the states immediately before \( S \) in all correct solutions to this plan (see figure 7.15). It is an NP-Complete task to find a truth assignment in which a formula of the form shown in equation 7.7 is valid but it is a polynomial time task to translate equation 7.7 to equation 7.10 and to specify the problem domain. Hence finding a correct solution to this planning problem must be NP-Hard (at least as hard as all NP-Complete tasks).
7.7 Conclusion

In this chapter we have discussed the concept of a *safe plan* and provided the idea of a *goal of prevention* to represent plan safety. As discussed, the concept of a goal of prevention is a formalisation of many earlier concepts that have appeared in the planning literature.

We have described two methods for planning with goals of prevention. The first involves the representation of goals of prevention implicitly in the operators of a planner. Similar techniques have been used in the past to represent domains like the blocks world. We formalise the process of modifying a planning domain to make it safe.

As an alternative we show how goals of prevention can be represented explicitly during the planning process. An algorithm for working explicitly with goals of prevention is described. This algorithm is made up of two parts: a method for detecting goal of prevention violations and a method for resolving these violations.

Our implementation of the explicit strategy works well enough to solve quite impressive OPS problems. However, we feel that the implementation could be improved significantly. Currently we resolve a goal of prevention violation by trying each possible resolution method in turn. It would be better to make use of disjunctive preconditions to represent all or most of the resolution strategies at the same time. Benchmarking the explicit strategy is not sensible until this and other changes have been made.

Future work should look at three unresolved problems: how to plan with goals of prevention in domains that contain quantified variables; how to allow new goals of prevention to be added during planning; and how to allow goals of prevention to apply only to a short period of time within a large plan. It seems likely that these three problems can be solved by extending the explicit handling strategy described here. The solution to these problems will help to improve the modelling of flows of chemicals during OPS.
Chapter 8

Case Study: The Blender Plant

Implementation of CEP is described in chapters 5, 6 and 7. This chapter completes the description of CEP by showing how the system is used to solve OPS tasks.

This chapter has the following objectives:

- to show the information required by CEP in order to solve an OPS task.
- to show how the valve sequencing and safety checking algorithms are used in procedure synthesis.
- to demonstrate that CEP can solve tasks which would require significant computation time or user assistance using earlier OPS tools.
- to demonstrate that one plant model can be used for a number of OPS tasks.

In this chapter, four progressively more difficult tasks will be solved using the same model.

This chapter provides an overview of the use of CEP. For a more detailed description of the use of CEP please consult one of the following five appendices. Appendix A provides a manual for the CEP language. Appendix B provides a user manual for CEP. Appendices C to E together provide three additional examples of procedure synthesis using CEP.

The rest of this chapter is broken down into sections as follows. The first section describes the blender plant and the four procedures to be generated for this plant.
The second section describes the CEP model for the blender plant. The third section describes the synthesis of the four procedures. The chapter then ends with a short conclusion.

8.1 The four synthesis tasks

The blender plant used in this chapter is taken from Foulkes et al. (1988). It was chosen because it is relatively complex.

The plant is shown in figure 8.1. The units $a_1$ to $a_5$ represent storage tanks. The units $b_1$ to $b_3$ are blenders. The operation of the plant involves producing batches of chemical by using the blenders to mix different chemicals.

This chapter examines the creation of four different operating procedures for the blender plant. The procedures are considered in increasing order of difficulty. The first procedure involves creating a simple flow of oxygen from a storage tank to a blender. The second procedure requires generating an equivalent flow but with some units in the plant taken out of service. These first two procedures demonstrate CEP’s ability for valve sequencing. The third procedure requires creating an equivalent flow
with one of the units along the flow route contaminated with natural gas. Natural gas and oxygen are not allowed to mix. The task demonstrates CEP's ability to use purge operations when required. The fourth procedure involves temporarily suspending an existing flow of one chemical so that another blender can be filled. This problem demonstrates CEP's ability to solve tasks where the start and end states are very similar.

The four synthesis tasks used in this chapter do not come from Foulkes et al. (1988). CEP is more capable than this earlier system and so more complex problems are used to demonstrate CEP.

### 8.2 Modelling a synthesis task

Five basic types of information are required by CEP to solve an OPS problem. CEP requires a ‘Plant Model’ which describes the units within the plant and their connectivity. CEP also requires a ‘Plant Action Model’ which describes the possible actions (steps) within an operating procedure. CEP requires information about ‘Safety Considerations’ to prevent the creation of unsafe operating procedures. CEP also requires ‘State Information’ to describe the synthesis problem in terms of the current state of the plant and the intended state after a procedure has been carried out. Finally, CEP can be given ‘Heuristic Information’ to help guide synthesis and so reduce planning time. Heuristic Information is strictly optional.

The five basic types of information, together with the subclasses for each type, are shown in figure 8.2.

The information required by CEP to create a procedure can be examined in terms of re-usability as well as being examined in terms of representation. Some information required by CEP is task specific. An example is the state information describing the task to be solved. Some information is task independent but plant specific. A good example in this case is the connectivity model for the units in the plant. Finally, some information is plant independent. An example is the action models for a generic unit like a valve.

A list of the information required by CEP ordered in terms of reusability is pro-
<table>
<thead>
<tr>
<th>Classification</th>
<th>Information Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Model</td>
<td>Unit Model</td>
</tr>
<tr>
<td></td>
<td>Connectivity Model</td>
</tr>
<tr>
<td></td>
<td>Declaration of Concepts</td>
</tr>
<tr>
<td></td>
<td>Declaration of Chemicals</td>
</tr>
<tr>
<td>Action Model</td>
<td>Operators</td>
</tr>
<tr>
<td>State Information</td>
<td>Goal State</td>
</tr>
<tr>
<td></td>
<td>Start State</td>
</tr>
<tr>
<td></td>
<td>Default State</td>
</tr>
<tr>
<td>Safety Considerations</td>
<td>Don't Use Constraints</td>
</tr>
<tr>
<td></td>
<td>Production Constraints</td>
</tr>
<tr>
<td></td>
<td>Global Safety Constraints</td>
</tr>
<tr>
<td>Heuristic Knowledge</td>
<td>Task Specific Heuristics</td>
</tr>
<tr>
<td></td>
<td>Goal Priority Information</td>
</tr>
</tbody>
</table>

Figure 8.2: CEP input classified by the representation used within CEP

<table>
<thead>
<tr>
<th>Classification</th>
<th>Information Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plant Independent</td>
<td>Operators</td>
</tr>
<tr>
<td></td>
<td>Unit Model</td>
</tr>
<tr>
<td></td>
<td>Declaration of Concepts</td>
</tr>
<tr>
<td></td>
<td>Declaration of Chemicals</td>
</tr>
<tr>
<td></td>
<td>Goal Priority Information</td>
</tr>
<tr>
<td></td>
<td>Global Safety Constraints</td>
</tr>
<tr>
<td>Task Independent</td>
<td>Connectivity Model</td>
</tr>
<tr>
<td></td>
<td>Production Constraints</td>
</tr>
<tr>
<td>Task Specific</td>
<td>Goal State</td>
</tr>
<tr>
<td></td>
<td>Start State</td>
</tr>
<tr>
<td></td>
<td>Don't Use Constraints</td>
</tr>
<tr>
<td></td>
<td>Task Specific Heuristics</td>
</tr>
</tbody>
</table>

Figure 8.3: CEP input classified by re-usability
The task of specifying a problem to CEP is simplified by reusing information. Ideally, as much plant independent information as possible should be provided in a general library. Similarly, as much task independent information as possible should be captured during design or from design documents. CEP facilitates knowledge reuse by separating out different types of knowledge.

This idea of knowledge reuse is not new. The only OPS paradigm that does not allow information to be broken down for reuse is the state graph paradigm. The state graph paradigm is the exception to a lot of rules. Ideally we would like to say that CEP can solve problems that no previous OPS system can solve. However, this is not entirely true because in theory, the state graph approach to OPS can be used to generate any procedure which can be modelled using discrete states; including the four procedures considered in this chapter. However, the four procedures cannot be generated using the same state graph (plant representation). The state graph representation for the plant during the first procedure is significantly different to the representation used in the second procedure because three units are taken out of service.

8.2.1 The static model

The static model consists of four components: the unit model, the connectivity model, the declaration of chemicals and the declaration of concepts.

The connectivity model describes the individual units which make up the plant and the flow paths between these units. For example, in the blender plant, valve v06 is defined as a ball-valve that is upstream of pump p1. The CEP coding of this definition is shown in figure 8.4. The connectivity model is intended to be plant specific but not problem specific. Most connectivity information is contained within

```prolog
instance(v06 isa ball-valve,
[ outports info [out is [p1, in]] ]).
```

Figure 8.4: The model of valve v06 — plant modelling information provided in figure 8.3
The unit model provides a plant independent description of each different type of unit found in a plant. The model of a unit can be used to store any plant independent information about that unit. However, it is common to include a description of the flow paths between the ports in the unit. For example, the unit model for a generic valve defines that, when the valve is open, any chemical flowing into the valve will flow out of the valve. The CEP representation of a generic valve is given in figure 8.5.

The declaration of chemicals provides a definition of each different type of chemical that may appear within a plant. Chemicals are often grouped into subclasses with names like 'toxic' or 'flammable'. These classifications are used to simplify the declaration of safety constraints later in the plant model. The chemical model for the blender plant has no subclasses and is shown in figure 8.6.

The declaration of concepts is used to define abstract ideas like the aperture of a valve. These ideas are defined qualitatively as a class of values. For example, the concept 'aperture' is defined as a class containing two values, open and closed. This definition is shown in figure 8.7.

The static model as a whole is the largest section in the CEP plant model for the blender plant. Of the 400 lines of information that make up the plant model, 320 lines are dedicated to static modelling. The connectivity model is the largest section.
frame(aperture).
instance(open isa aperture).
instance(closed isa aperture).

Figure 8.7: The class aperture in the declaration of concepts

operator open {
  ball-valve ?v;
  achieve
    * aperture of ?v is open;
  end
  print 'Open valve ' ?v; }

Figure 8.8: The model of opening a ball-valve

of the static model for the blender plant. The connectivity model makes up about 80% of the static model and so about 65% of the plant model.

The static model is also one of the simplest parts of the plant model to capture. An AutoCAD based tool has been developed at Loughborough to capture a connectivity model during design. The unit models and concept models are plant independent and can be taken from a library of information.

8.2.2 The action model

The action model describes the possible actions (steps) that can be performed during an operating procedure.

Actions are defined using a STRIPS operator representation. A simple operator to open a valve is shown in figure 8.8. The symbol ?v in the figure represents the variable to take the value of the value which will be opened. The operator defines that the effect of opening ?v is the logical statement ‘aperture of ?v is open’. There are no preconditions that must be satisfied before ?v can be opened. The operator is associated with the instruction ‘Open valve ?v’ in the text of an operating procedure.

In the description of the blender plant, there are about 70 lines describing the six operators in the domain. One operator to open a valve, one to close a valve, one to start a pump, one to stop a pump, one to describe how to fill a tank and one
restrictions {
    unit ?u; port ?p;
    prevent
    contains(?u, ?p, oxygen) is true;
    contains(?u, ?p, natural-gas) is true;
    end }

Figure 8.9: A goal of prevention

to describe when to perform a purge. The operator set used here is similar to the operators used in the bottleneck example given in section A.3.5 of the appendix on the CEP Programming Language.

8.2.3 Safety constraints

CEP uses information about safety to prevent the creation of unsafe operating procedures. Safety considerations are described using goals of prevention representation from chapter 7.

There are three different types of safety considerations for each of the three different levels of reusability. We call these three types of safety considerations 'global safety constraints', 'production constraints' and 'don’t use constraints'. Global safety constraints are plant independent, e.g. 'natural gas and oxygen should not be mixed'. Don't use constraints are task specific, e.g. 'pump p1 has been taken off line for maintenance and so can't be used'. Production constraints are task independent but plant specific. There are no production constraints for the blender plant but an example of a production constraint is 'the chemicals used to make this particular drug should not be contaminated at all'.

Figure 8.9 shows the goal of prevention used to represent that oxygen should not mix with natural gas. The variables in the statement are universally quantified so that (?u, ?p) represents any one point in the plant. The statements between prevent and end cannot all occur together and so the goal of prevention literally states 'in no situation during the plan should a unit contain both natural gas and oxygen'.

The goal of prevention in figure 8.9 is the only global safety constraint used in the model of the blender plant.
8.2.4 State information

A planning problem is defined in terms of two plant states. The initial state describes the plant at the time that the operating procedure will be carried out. The goal state lists particular statements which must be made true by carrying out the procedure.

In CEP, a state of a plant is defined by a set of function literals. The start state is assumed by CEP to be a complete state description, assigning a value to each predicate describing the plant. The goal defines conditions that should be true at the end of the procedure and so is usually only a partial state description.

To reduce the amount of information in the initial state, CEP allows the declaration of a default state. The default state is a plant specific description of the 'normal state' of each plant item. The start state describes how the plant differs from this default state.

In the default state for the blender model, all valves are closed and all pumps are off and all pipes are completely empty. The statement takes up about 25 lines. Each of the four planning problems in section 8.3 will describe their own start and goal states.

8.2.5 Heuristic information

Information can also be provided to help CEP solve a particular problem. This information is strictly optional but may reduce the time required to generate a procedure.

One type of heuristic knowledge supported by CEP is goal priority information. Some objectives are easier to achieve than others. For example, it is easier to open a valve than it is to create a flow of chemical because creating a flow will require the planner to open many valves. Planning speed can be improved by having the planner achieve difficult objectives before easy ones. The difficulty of a goal is often plant independent and so priority information can be stored with the action model.

CEP also supports the use of task specific heuristics. Currently, if the user can approximate the number of actions in a procedure then this may further reduce planning time.

The CEP model of the blender problem contains 5 lines of priority information.
goals {
  require
    flow(a1, out, b1, in1, oxygen) is true;
  end }

Figure 8.10: Goal state

but no estimate is made on the number of actions in the procedure.

8.3 Procedure synthesis

This chapter examines the creation of four different operating procedures for the blender plant. When taken in order, each of the four procedure is more difficult to create than the one before.

8.3.1 Creating a flow

The first problem is simply to create a flow of chemical from the default state of the plant. The problem is very easy to specify. It requires only a single goal as shown in figure 8.10. The goal requires a flow of oxygen between the outlet of vessel a1 and the inlet of blender b1.

When given this problem, CEP produces a solution in 0.03 seconds on a SPARC-station IPX. The resulting operating procedure involves opening valves 1, 6, 7, 38, 8 and starting pump p1 as shown in figure 8.11. According to CEP, the five steps can be carried out in any order.

8.3.2 Removing pumps from service

The second problem involves creating the same flow but with pumps p1, p3 and p4 taken out of service. Goals of prevention are added to prevent these pumps from being used to create a flow of chemical (see figure 8.12). Literally, the goals of prevention state that the three pumps should not contain anything.

This problem shows that CEP can cope with changes in the plant. CEP solves this modified problem in 0.25 seconds on the same machine. The resulting operating
Figure 8.11: The flow path created in solving the first problem

restrictions {
  chemical ?chem;
  prevent contains(p1, in, ?chem) is true; end
  prevent contains(p3, in, ?chem) is true; end
  prevent contains(p4, in, ?chem) is true; end
}

Figure 8.12: Pumps 1, 2 and 4 are now off line
8.3.3 Purging a chemical

In the last problem, the planner had to choose a new flow route because a pump was taken out of service thus making the original flow route unsafe. In this example, a flow route is found to be unsafe but the planner chooses to remove the hazard rather than re-routing the flow. No previous OPS system has supported both automatic purge and automatic flow re-routing although both of these ideas are required in a complete OPS system. In the development of CEP, a lot of effort has been put into developing a complete way of handling goals of prevention, including safety constraints relating to flow.

In this problem the pipe work between valves 13, 14 and 16 is contaminated with natural gas. Valve 14 is on the flow route that the planner would otherwise use (see figure 8.13).

The procedure that CEP generates involves purging the pipe work around valve 14 and then continuing as before (see figure 8.14). The procedure is generated in
### Actions

- Open v02 v11 v12 v13 v16 v27 v28 Start p2
- Stop p2 Close v02 v11 v12 v13 v16
- Open v01 v10 v11 v12 v13 v14 v15 Start p2

### Related Goal

- Purge contaminated area with nitrogen
- Stop flow of nitrogen
- Flow oxygen to blender b1

**Figure 8.14:** An annotated version of CEP's procedure for the third blender task

Approximately 1.5 second on a SPARCstation IPX. There are four points that should be made about the procedure that CEP generates.

- Valves 27 and 28 were not closed after the purge with nitrogen. It was not necessary for CEP to close these valves in order to achieve its objective. In future work we should develop a mechanism to leave plant items in their normal running state where this is appropriate.

- Blender b3 was chosen as the destination for the purge because it is defined as compatible with all the chemicals in the plant. In the default start state, b1 and b2 were each defined to be incompatible with some chemicals. This is not obvious from the description of the blender plant so far.

- Valves 11, 12, 13 etc. were closed and then opened straight away during the procedure. This happens because of the way that flow routes are chosen. In future work, a filter will be written to remove these redundant actions.

- In the CEP model of the blender plant, a pump is treated as a way of creating or blocking a flow and is described as a kind of valve. The valve and pump operations on each line of figure 8.14 are unordered relative to one another as a result. A discussion on the use of pump operations in OPS is given in Section 6.5.3.

### 8.3.4 The final problem

The final problem is based on the same set of goals of prevention but had a new start and goal state.

The problem is interesting because the start and goal states are almost identical. The only change is that blender b1 has been filled with oxygen after the procedure.
start {
aperture of v03 is open; aperture of v20 is open;
aperture of v12 is open; aperture of v13 is open;
aperture of v16 is open; aperture of v27 is open;
aperture of v37 is open; aperture of p2 is open;
contains(v03, in, natural-gas) is true;
contains(v20, in, natural-gas) is true;
contains(v12, in, natural-gas) is true;
contains(v13, in, natural-gas) is true;
contains(v16, in, natural-gas) is true;
contains(v27, in, natural-gas) is true;
contains(v37, in, natural-gas) is true;
contains(p2, in, natural-gas) is true;
contains(b2, in2, natural-gas) is true; }
goals {
require
flow(a3, out, b2, inl, natural-gas) is true;
contains(b1, in1, oxygen) is true;
end }

Figure 8.15: Start and end state for the final problem

Earlier OPS systems find this problem difficult because the difference in the start and
goals states tells very little about the procedure that is required. To solve this problem
correctly requires an integration of powerful solutions to the issues of planning, valve
sequencing and safety checking.

The start and goal states for the final problem are defined as shown in figure 8.15. In
the start state, natural-gas is flowing between a3 and b2 (see figure 8.16). In
the goal state this flow should continue, although possibly along a different path.
However, blender b1 should also contain oxygen and this condition is not true at the
start.

CEP's solution is shown in figure 8.17. CEP requires about 3.5 seconds to generate
this procedure.

8.4 Conclusions

CEP represents a very powerful approach to OPS. This chapter has demonstrated
four important features of CEP.
Figure 8.16: The start state for the final problem

Actions

Stop p2  Close v03 v20 v12 v13 v16 v27 v37
Open v02 v11 v12 v13 v16 v27 v28  Start p2

Stop p2  Close v02 v11 v12 v13 v16 v27 v28
Open v01 v10 v11 v12 v13 v14 v15  Start p2

Stop p2  Close v10 v11 v12 v13 v14
Open v02 v11 v12 v13 v16 v27 v28  Start p2

Stop p2  Close v11 v12 v13 v16 v27 v28
Open v03 v20 v12 v13 v16 v27 v37  Start p2

Related Goal

Stop flow of natural gas to b2
Purge natural gas with nitrogen
Stop flow of nitrogen
Flow oxygen to b1
Stop flow of oxygen
Purge oxygen with nitrogen
Stop flow of nitrogen
Restart natural gas flow to b2

Figure 8.17: An annotated version of CEP's procedure for the fourth blender task
• CEP is able to generate procedures for a non-trivial plant within seconds or fractions of a second.

• Unlike many earlier systems, CEP does not require ‘hints’ to solve synthesis problems. For example, intermediate states or planning islands were not used in the specification of the problems given here although planning islands are required by most earlier valve sequencing tools when solving complex problems.

• CEP supports the reuse of knowledge. All the four problems shown here were solved using the same model of the blender plant.

• CEP is able to solve a set of tasks that could not be solved by earlier OPS systems unless extra guidance was provided. For example, the last task demonstrated in this chapter required two purge operations. Most previous OPS systems would have needed some form of stationary state to be declared between the two purge operations in order to solve this problem.

In the current implementation of CEP, minimal work has been done on user interface design because the emphasis of the project has been on functionality. CEP requires more than 400 lines of information in order to solve the simplest blender plant problem and these 400 lines must be entered by hand into a single large text file. Throughout this chapter we have pointed out large sections of this information can either be stored in a library or captured during design. From this analysis, it is clear that the CEP framework supports the creation of an OPS tool which will require very little information to create each procedure.
Chapter 9

Conclusions and Future Work

This thesis examines three aspects of Operating Procedure Synthesis (OPS). We call these three aspects 'safety', 'planning' and 'valve sequencing'. The three aspects are fundamental to OPS. Any general OPS system will have to contain safety, planning and valve sequencing techniques.

No previous system has been able to address the three aspects of OPS together and in a correct and complete way. Through an in depth literature review we examined why this is the case. We concluded that the limitations of earlier work are fundamental to the approaches taken by earlier systems.

To overcome the limitations of earlier work, we have proposed a new approach to procedure synthesis. The approach is based on the use of AI planning technology to solve the planning problem and the use of novel algorithms to solve the safety and valve sequencing problems.

We have created an OPS system based on the ideas developed in this thesis. Using this system, called the Chemical Engineering Planner (CEP), we have demonstrated that our approach is capable of solving problems that no previous OPS system could solve.

9.1 Contribution

This thesis contributes to OPS research in four ways.
First is the idea that safety, planning and valve sequencing are distinct aspects of OPS. Previous work considers OPS to be an indivisible entity. The philosophy is that a general approach to OPS should be able to solve general problems given the right kind of modelling. CEP is the first OPS system to provide separate but integrated solutions to the planning, safety and valve sequencing problems. This separation is interesting because planning systems and valve sequencing systems work differently but have the same model of safety. This separation is also interesting when examining the OPS literature. No previous review has looked for the planning limitations in earlier approaches to OPS.

Second is the demonstration that current AI planning techniques can be used to solve OPS problems. By using AI planning techniques, CEP benefits in three ways. Firstly CEP can solve planning problems that were beyond most earlier OPS systems. Such a problem was described in chapter 8. Secondly, CEP can make use of the large body of AI planning theory. The idea of the STRIPS assumption (Georgeff, 1987) and the modal truth criterion (Chapman, 1987; Kambhampati & Nau, 1996) have directly influenced the way that both goals of prevention and valve sequencing tasks are handled within CEP. Finally, CEP can make use of the large body of AI planning experience. Ideas used to improve the performance of the CEP planning algorithm have come from the experience of other planning groups.

Third is the valve sequencing algorithm used in CEP. Earlier valve sequencing work is based on the assumption that a forward chaining search will be used in the synthesis of a procedure. These earlier tools require complete state information to predict the effect of creating a flow. The algorithm used in CEP is designed to work in a backward chaining environment where complete state information is not available.

Fourth is the algorithm for evaluating and correcting the safety of a procedure given in chapter 7. Previous OPS systems monitor the safety of a procedure using simulation. The use of simulation is not practical when least commitment AI planning techniques are used, as in CEP. Simulation also does not reveal how a procedure can be made safe. The algorithm described in this thesis overcomes these limitations.
9.2 Future work on CEP

This section looks at the scope for future work on the development of CEP. The section is restricted to the consideration of safety, planning and valve sequencing issues. The next section examines the aspects of OPS that have not yet been considered in the development of CEP.

The limitations of the current version of CEP system have already been discussed. Chapter 6 presented four problems with the valve sequencing engine used in CEP. These problems also appear in most earlier OPS systems. The problems involved (1) protecting a flow, (2) passing two flows down the same pipe, (3) operating pumps safely and (4) the mixing of trapped chemicals. These four problems and their possible solutions are discussed in detail in the chapter.

Chapter 7 examines the safety maintenance engine in CEP. The chapter concludes that there are three unresolved problems in this area: how to plan with goals of prevention in domains that contain quantified operators, how to allow new goals of prevention to be added during planning and how to allow goals of prevention to be applied only to a period of time within a long plan. It seems likely that these three problems can be solved by extending the explicit handling strategy described in the chapter. The solution to these problems will help to improve the modelling of flows of chemicals during OPS.

There are many ideas in the OPS literature that have not been used in the development of CEP. Some of these ideas can be viewed as separate aspects of OPS and are discussed in the next section. For example, the creation of optimal procedures. Three ideas of interest are discussed here.

Some earlier OPS systems can be told about sequences of steps that will produce specific effects. We could call these sequences of steps recipes. The operation of a unit (node) in Tomita et al. (1989b) is associated with a recipe providing the steps needed to prepare for the operation of that unit. The method for producing a batch in Crooks and Macchietto (1992) is also a recipe even though each batch may be produced in more than one way. CEP can currently represent most recipes by sets of STRIPS operators. The modelling of some problems would be improved if CEP
could represent recipes explicitly. Recipes could be implemented using the idea of macro actions. The difficulty comes in protecting disjunctive conditions between the steps in a recipe. This difficulty, in the context of protecting a flow, was discussed in section 6.5.2.

The OPS system described in Foulkes et al. (1988) allows the declaration of normal flow routes to represent the paths along which chemicals should normally flow. The system is not bound to using the normal flow route and will use an alternative route if necessary. The idea of normal flow routes can be seen as an attempt to capture the designer’s intention of how the plant should be operated. This kind of knowledge in CEP may help in the creation of procedures which are acceptable to the plant operator.

The model of safety used in CEP is based on at least the following two premises: (1) that actions are always successful, e.g. a valve will always close fully, and (2) that the operator will carry out actions correctly, e.g. the wrong valve will not be closed. Rivas and Rudd (1974) suggest that OPS systems should develop procedures that are tolerant to human error and plant failure. Related work in Mansell et al. (1995) considers procedures that have steps that are likely to fail. The planner is designed to develop contingency plans to cope with this possible failure. CEP could be made to plan for contingencies.

9.3 Future work in OPS

This thesis has considered three aspects of OPS and presented a new approach to procedure synthesis which allows these three aspects to be addressed in an algorithmically correct and complete way.

Future work should look at incorporating other aspects of OPS into the framework developed here. Four interesting aspects of OPS are listed below:

numeric modelling. Numeric modelling is necessary in the solution to some OPS problems. An example is presented in Aelion and Powers (1991) where a procedure is unsafe because the concentration of two chemicals cannot be controlled
accurately enough by a particular valve. In another example considered during
the project, a vessel had to undergo more than one pressure up blow down cycle
to reduce a specific chemical to within an acceptable region of concentration.
These and similar problems require some form of numeric modelling. The first
attempt to use numeric modelling in OPS is described in Fusillo and Powers
(1988b). This attempt is largely unsuccessful because the system cannot reason
about the model of each action and so cannot find how the undesirable side ef-
teffects of an action can be avoided. Numeric modelling is also examined in Aelion
not clear that the existing work can solve the two example problems considered
here.

optimisation . There is sometimes a motivation to find the best way to perform
a task. Some tasks are particularly expensive to perform and others tasks are
carried out particularly frequently. There is also a motivation to avoid using
the worst way to perform a task. Some tasks can be achieved in a clumsy way
that should be avoided. The scheduling paradigm uses the idea of optimisation.
However, its not clear how optimisation and AI planning can be combined in
an efficient way.

user interface design . People often want to have some influence on procedures
they will use. For OPS to be acceptable for many applications, a user must
be able to work with the OPS system to generate a procedure. It is not clear
how this interaction should be structured. No work has been done in this area
in the OPS community and little work has been published in the AI planning
community (Allen & Coworkers, 1994; Christianson & Kwoc, 1995).

reasoning about time . Some problems require an OPS system to reason about
time. In some problems, one event has to occur within a fixed time. For
example, "a particular chemical is unstable and must be used as quickly as
possible after it is synthesised". In some problems, an event must occur some
time after another event. For example, "three hours after starting the batch
add the contents of vessel 5". An OPS system should be able to reason about time so that these constraints can be represented. In the OPS community, only the scheduling paradigm uses an explicit representation of the notion of time. However, there is work in the AI planning community on reasoning about time (see Allen, Hendler, & Tate, 1990).
References


Tate, A. (1976). Project Planning using a Hierarchical Non-Linear Planner. D. a.i. research report 25, Department of Artificial Intelligence, University of Edinburgh.


Weld, D. & Etzioni, O. (1994). The first law of robotics (a call to arms). In *Proc. 12th Nat. Conf. of A.I.*


Appendix A

The CEP Modelling Language

The Chemical Engineering Planner (CEP) is an operating procedure synthesis system that has been under development at Loughborough since mid 1993. This manual describes modelling language used in CEP version 0.15.315.

This manual is divided into three sections. The first section develops a model for a simple plant and the basic concepts in the CEP language are presented in the process. Second section continues the development of the example in order to show the more advanced features of CEP. The third section looks at the model of valve sequencing tasks in CEP.

A.1 Introduction to the CEP language

This section is based on developing a model for the startup of the forced reboiler in figure A.1. The model that will be developed will be sufficient to create the very simple procedure “start pump-201” then “start heater-201”.

The most basic component of the CEP modelling language is a name. A name is a word used to describe a particular object or operator or variable in a CEP program.

The syntax of a name is defined by three rules.

1. Any sequence of characters in single quotes is a name. For example ‘vent valve’.

2. A name is also a sequence of characters, starting with one of the characters in \{a-z, A-Z, \_ \} and proceeding up to, but not including, the first character which
Figure A.1: A forced reboiler

is not in \{a-z, A-Z, _, +, -\}. For example CEP, valve-24, C++ and _off_.

3. However, the CEP keywords given in figure A.2 are not names. For example, 'info' is not a name although it satisfies the second rule.

The simplest named structures in CEP are instances and frames. An instance is the name of an individual unit, chemical or state. A frame is a class of instances. For example, if methane, chlorine and oxygen are declared as instances then the frame gas may be defined as a class containing these instances.

Instances are declared using the instance construct. The three basic formats of this construct are shown in figure A.3. Each format defines an instance name and declares it as a member of none, one or many frames. Frame declarations have a similar format, also shown in figure A.3.

The declaration of the instances and frames in the reboiler model is shown in figure A.4. It is good practice to define frames for all common classifications of an object. For example in figure A.4, pump-201 is defined to be a pump and also a machine.

Instances are one type of plan symbol. Variables are the other type. A variable is written ?name. In the CEP language, variables and instances can often be used interchangeably.

Procedure synthesis involves forming a plan to changing a plant section from an initial state to a goal state. A fundamental construct in the CEP language is a condition, a description of the state of some aspect of a plant. An examples of a
Reserved Words

<table>
<thead>
<tr>
<th>Reserved Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>achieve</td>
</tr>
<tr>
<td>arc</td>
</tr>
<tr>
<td>default</td>
</tr>
<tr>
<td>depend</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>exclude</td>
</tr>
<tr>
<td>expand</td>
</tr>
<tr>
<td>frame</td>
</tr>
<tr>
<td>goals</td>
</tr>
<tr>
<td>of</td>
</tr>
<tr>
<td>operator</td>
</tr>
<tr>
<td>property</td>
</tr>
<tr>
<td>ref</td>
</tr>
<tr>
<td>require</td>
</tr>
<tr>
<td>restrictions</td>
</tr>
<tr>
<td>set</td>
</tr>
<tr>
<td>start</td>
</tr>
<tr>
<td>tis</td>
</tr>
<tr>
<td>unknown</td>
</tr>
</tbody>
</table>

Figure A.2: Reserved words in CEP

<table>
<thead>
<tr>
<th>Figures</th>
<th>Instances</th>
<th>Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.3</td>
<td>instance</td>
<td>frame</td>
</tr>
<tr>
<td></td>
<td>(name)</td>
<td>(name)</td>
</tr>
<tr>
<td></td>
<td>(name isa frame)</td>
<td>(name isa frame)</td>
</tr>
<tr>
<td></td>
<td>(name isa [frame, ...])</td>
<td>(name isa [frame, ...])</td>
</tr>
</tbody>
</table>

Figure A.3: Frames and instances have three basic formats

```
frame(state).
instance(off isa state).
instance(on isa state).
frame(machine).
frame(heater isa machine).
instance(heater-201 isa heater).
frame(pump isa machine).
instance(pump-201 isa pump).
frame(column).
instance(column-201 isa column).
```

Figure A.4: The atoms and classes in the forced reboiler model.
Figure A.5: Conditions can be written in either of two formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>property of subject is state</code></td>
<td><code>state of heater-201 is on</code></td>
</tr>
<tr>
<td><code>property (subject, ... ) is state</code></td>
<td><code>contains(valve-1, methane) is true</code></td>
</tr>
</tbody>
</table>

Figure A.5: Conditions can be written in either of two formats

```
start {
  state of pump-201 is off;
  state of heater-201 is off;
}
```

Figure A.6: The start state for the forced reboiler problem

condition is 'state of heater-5 is on' and 'aperture of valve-1 is open'.

There are two methods for describing conditions in CEP, both shown in figure A.5. The first format is more readable but the second is more expressive and so its common to use both formats within the one program. In the format templates shown in figure A.5, the subject and state fields are filled by plan symbols while the property field can be assigned any arbitrary name.

Conditions can be viewed as function value pairs; \( f(a) = b \) or in this case \( property(subject) is valve \). Using this view, it is easy to see that two conditions are contradictory if they have the same property and subject but have different states. For example, state of heater-1 is on contradicts state of heater-1 is off.

Contradictory conditions can exist in a valid plan so long as these conditions are not achieved at the same time. For example, if state of heater-1 is on is achieved at one point and state of heater-1 is off is achieved some time later, heater-1 will simply be on up until the point it is turned off.

Using the condition construct, an OPS problem can be defined by defining an initial state and a goal state. The initial state is the state the operating procedure created by CEP will be used. The goal state describes the objectives the operating procedure should achieve.

The initial state is described by the start construct. The start construct has the format `start { condition_list }`. The start state for the reboiler problem is shown in figure A.6.
goals {
  require
  state of pump-201 is on;
  state of heater-201 is on;
  end
}

Figure A.7: The goal state for the forced reboiler problem

Ideally, the start state should specify a state for every property/subject pair the planner might need to consider. If the state of some property/subject pair is not specified, it is assumed to be in the least helpful state. That is, if CEP is trying to achieve a condition like state of heater-1 is on and the start state says nothing about the state of heater-1 then CEP will assume that heater-1 is not on.

The goal construct defines the goal state for a problem. The goal state lists the conditions that should be valid at the end of the operating procedure. The goal state for the reboiler problem is shown in figure A.7.

In contrast to the start state, the goal state need not provide a complete description of the plant. The goals should list only the conditions which should necessarily be true at the end of planning. For some problems, this may be simply be a high level condition like 'shutdown of pump-201 is true'.

The start and goals constructors differ in their format because the goals constructor allows the user of variables. This feature is not often and will not be described explicitly except to say that use of variables in the goal state is similar the use of variables in the operators construct, to be described.

Actions are the basic tasks which the agent will be told to perform when carrying out a procedure. The level of detail required in an action, e.g. the definition of a basic task, is dependent on the agent. Obviously a human agent will need different kinds of instructions from a mechanised agent. It is the responsibility of the CEP programmer to get this detail right.

An operating procedure can be viewed as an ordered sequence of actions. CEP's task is to find a sequence of actions to achieve a given goal state from a given initial state. This task is known as planning.

To facilitate planning, actions are modelled by their preconditions and effects. The
effects define what goals the action can be used to solve. The preconditions define the subgoals that must be achieved before the action can be used. For example, the effect of turning on heater-201 is that the heater becomes active. The precondition is that the heater-201 must be off to start with.

In the CEP language, actions are described by operators. An operator is a template for a class of actions. For example, an operator might be written to describe the class of actions for turning on a machine.

An operator for the class of actions which turn on machines in the forced reboiler problem is shown in figure A.8. The operator can be read as "in order to turn on a machine ?m, achieve some situation where ?m is off and then tell the agent to turn on ?m."

The operator shown in figure A.8 introduces a few new concepts. One of the most noticeable is the concept of a variables. The support for variables in operators is important because it allows an operator to represent many similar actions and it often also allows CEP to delay the selection of a particular action from the class of actions represented by an operator. Delaying action selection is a very good mechanism for reducing planning time.

Variables are declared in the top of an operator command. Each declaration has the format 'frame ?name;'. The frame must be a declared frame but the ?name can be any arbitrary name that is preceded by a question mark. For example, 'heater ?heater;' is valid and defines ?heater to be any one of the members of the set heater.
During planning operators are *instantiated* to form actions. This involves creating a unique set of internal variables from the external variables in the operator definition. The process is similar to calling a C function or a Prolog predicate; in all cases a unique set of internal variables are created for the local variables in the function.

The handling of internal variables is intelligent. As planning goes along, the variables refine their set of values as part of the process of keeping within the constraints on the plan. For example, if an action achieves state of ?machine is on and its unsafe to use heater-201 during this particular procedure, then ?machine will exclude heater-201 from its set of possible values. If the set of possible values for an internal variable contains only one value, the variable is bound to that value. In this case, ?machine would bind to pump-201.

Of course variables can also be bound directly. For example, if the planner needs to achieve state of pump-201 is on then the planner will bind ?machine to pump-201 directly.

To make the best use of these intelligent variables, avoid using classes which contain more than the atoms which are needed for a particular task. For example, in the reboiler problem, it would be possible to create a frame *everything* as the set of all machines, columns and states. It would then be legal but undesirable to assign a variable to the set *everything* when that variable actually represents a state.

Another interesting feature of the operator in figure A.8 is the use of starred conditions in the list of effects. There are actually two different types of effects; useful effects and side effects. Actions are added to the plan in order to achieve some effects. The useful effects of an action’s operator describe the effect that the action might be proposed to achieve. Side effects are effects which can be achieved much more simply by some other method. For example, an operator to clean out a particular type of vessel may cause the drain of the vessel to be opened but the operator should not be used just to open the drain. In the CEP language, the useful effects of an action are the effects which have a star next to them.

Apart from the confusion of useful effects and side effects, preconditions and effects are both just a sequences of conditions.
CEP Version 0.12.0

A possible plan is ...

{} start-1 [3, 4]
[3, 4] end-2 []

1) Turn on pump-201.
2) Turn on heater-201.

planning succeeded.

Figure A.9: Running a CEP program.

The final section of the operator is the print section. This section is just a sequence names and or variables, terminated by a semicolon. For example, the StartHeater operator in figure A.8 has the print statement `print 'Turn on ' ?m;`. When print instructions are processed, names print as themselves and variables print as the name of the atom that the variable is bound to, or the variable name if unbound. If ?m is bound to heater-201 then the print statement of StartMachine is equivalent to 'print 'Turn on heater-201';. Note that space is not automatically inserted around the variable ?h and so a trailing space is needed in the name 'Turn on '.

Now that we have provided models for the atoms and operators in the forced reboiler, and now that the start and end states have been defined for the problem, our CEP program is almost ready to run. All that is left to do is to save these definitions into a single file, which we will call 'dom.reboil.1'.

CEP is a one pass interpreter. As a result, structures can’t be referenced unless they have been defined higher up in the program file. The result is that atom definitions usually have to come at the top of the file. Otherwise, the statements in a CEP program can appear in any order.

Figure A.9 shows the effect of running the forced-reboiler program. Planning time for this simple problem is a few milliseconds.

The first section of the output describes the order constraints on actions in the generated operating procedure. All the actions in the plan are listed in an order
which forms one of the acceptable orders of the plan. To the left and right of each action name is a list action numbers. Actions numbered to the left of an action must performed strictly before the action. Actions numbered to the right must be performed strictly after the action. These constraints can be modelled as a PERT chart as shown in figure A.10.

The second section of output list the instructions to form one possible operating procedure. As discussed, reordering the actions within the constraints provided will form other procedures.

A.2 More advanced programming

In this section we go back and refine the model of the reboiler. In the process some of the features of CEP that were glossed over in the last section are introduced.

A.2.1 Restrictions

Some situations are unsafe or otherwise undesirable. For example, in the procedure for the forced flow reboiler shown in figure A.9 the heater may be started before the pump. However, some heaters will melt if a flow of chemical is not provided for cooling.

CEP provides restrictions to define the situations that should not occur during the execution of a plan. A restriction is simply a set of conditions, all of which cannot all be true at the same time. For example, the forced reboiler problem already contains the implicit restriction ‘state of pump-201 is on, state of pump-201 is off’ simply from the semantics of the CEP condition structure.

An example of a restriction declaration in the CEP language is shown in figure A.11. Note that the statement allows for the use of variables. These variables
restrictions {
  pump ?pump;
  heater ?boiler;
  column ?column;
  prevent
    pump of ?column is ?pump;
    heater of ?column is ?boiler;
    state of ?pump is off;
    state of ?boiler is on;
  end
}

Figure A.11: A restriction to prevent a boiler overheating

are implicitly universally quantified. For example, the restriction in figure A.11 reads “for each column, do not allow a situation to occur where heater of that column is on and the boiler is off”.

Before the restriction in figure A.11 can be used, the following two conditions must be made true throughout the plan:

\[
\begin{align*}
pump \text{ of } column-201 &= pump-201 \\
heater \text{ of } column-201 &= heater-201
\end{align*}
\]

No operator in the domain can be used to deny either of these conditions. As a result, the conditions can be made true throughout the plant by placing them in the start state. This is undesirable for some complex reasons which are not important right now. It is better represent the statements in the frame hierarchy as described in section A.2.2.

The current version of CEP assumed that the start state is safe with respect to the goals of prevention in the plan. It is the responsibility of the programmer to make sure that this assumption holds.

A.2.2 Frames/instances with properties

Some conditions are valid throughout the plan. We will call these conditions constants. Constants usually form part of the description of a piece of equipment, for example heater of column-201 is heater-201.
Constants can be listed as part of the start state for an OPS problem. However, in theory the start state should only contain problem-specific information and so representation scheme is undesirable. Ideally, a declaration of constants should form part of the plant model. This declaration can be made using the frame or instants constructs.

The frame and instance constructs have an extra parameter that has so far not been discussed. This parameter holds a list of slot/fillers pairs as shown below:

```
instance(item isa frame,
        [ slot1 type1 filler1, slot2 type2 filler2, ...]).
```

There are a number of different values that can be given to the type of a slot. Type ‘is’ slots take a property as the slot name and a atom as the filler and create a constant of the form:

```
item of slot is filler
```

For example, the restriction given in section A.2.1 requires the definition of two constants; heater of column-201 is heater-201 and pump of column-201 is pump-201. These conditions can be defined using the following statement:

```
instance(column-201 isa column,
        [ heater is heater-201,
          pump is pump-201 ]).
```

Note that constants with more than one subject cannot be predefined in this way. Currently, these more complex constants have to be specified in the start state.

Frames can be associated with slot/filler pairs in the same way as instances. The slot/filler pairs of a frame are inherited by each of the children (instances) of that frame unless the child defines the same slot itself.

A.2.3 Predefining frames and instances

Sometimes it is necessary to write instance statements which reference each other. For example:
instance(column-201 isa column, [ heater is heater-201 ]).  
instance(heater-201 isa heater, [ column is column-201 ]).  

As discussed, CEP will not allow instances or frames to be used before they are defined. As a result, the above two statements cannot be written in any order without causing an error.

To get around this problem, CEP provides a preDef statement to predefine frames and instances. The command preDef is a function which takes the keyword instance or the keyword frame as its first argument and a square bracketed list of atoms as its second argument. For example, the following sequence of instructions will not produce an error.

\[
\text{preDef ( instance, [column-201, heater-201] ).}
\]

\[
\text{instance(column-201 isa column, [ heater is heater-201 ]).}
\]

\[
\text{instance(heater-201 isa heater, [ column is column-201 ]).}
\]

A.2.4 Printing slot values

In an operating procedure, the agent quite often needs to be given some quite detailed, machine specific instructions. For example “turn on heater-201 by initiating automatic startup sequence 5901 from a console in the control room”. Similarly “turn on heater-95 by pressing the red button on the side of the machine”.

To represent these complex instructions, it is possible to create a planning operator for each component. This is undesirable both because it increases the risk of error in the plant model and because it increases the number of similar operators in the model which will in turn increase planning time. As an alternative, CEP allows print statements to access the slot/fillers of an instance through the use of complex variables. A complex variable has the format shown below.

\[
(?z) \{ \text{property of } ?opVar \text{ is } ?a; \text{property of } ?a \text{ is } ?b; \ldots; \text{property of } ?y \text{ is } ?z; \};
\]
In the above format, \( ?a \) through \( ?z \) are arbitrary variable names which do not form part of the operators defined variables; and \( ?opVar \) is a variable which is defined in the operator. Strictly speaking, the \( ?opVar \) can also be a constant but this feature is used vary rarely. An example of the use of complex variables is shown below.

\[
\text{instance('turning on switch SW-203').}
\]
\[
\text{instance(heater-201 isa heater,}
\]
\[
\quad \text{[ startup-method is 'turning on switch SW-203' ]}.
\]

\[
\text{operator {}
\]
\[
\quad \text{heater } ?h;
\]
\[
\quad \ldots
\]
\[
\quad \text{print 'Turn on ' } ?h ' \text{ by ' } (?i)(\text{ startup-method of } ?h \text{ is } ?i;);
\]
\[
\quad \ldots
\]

In the case that \( ?h \) is bound to heater-201, the print statement above would produce the instruction Turn on heater-201 by turning on switch SW-203.

### A.2.5 The ‘tis’ slot type

Description information is often included in the slots of an instance or frame. CEP requires that is slots take a defined instances as fillers. This often results in the need for some quite awkward code, for example:

\[
\text{instance('turning on switch SW-203').}
\]
\[
\text{instance(heater-201 isa heater,}
\]
\[
\quad \text{[ startup-method is 'turning on switch SW-203' ]}.
\]

To avoid this clumsiness, CEP provides the tis slot type. The tis slot behaves like an is slot except that the slot’s filler is defined as an instance if the filler is not defined already. For example, the code given above is equivalent to:

\[
\text{instance(heater-201 isa heater,}
\]
\[
\quad \text{[ startup-method tis 'turning on switch SW-203.' ]}.\]
A.2.6 Defining properties

By default, a condition can take any arbitrary name as its property. The disadvantage is that CEP cannot detect and warn about spelling mistakes in property names. Similarly, CEP cannot detect and warn about a property with the wrong arity, or number of subjects.

However, CEP has an optional mechanism for defining legal properties and their arity. After one property has been defined, the use of an undefined property will produce an error. The property statement has the format shown below. A CEP program can contain any number of property statements.

property property/arity, property/arity, ...;

For example, the forced reboiler model given in section A.1 contains only one property. The property is defined below.

property state/1;

A.2.7 The expansion operator

When a person is asked to write an operating procedure, the person is usually asked to write a procedure for some high level task. For example, a procedure to start the reboiler. In contrast, the problem specification given in section A.1 defines its goal state in terms of low level conditions; conditions describing the state of the heater and the pump. In this section we show how the forced reboiler problem can be specified using a single high level goal.

High level goals don’t have meaning in themselves, they have meaning by convention. If I am asked to switch off a computer then I will switch off the screen and the processor and any other devices. Somewhere in my mind I have a mapping between the high level objective ‘switch off the computer’ and the lower level objectives ‘switch off the processor’, ‘switch off the screen’ etc.

If the computer is required to perform high level tasks then it must be given a mapping from the high level task to a set of lower level goals.
operator StartReboiler {
  heater ?boiler;
  pump ?pump;
  column ?c;

  expand
  * reboiler-state of ?c is on;
  using
    heater of ?c is ?boiler;
    pump of ?c is ?pump;
    state of ?boiler is on;
    state of ?pump is on;

  end
}

Figure A.12: An expansion operator

In CEP, this mapping is declared using an expansion operator. An example is given in figure A.12. The format of the operator is very similar to the format of the normal achieve operator shown in figure A.8. There are only three differences. Firstly, the keyword achieve is replaced by the keyword expand. Secondly, the effects and preconditions of the operator are replaced by a single high level goal and a set of equivalent low level goals respectively. Finally, the print statement has no meaning in the expand operator because the operator is used to modify the preconditions of an existing action rather than to solve a precondition by creating a new action.

With this expansion operator, the goal for the reboiler problem becomes simply ‘reboiler-state of column-201 is on’.

A.2.8 Variable constraints

Section A.1 provides a planning operator to start a machine (see figure A.8). For completeness, the domain description should also include a StopMachine operator.

It is possible to write StopMachine so that is just the opposite of StartMachine. This can be done by simply replacing every occurrence of on in StartMachine by off and every occurrence of off by on. For example, rather than achieving the goal state of ?m is on, StopMachine will achieve state of ?m is off.

It is undesirable to have to write and maintain two very similar operator, like StopMachine and StartMachine. CEP is powerful enough to combine these two op-
operators into a single construct by using variable constraints.

Variable constraints are statements about the relationship between plan symbols. There are two types of variable constraint supported by CEP. Unification constraints equate two symbols. The unification constraint \(?a == ?b\) implies that if \(?a\) was bound to a particular value then \(?b\) should also be bound to that value and similarly if \(?b\) is bound to a particular value then \(?a\) should be bound to that value (eqn. A.1). Separation constraints prevent the unification of symbols. For example the separations constraint \(?a != ?b\) implies that if \(?a\) is bound to a particular then \(?b\) cannot be bound to that value and similarly if \(?b\) is bound to a particular value then \(?a\) cannot be bound to that value (eqn. A.2). Although the two plan symbols in the equations are represented as variables, in practice either or both can be atoms.

\[
\begin{align*}
?a == ?b & \iff (?a = x \to ?b = x) \land (?b = y \to ?a = y) \\
?a != ?b & \iff (?a = x \to ?b != x) \land (?b = y \to ?a != y)
\end{align*}
\]

(A.1) (A.2)

In CEP, variable constraints are recorded and evaluated as soon as the constraint has a clear meaning. For unification constraints involving two atoms, the constraint is evaluated immediately and planning either continues or backtracks depending on whether the atoms are the same. For separation constraints involving two unbound variables with overlapping sets of possible values, the constraint is evaluated when one of the variables is bound to an atom. The result of this separation is to remove the atoms from the set of possible values of the unbound variable. The unbound variable may then have only one possible value and so may become bound, causing more separation constraints to fire. This delayed evaluation strategy can be seen as an improvement on the immediate evaluation policy used in Prolog\(^1\).

As stated, variable constraints can be used to create an OperateMachine operator

---

\(^1\)Prolog variable constraints are evaluated immediately. This can lead to some surprising results. In the Prolog \(A \leftarrow B, A = 1, B = 1\) evaluates to true and \((\leftarrow A = B), A = 1, B = 2\) evaluates to false. These anomalies do not appear in CEP.
operator OperateMachine {
    machine ?m;
    state ?startState;
    state ?endState;

    ?startState != ?endState;

    achieve
        state of ?m is ?endState;
    require
        state of ?m is ?startState;
    end

    print 'Turn ' ?endState ' ' ?m ' by operating ' (?method) [ operate-method of ?m is ?method; ]. }

Figure A.13: An operator to start and stop a machine

for the forced reboiler example. The class state has already been defined to describe
the set of instances on and off. If ?startState and ?endState are defined as states and
?startState != ?endState then ?startState and ?endState take opposite values, if one is
off then the other is on. We cannot create an operator that represents StartMachine
and StopMachine as shown in figure A.13.

In planning, variable constraints tend to be used in three ways. Firstly, the con-
straints can be used to define a pair of opposite values as described above. Secondly,
constraints can be used to create variables that belong to more than one set. For
example, a variable ?flammableGas can be created by defining the variable as a gas
and then unifying it with a variable ?flammable which is defined as a combustible
chemical. Finally, separation constraints are used in defining restrictions. For ex-
ample, the restriction 'a hot pipe cannot contain oxygen' must be written as 'the
pipe must either be not hot or must not contain oxygen'. The concept of 'not hot'
is created by defining a variable ?temperature and separating this variable from the
value hot.

A.2.9 Early effects

Sometimes it is important that some effects of an action happen earlier than other
effects of the action. Consider the case when a pipe is leaking some flammable gas and
there is a fan that will sufficiently disperse the gas to avoid an explosion. Starting the
fan involves turning on a particular switch and the switch may spark. In reasoning about this problem it is important to know whether the spark will occur before the gas is dispersed, in which case there will be an explosion, or whether the spark will occur when the gas is dispersed, in which case the action is safe. The CEP modelling language, as described so far, can only represent the latter situation.

CEP allows an effect of an action to be declared as occurring early. Early effects are flagged by the symbol '&lt;'. All early effects are modelled as occurring at a single time point and this time point is before the point at which the other effects occur.

Early effects can be used to describe the operation of turning on a fan. In the model in figure A.14, a spark is created early during the action but the spark is gone before the action is over. The action also clears away gas from near the fan but this happens after the spark is created.

A.2.10 Comments

Almost all programming languages support the use of comments, text which is ignored by the computer but is included to make the program easier for a person to read. The CEP language allows two kinds of comments, the C style (/\* \* /) and the Prolog style (\% \* <cr>). In the C comments, anything between a /\* and */ is treated as a comment and ignored by the interpreter. These comments may extend over many lines. In the Prolog comments, anything between the % and the end of line is treated as a comment. Both styles of comment may be used in the one program.
A.2.11 The final program

The final CEP program to model the forced-reboiler and define the start-up task is shown below:

```plaintext

property heater/1, compressor/1, operate-method/1, state/1, reboiler-state/1;

frame(state).
instance(off isa state).
instance(on isa state).

preDef ( instance, [ heater-201, compressor-201 ] ).

frame(column). % "column" could be defined later
instance(column-201 isa column, % to avoid the need for a preDef.
  [ heater is heater-201,
    compressor is compressor-201 ] ).

frame(machine).
frame(heater isa machine).
instance(heater-201 isa heater,
  [ operate-method tis 'the switch on the heater' ]).

frame(compressor isa machine).
instance(compressor-201 isa compressor,
  [ operate-method tis 'switch SW-502 in the control room' ]).

operator StartReboiler {
  heater ?boiler;
  compressor ?pump;
  column ?c;

  expand
    * reboiler-state of ?c is on;
    using
    heater of ?c is ?boiler;
    compressor of ?c is ?pump;
    state of ?boiler is on;
    state of ?pump is on;
  end }

operator OperateMachine {
  machine ?m;
  state ?startState;
  state ?endState;

  ?startState != ?endState;

```
achieve
    * state of ?m is ?endState;
using
    state of ?m is ?startState;
end

print 'Turn ' ?endState ' ' ?m ' by operating '
    (?method) [ operate-method of ?m is ?method; ]; }

restrictions {
    compressor ?pump;
    heater ?boiler;
    column ?column;

    prevent
        compressor of ?column is ?pump;
        heater of ?column is ?boiler;
        state of ?pump is off;
        state of ?boiler is on;
    end
}

/*
    Problem specification to start up the reboiler.
*/
start { state of compressor-201 is off;
          state of heater-201 is off; }
goals { require
          reboiler-state of column-201 is on;
        end }

A.3 Valve sequencing

Valve operation cannot be modelled easily using the operators described in the last two sections. Primarily, this is because the effect of opening a valve is too dependent on the state of the plant before the valve is open. Consider figure A.15. Opening valve-a will cause hydrogen to flow into the reactor. However, the hydrogen will also flow through the reactor and downstream if valve-b or valve-c allows this to happen. See table A.1.

One solution to the valve modelling problem is to create a set of open valve operators, each specific to a particular valve and a particular state of the plant. However, planning becomes more inefficient as more operators are added to do equivalent tasks.
Figure A.15: A simple valve sequencing problem

<table>
<thead>
<tr>
<th>Plant State</th>
<th>Resulting propagation of Hydrogen through the plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valve b</td>
<td>Valve c</td>
</tr>
<tr>
<td>closed</td>
<td>closed</td>
</tr>
<tr>
<td>open</td>
<td>closed</td>
</tr>
<tr>
<td>closed</td>
<td>open</td>
</tr>
<tr>
<td>open</td>
<td>open</td>
</tr>
</tbody>
</table>

Table A.1: The effect of opening valve-a in a section of plant

While it is difficult to predict the effect of opening a valve, it is relatively easy to predict the effect of creating a flow. Consider the problem of creating a flow from the Hydrogen Inlet to Product Outlet in figure A.15. Obviously valve-a and valve-b have to be opened. In the general case, a route must be found between the source and the sink and this route must be opened up. If there is more than one possible route, one possibility should be selected now and the other routes should be examined on backtracking. When creating the path for the plant in figure A.15 it is also necessary to have valve-c closed before opening the other valves so that hydrogen does not flow through the vent. In general, it is good practice to close all valves bordering the flow path so that the flowing chemical contaminates as few components as possible. The effect of closing valve-c and then opening valve-a and valve-b is that hydrogen flows to the outlet and also contaminates the reactor. In general, all the units along the flow path are contaminated by the chemical that is flowing. The chemicals previously trapped in the flow path may also mix. There is no other effect.

CEP contains built in functions to translate flow goals into goals for the operation of valves. In the process, these functions deduce the effect of the flow. Note however that in the current implementation of CEP, the system ignores mixing between the
residue chemicals in the intermediate pipes of a flow or purge operation. In the models that have been implemented in CEP so far, this has not caused a problem. Chapter 6 contains a discussion on how this limitation might be overcome.

A.3.1 Plant modelling

For valve sequencing, CEP uses QUEEN style flow modelling. Modelling takes place in two stages. Firstly a model is made of the flow through each type of unit. Later instances of units are joint together to form the model of a plant.

In CEP, the model of a unit simply describes the flow of chemical from the in-ports to the out-ports of a unit. For a pipe, the chemical that flows in one end also flows out the other. For the model of a divider, chemical that flows through the in-port will end up in both of the out-ports. For the model of a heat exchanger, two separate flows of chemical species are formed and the two flows don't mix.

Generic Units are modelled by filling the propLinks slot when the unit type is defined as a frame. The template for this definition is shown below.

```plaintext
frame(new_frame isa frame,
    [ propLinks info [
        arc([in, composition], 1, [out, composition]),
        arc( ...]
    ])
```

In the template given above, in and out are names of a in-ports and a out-ports of the unit respectively. Port names do not have to be declared and so any arbitrary name can be used. Essentially the template declares a qualitative model, positively relating the composition of the in-port to the composition of the out-port. In QUEEN, arcs can also be defined to relate to the temperature, pressure, etc. of an in-port and an out-port. In CEP, arcs which are not concerned with composition have no meaning to the system and are ignored.

As an example of a unit models, consider modelling the reactor shown in figure A.15. The reactor has three ports, an inlet from valve a, an outlet to valve b and an outlet to valve c. Call these three ports in, out1 and out2 respectively.
The links between the individual units in a plant are declared by filling in the *outports* slot when the individual unit is defined as an instance. The slot simply relates an out-port of one unit to the in-port of another. The template for this definition is shown below.

\[
\text{instance}(atom \text{ isa unit.type, [}
\text{ outports info [out is [unit, in]],}
\text{ outports info ...}]
\]}

In the template given above, *out* and *in* are port names and *unit* is an instance representing some unit of the plan. To simplify modelling, if *unit* is not already declared then it is automatically predefined as instance.

Note that the template has more than one outports slot. The *info* slot type takes a comma separated list of structures as its filler. If a frame or instance has two or more info slots with the same name, the list of fillers for these slots are concatenated together. The template above is equivalent to a template with a single outports slot and a comma separated list of `out is [unit, in]` structure within that slot.

As an example of a outports definition, the reactor for the plant shown in figure A.15 has been modelled as presented below.

\[
\text{instance}(reactor-1 \text{ isa reactor, [}
\text{ outports info [out1 is [valve-b, in]],}
\text{ outports info [out2 is [valve-c, in]]}
\]}
\]
<table>
<thead>
<tr>
<th>Frame</th>
<th>Instances</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true</td>
<td>The validity of a statement.</td>
</tr>
<tr>
<td></td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>aperture</td>
<td>open</td>
<td>The state of a valve.</td>
</tr>
<tr>
<td></td>
<td>closed</td>
<td></td>
</tr>
<tr>
<td>chemical</td>
<td>any</td>
<td>The set of all chemicals that will be used in the plant.</td>
</tr>
<tr>
<td>valve</td>
<td>any</td>
<td>Any unit that controls flow.</td>
</tr>
<tr>
<td>port</td>
<td>none</td>
<td>Defined by CEP automatically. It is the set of ports used in the model.</td>
</tr>
</tbody>
</table>

Figure A.16: The required frames and instances

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aperture of valve is aperture</td>
<td>The state of a valve.</td>
</tr>
<tr>
<td>contains(unit, port, chemical) is boolean</td>
<td>The contents of a unit.</td>
</tr>
</tbody>
</table>

Figure A.17: The required conditions

A.3.2 Special frames, instances and properties

The CEP valve sequencing functions translate high level goals into low level goals and effects. The low level information must be in a format which can be used in the current plant model. For example, the structures used in the sequencing engine must also be used in the plant model.

Figure A.16 shows the frames and instances which are used by the valve sequencer. These structures must be defined in a model before valve sequencing can be used. However, if valve sequencing is not used then the frames and instances in figure A.16 have no special meaning.

Figure A.17 shows the format of the conditions which are produced by the valve sequencer. If valve sequencing is to be used, these conditions should have meaning in the plant model. For example, operators should be available in the model to solve the goal aperture of ?v is closed for each different type of valve ?v.

The conditions in figure A.17 should perhaps be viewed as signals in an object oriented language rather than as low level facts. For example, the goal aperture of ?v is open may be solved differently for different valve types. For example, the class of valves might be sub-classified into ‘ball valves’, ‘block and bleed valves’ and ‘hose connectors’. Using this sub-classification scheme, opening a hose might involve a
number of steps so that the hose is moved to its connector and then fastened and
the valves around the connector are then opened. In the same model, opening a ball
valve might simply involve pressing a button in the control room.

Just as aperture goals can be solved differently for different valve types, contents
goals can be solved differently for different unit types. A goal of the form ‘contents
(?unit, ?port, ?chem) is false’ can be viewed as a request for a purge. In smaller units,
the goal might be solved by simply creating a flow of purgative through the unit.
Larger units may require a number of pressure up blow down cycle before they are
cleaned sufficiently.

A.3.3 Valve sequencing functions

The valve sequencer, if active, will solve the conditions listed below. The valve
sequencer can be activated by defining the special frames, instances and properties
described in section A.3.2 are defined.

noFlowUpstream(unit, port) is true

Prevent chemical flowing into unit, port by closing all valves that are connected
directly, i.e. not via other valves, to the upstream side of unit, port.

noFlowDownstream(unit, port) is true

Prevent chemical flowing out of unit, port by closing all valves that are connected
directly, i.e. not via other valves, to the downstream side of unit, port.

flow(start-unit, start-port, dest-unit, dest-port, chemical) is true

Open the valves necessary to create a flow of chemical from start-unit, start-port to
dest-unit, dest-port. Valves around this flow are closed before the flow is created to
prevent the chemical propagating to unnecessary places. The chemical is added to
the correct ports on all the intermediate units and to the destination unit and port
and in branch units between the flow path and the closed downstream valves. The
chemical species already in the intermediate units are not purged (removed).
Figure A.18: The units contaminated by a flow from Start to End

Figure A.18 shows the effects of a flow between start and dest. All valve positions in the diagram are set by the planner. Note that even the valves marked a and b are closed although chemical flowing through these valves will not reach the chemical flowing directly between start and dest. This is to prevent mixing at the pipe c. Also shown in the diagram is the propagation of the chemical in the flow.

\[ \text{flow}(\text{start-unit, start-port, dest-unit, dest-port, chemical, close-upstream, close-downstream}) \text{ is true} \]

Often it is necessary to block flow into a tank and then to vent or drain the tank by creating a new flow. Goals can be achieved in any order and so if the block/vent task is specified as a flow goal and a noFlow goal, the tank may end up being drained while it is still being filling. In response to this problem, CEP provides a second format for the flow condition. If close-upstream is true then the valves upstream of start-unit, start-port will be closed before the flow is created. if close-downstream is true then the valves downstream of dest-unit, dest-port will be closed first.

\[ \text{purge}(\text{start-unit, start-port, purge-unit, purge-port, dest-unit, dest-port, chemical}) \text{ is true} \]

Open the valves necessary to create a flow of chemical from start-unit, start-port to dest-unit, dest-port. The chemical is added as in a flow. The chemical species originally in the relevant ports of all intermediate units along the flow path is removed. Units are also purged if they are separated from the flow path by at most a single unit (this distance can be change in the plant model) and which are inside the ring
of closed valves. The valve sequencer guarantees that \textit{purge-unit}, \textit{purge-port} will be included in the purged items.

Figure A.19 shows the effect of a purge flow from \textit{Start} to \textit{Dest} via \textit{Purge}. Note the units which are purged by the flow. The current algorithm for choosing purged units is only a first attempt to model the physical process. It highlights some issues involved in purge, like the problem of purging short branch lines. Although a more accurate model will be needed in later work, the current approach seems adequate to support such a model.

\section*{A.3.4 The default start state}

Valve sequencing problems usually involve working with a lot of very similar components. Associated with each component is a large amount of state information. For example, to define the contents of a unit, a statement is needed to relate the unit to each known chemical.

The start state should define the contents of each unit and the state of each valve in the plant. This is a very monotonous and error prone task if the \textit{start} language construct is used. To help, CEP provides the \textit{default} language construct. In \textit{default}, variables can be specified which are implicitly universally quantified. In other words, a conditions written using variables are true for all possible values of the variables they contains. Using this approach it is very easy to write very broad statement about the plant. For example, the statement below declares that all valves in the plant are closed.
Conditions defined in the start override the conditions in the default state. In the example, if there is a single open valve in the plant then default construct should be used to state that all valves are closed and the start construct should be used to define that the particular valve is open.

A.3.5 The bottleneck problem

To demonstrate the use of valve sequencing, the bottleneck problem will be examined. The problem is based on the plant shown in figure A.20. In the start state, hydrogen is flowing through the common pipe via valve-a and valve-x. The goal is to fill the oxygen tank and then restore the hydrogen flow.

The structure of this problem is quite common. Most maintenance tasks involve starting in the normal running state for the plant, performing some repair operation and then ending in the normal running state. For instance, the task of regenerating the catalyst in a reactor, given as an example in Rivas and Rudd (1974), is similar. CEP is the first OPS system that can solve these tasks without requiring the user to provide intermediate states or “planning islands”.

The problem also demonstrates the use of valve sequencing conditions, like aperture, as signals. Most valves are ball valves but the valve marked ‘b’ in figure A.20
is in fact a hose connector. The nitrogen hose must be connected to the appropriate port before a purge can begin.

property aperture/1, closed/1, compatible/2, connected/1, contains/3;
property flow/5, hosePort/1, noFlowUpstream/2, noFlowDownstream/2;
property purge/7, supply/1;

/∗ Required frames and instances ∗/

frame(boolean).
instance(true isa boolean).
instance(false isa boolean).

frame(aperture).
instance(open isa aperture).
instance(closed isa aperture).

frame(chemical).
frame(inert isa chemical).
frame(gas isa chemical).
instance(hydrogen isa gas).
instance(oxygen isa gas).
instance(nitrogen isa [gas, inert]).

/∗ Models ∗/

frame(unit).
frame(inlet isa unit). % Has a provides slot.
frame(outlet isa unit).

frame(header3 isa unit,
   [ propLinks info [  
   arc([in1, composition], 1, [out, composition]),  
   arc([in2, composition], 1, [out, composition]),  
   arc([in3, composition], 1, [out, composition]) ]]).

frame(divider3 isa unit,
   [ propLinks info [  
   arc([in, composition], 1, [out1, composition]),  
   arc([in, composition], 1, [out2, composition]),  
   arc([in, composition], 1, [out3, composition]) ]]).

frame(tank isa unit,
   [ propLinks info [  
   arc([in, composition], 1, [out, composition]) ]]).

frame(valve isa unit,
   [ propLinks info [  
   arc([in, composition], 1, [out, composition]) ]]).

frame(hose isa valve).
frame(ball-valve isa valve).

/* Plant Model */

instance(valve-a isa ball-valve, [ 
  outports info [out is [head-1, in1]] ]).

instance(nitrogen-hose isa hose, [ 
  outports info [out is [head-1, in2]], 
  hosePort tis 'Nitrogen Hose-Port' ]).

instance(valve-c isa ball-valve, [ 
  outports info [out is [head-1, in3]] ]).

instance(valve-x isa ball-valve, [ 
  outports info [out is [hydrogenOut, in]] ]).

instance(valve-y isa ball-valve, [ 
  outports info [out is [nitrogenOut, in]] ]).

instance(valve-z isa ball-valve, [ 
  outports info [out is [oxygenTank, in]] ]).

instance(hydrogenIn isa inlet, [ 
  outports info [out is [valve-a, in]], 
  supply is hydrogen ]).

instance(nitrogenIn isa inlet, [ 
  outports info [out is [nitrogen-hose, in]], 
  supply is nitrogen ]).

instance(oxygenIn isa inlet, [ 
  outports info [out is [valve-c, in]], 
  supply is oxygen ]).

instance(head-1 isa header3, [ 
  outports info [out is [div-1, in]] ]).

instance(div-1 isa divider3, [ 
  outports info [out1 is [valve-x, in]], 
  outports info [out2 is [valve-y, in]], 
  outports info [out2 is [valve-z, in]] ]).

instance(oxygenTank isa tank).

instance(hydrogenOut isa outlet).
instance(nitrogenOut isa outlet).

/* Operators */

operator OperateBallValve { 
aperture ?state;
ball-valve ?valve;

achieve
* aperture of ?valve is ?state;
end

print ?state ' valve ' ?valve; }

operator OpenHose {
  hose ?hose;

  achieve
  * aperture of ?hose is open;
  * closed of ?hose is false; % Needed for CloseHose.
  using
  connected of ?hose is true;
  end

  print 'Open the valve on the connector of ' ?hose; }

operator ConnectHose {
  % Second stage of OpenHose
  hose ?hose;

  achieve
  * connected of ?hose is true;
  end

  print 'Connect hose ' ?hose ' to ' (?p) [ hosePort of ?hose is ?p; ]; }

operator CloseHose {
  hose ?hose;

  achieve
  * aperture of ?hose is closed;
  * connected of ?hose is false;
  using
  closed of ?hose is true;
  end

  print 'Disconnect hose ' ?hose ' from ' (?p) [ hosePort of ?hose is ?p; ]; }

operator CloseHoseActual {
  % Second stage of CloseHose
  hose ?hose;

  achieve
  * closed of ?hose is true;
  * aperture of ?hose is closed; % No '*' means that this operator cannot be
  end
  % added to achieve aperture = closed.
print 'Close the valve on the connector of ' ?hose; }

operator FillTank {
    unit ?u;
    port ?p;
    chemical ?chem;
    inlet ?i;

    achieve
    * contains(?u, ?p, ?chem) is true;
    using
    supply of ?i is ?chem;
    flow(?i, out, ?u, ?p, ?chem) is true;
    end

    print 'Wait for <' ?u ', ' ?p '> to fill with ' ?chem ; }

operator DoPurge {
    unit ?u;
    port ?p;
    chemical ?chem;
    inlet ?in;
    outlet ?out;
    inert ?purgative;

    /* If you copy this operator...

    Increase the priority of purge/7 well above the priority of contains/3. A single purge operation may purge many units. In other words, the solution to a purge/7 goal may solve many contains/3 goals. To avoid confusing CEP, contains/3 goals should be converted to a purge/7 goal one at a time (there should never be more than one outstanding purge goal). */

    achieve
    * contains(?u, ?p, ?chem) is false;
    using
    supply of ?in is ?purgative;
    compatible(?out, ?purgative) is true;
    purge(?in, out, ?u, ?p, ?out, in, ?purgative) is true;
    end

    print 'Wait 10 minutes for the purge to complete.'; }

    /* Priority:

    supply/1 and compatible/2 are constants and should be solved as early as possible. purge/7 should be solved earlier than contains/3. aperture/1 is easy to solve and should be ignored as long as possible. */

    priority 1000 { supply of <unit> is <chemical>; }
priority 1000 { compatable(<unit>, <chemical>) is <boolean>; }
priority 400 { purge(<unit>, <port>, <unit>, <port>, <unit>, <port>, <chemical>) is true; }
priority -100 { aperture of <valve> is <aperture>; }

/* Problem Specification */

restrictions {
    chemical ?notHydrogen;
    chemical ?notOxygen;
    unit ?unit;
    port ?port;

    ?notHydrogen != hydrogen;
    ?notOxygen != oxygen;

    prevent % Don't send waste to hydrogenOut. 
        contains(hydrogenOut, in, ?notHydrogen) is true;
    end

    prevent % Don't send waste to the oxygen tank. 
        contains(oxygenTank, in, ?notOxygen) is true;
    end

    prevent % Don't mix hydrogen and oxygen. 
        contains(?unit, ?port, hydrogen) is true;
        contains(?unit, ?port, oxygen) is true;
    end }

default {
    valve ?valve;
    unit ?unit;
    chemical ?chem;
    header3 ?head;

    achieve
    aperture of ?valve is closed;
    connected of ?valve is false;
    contains(?unit, in, ?chem) is false;
    contains(?head, in1, ?chem) is false;
    contains(?head, in2, ?chem) is false;
    contains(?head, in3, ?chem) is false;
    compatable(hydrogenOut, hydrogen) is true;
    compatable(nitrogenOut, nitrogen) is true;
    compatable(oxygenTank, oxygen) is true;
    end }

start { aperture of valve-a is open;
    aperture of valve-x is open;
    contains (div-1, in, hydrogen) is true;
    contains(oxygenTank, in, oxygen) is false; }
goals { require
contains(oxygenTank, in, oxygen) is true;
flow(hydrogenIn, out, hydrogenOut, in, hydrogen) is true;
end }

When this model is run, the following output is produced. Note that the actions in square brackets are in fact null actions, they simply informs the operator that the plant is ready for a particular flow or purge to start.

CEP Version 0.14.39
A possible plan is ...

[ ] start-1 [11, 12, 26]

[11, 12] ClosePoint-6 [25, 27]

[25, 27] DoPurge-5 [15, 16]
[15] CloseHose-14 [4, 18]

[14, 16] ClosePoint-4 [28, 29]

[14] ConnectHose-18 [17]

[28, 29] FillTank-3 [10, 13]

[10, 13] ClosePoint-9 [17, 19]

[9, 18] OpenHose-17 [8]
[17, 19] DoPurge-8 [21, 22]

[20, 22] ClosePoint-7 [23, 24]

1) closed valve valve-a.
2) closed valve valve-x.
3) [ Close point for Purge of nitrogen from <nitrogenIn, out> to <nitrogenOut in>.
4) open valve valve-y.
5) Connect hose nitrogen-hose to Nitrogen Hose-Port.
6) Open the valve on the connector of nitrogen-hose.
7) Wait 10 minutes for the purge to complete.
8) Close the valve on the connector of nitrogen-hose.
9) Disconnect hose nitrogen-hose from Nitrogen Hose-Port.
10) closed valve valve-y.
11) [ Close point for Flow of oxygen from <oxygenIn, out> to <oxygenTank, in> ].
12) open valve valve-z.
13) Connect hose nitrogen-hose to Nitrogen Hose-Port.
14) open valve valve-c.
15) Wait for <oxygenTank, in> to fill with oxygen.
16) closed valve valve-c.
17) closed valve valve-z.
18) [ Close point for Purge of nitrogen from <nitrogenIn, out> to <nitrogenOut in>.
19) open valve valve-y.
20) Open the valve on the connector of nitrogen-hose.
21) Wait 10 minutes for the purge to complete.
22) closed valve valve-y.
23) Close the valve on the connector of nitrogen-hose.
24) Disconnect hose nitrogen-hose from Nitrogen Hose-Port.
25) [ Close point for Flow of hydrogen from <hydrogenIn, out> to <hydrogenOut, in> ].
26) open valve valve-x.
27) open valve valve-a.

planning succeeded.
Appendix B

Running CEP

This appendix looks at the practical issues involved in using CEP. Issues about how to invoke CEP, how to use the CEP debugger and how to use the command line switches. The appendix is written as a compliment to chapter 6, "The CEP programming language".

B.1 Control variables

CEP provides control variables to modify the planners behavior when running a particular plant model. These variables are listed in table B.1. The variables are divided into four groups and these groups will be described in the next sections.

Variables are set in the domain description file using the set construct shown below.

\[ \text{set variable is value.} \]

Control variables can also be examined and manipulated using the debugger as will be described.

B.1.1 Loop control

The first four variables in table B.1 control the way that the planner explores the search space. By default, a depth first search strategy is used. Sometimes when
<table>
<thead>
<tr>
<th>Group</th>
<th>Variable Name</th>
<th>Default Value</th>
<th>Verbosity Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Looping</td>
<td>use.iterative.deepening</td>
<td>false</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>initial_depth</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>depth.increment</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>max.depth</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Priority</td>
<td>priority.mod.each_generation</td>
<td>true</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>last.mod.generation</td>
<td>95</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>priority.var.penalty.first</td>
<td>200</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>priority.var.penalty.remainder</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>priority.unsupported.goal.penalty</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>priority.mod.for.gop.solver</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>Detail</td>
<td>show.plan</td>
<td>false</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>use.new.plan.printer</td>
<td>true</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>number.plan.sequentially</td>
<td>true</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>show.plan.on.success</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>show.derivation.on.success</td>
<td>false</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>show.goals</td>
<td>false</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>verbose.goal.achievement</td>
<td>false</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>show.conflicts</td>
<td>false</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>verbose.conflict.resolution</td>
<td>false</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>verbose.violation.resolution</td>
<td>false</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>verbose.purge.resolution</td>
<td>false</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>verbose.flow.resolution</td>
<td>false</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>verbose.noflow.resolution</td>
<td>false</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>verbose.open_and.closed.detection</td>
<td>false</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>verbose.backtracking</td>
<td>false</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>show.flow.route</td>
<td>false</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>show.mem.stats</td>
<td>false</td>
<td>20</td>
</tr>
<tr>
<td>Modelling</td>
<td>max.purge.branch.depth</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

Table B.1: Control variables
using this strategy, the planner will enter an infinite loop before a valid solution is found to the planning problem. For example, if the planner is told "before a valve can be open it must be closed" and "before a valve can be closed it must be open" then the system may try to form the infinite sequence of actions to continuously open and close a valve.

Ideally a planner should contain an algorithm to detect and prevent looping. However, no correct and complete loop checking algorithms have been proposed in the non-linear planning community. In fact Chapman (1987) shows that the general planning problem is undecidable. This undecidability theorem implies that, for some problems, any correct and complete planner will always loop. This theorem was corroborated a few years later when Feldman and Morris (1990) showed that a popular, seemingly complete loop checking algorithm was in fact incomplete. However, Erol, Nau, and Subrahmanian (1992) has recently demonstrated that a large proportion of planning problems are in fact decidable. These are the problems with finite numbers of grounded conditions. It can be established that all finitely expressible CEP programs fall into this category simply because the finite program size restricts the number of possible object declarations to a finite but unbounded value.

So, although loop checking in CEP is theoretically possible, suitable loop checking algorithms are not yet available. CEP uses the loop checking compromise which is described in Chapman (1987). The planner that will always find the solution to a problem if one exists but which may loop forever if no solution can be found.

In CEP the loop checking compromise is made by using iterative deepening. The planner starts by trying to find a plan containing at most $n$ actions and if this fails, the planner will allow itself $n + m$ actions and then $n + 2m$ and so on. At each state the planner is allowed only a finite number of actions and so only a finite number of plans can be generated. At worst CEP will consider all of these plans before realising that a solution does not exist and increasing the action limit.

The values of $n$ and $m$ can be modified using the CEP control variables $\text{initial\_depth}$ and $\text{depth\_increment}$ respectively. It can be important to get the value of $n$ as close to the correct value as possible because CEP may take a long time to prove that no
solution exists for a given number of actions.

Some plant models don't allow looping and so iterative deepening may be turned off all together by using the variable `use_iterative_deepening`.

### B.1.2 Priority: modifying the order in which goals are solved

CEP maintains an agenda of goals to achieve. At each stage in the planning cycle, a goal is selected from the agenda and then solved by the planner. Goals can be selected in any arbitrary order without affecting the completeness and correctness properties of the CEP planner. However, planning speed can be dramatically improved by using good selection heuristics.

In CEP, each goal is assigned a *priority*, an integer value between -32768 and +32767. When selecting a goal from the agenda, CEP will always choose the goal with the highest (most positive) priority. The assignment of priority is based on a number of heuristics, most of which can be controlled by the user.

#### The `priority` construct

The solution to one goal in a plan will often constrain the options available for solving later goals and so will guide the search. For example, solving a goal will add a protection to the plan and this protection will constrain the time at which conflicting goals can be solved.

One heuristic suggests choosing the most difficult goal to solve first. In this scheme, creating a flow is considered more difficult than opening a valve because a flow will almost always involve operating many valves. The idea is to create a high level plan to constrain the search space before worrying about the lower level detail. This heuristic for planning was first described in Sacerdoti (1974).

CEP provides a construct in its modelling language to allow the priority of a goal to be modified. Using this construct the user is expected to increase the priority of harder goals and reduce the priority of easy goals. The construct has the following format.

```plaintext
priority value { pattern }
```

215
The value field in the priority construct is an integer between -32768 and +32767. This value is added to the base priority of any literal matching the pattern. It is usual for the value to be a multiple of 100 so that the modifier takes precedence over the more fine grained heuristics used in CEP.

The pattern field is simply a condition. The plan symbols in this condition are made from atoms and sets rather than from atoms and plan variables as is more usual. Sets are written in angled brackets. For example, the following statement indicates that it's easy to solve a goal like aperture of valve is closed:

```
priority -100 { aperture of <valve> is <aperture> }
```

The LIFO heuristic

The ability to solve one goal can often help improve confidence in the solution to an earlier goal. For example, consider a plan containing three goals, g, h and i, such that a solution for g will constrain the possible solutions for h but will not constrain the solutions for i. Assume that g has just been solved. The planner should now seek to solve h rather than i for two reasons. If h is found to be unsolvable then this blame may lie in the solution to g and backtracking will immediately change this solution. This would not be the case if i was found to be unsolvable because i is independent of g. Hence, choosing i first would simply delay the planner's discovery that h was unsolvable because of the current solution to g. The second reason for choosing h is that if h was found to be solvable then this would improve confidence in the solution to g because the solution would be known to be compatible with the goal h.

A common heuristic used in planning is to favor subgoals to sibling goals. If an action has two goals a and b and if a new action A is proposed to achieve a then the subgoals of A should be considered before the goal b. The idea is that a and b are more likely to be independent than a and the subgoals of a. The origins of this heuristic are not clear but the heuristic is used in many planners including UCPOP (Barrett et al., 1995).

This heuristic is often implemented by imposing some last in first out (LIFO) strategy on the goal agenda. In CEP this is done by simply adding n points of
priority when the preconditions of the \( n \)th action in the plan are added to the goal agenda.

The control variable `priority_mod_each_generation` can be used to prevent the use of this heuristic. The control variable `last_mod_generation` controls the maximum priority boost that an action will be given as a result of this heuristic. By setting a maximum priority, the heuristic will never overpower better indicators of priority.

**Most constrained choice first**

In solving a goal, the planner often gives up a certain amount of flexibility. For example, if a variable is bound as part of solving an easy goal then the planner can not bind that variable to solve a harder goal later. Nor can the planner constrain the variable to resolve conflict. Often the planner is better to delay solving the particular goal until the variable has been bound.

A goal selection mechanism is a balance between the desire to make decisions to constrain later search and the desire to avoid making decisions so as to preserve flexibility.

The *most constrained choice first* principle suggests choosing the goal that has the fewest options. Statistically, the fewer options for a goal, the more likely that goal will be solved correctly first time.

A very simple implementation of the most constrained choice first heuristic is used in CEP. The planner penalises a goal that contain variables by `priority_var_penalty_first` for the first variable plus, for every additional variable, `priority_var_penalty_remainder`. The planner also penalises goals that could be solved by matching to the effects of an action already in the plan if this match involves binding variables. The number of priority points lost in this case is controlled by `priority_unsupported_goal_penalty`.

**A plan should be safe first**

During the implementation of CEP, the code to resolve a goal of prevention was changed. The old algorithm had the option of adding new actions to the plan to
solve some goal of prevention violations. The current algorithm adds new goals to the agenda rather than adding new actions directly to the plan. As well as solving some earlier problems, the new approach allows the planner to delay the choice on how to resolve a goal of prevention violation.

This new algorithm was found to be slower than the old approach when considering some complex problems. The variable `priority_mod_for_gop_solver` was added to allow extra priority to be given to goals proposed by the goal of prevention handling algorithm. It seems that improving confidence in the solution to a goal of prevention violation by solving related goals is a good thing.

### B.1.3 Detail: how much CEP tells you

The eighteen variables in table B.1, between `show_plan` and `show_mem_stats`, control the amount of detail provided by the planner as it generates a procedure. The use of each variable should be clear from its name.

These variables can be modified individually but more usually, groups of variables are made true at the same time.

The planner maintains a value we will call the verbosity level. The verbosity level defaults to zero but can be modified using the `-v` command line switch or the `v (num)` debugger option.

Most of the variables in the ‘detail’ group are associated with a verbosity threshold (see table B.1). If the planner’s verbosity level is greater or equal to the verbosity threshold of a control variable then the control variable is turned on.

For example, CEP will produce a maximally verbose description of its decision making if ‘-v 999’ is added to the list of command line options. Currently, no control variable has a threshold greater than 21 and so ‘-v 21’ is currently equivalent to ‘-v 999’.

The variable `number_plan_sequentially` is the only variable in the ‘detail’ group does not have a verbosity level. By default, when a plan is printed out the instructions to the operator are numbered sequentially from 1 to whatever (see figure B.1). If `number_plan_sequentially` is made false then the instructions are numbered using...
1) Unstacking blockc from blocka to the table.
2) Moving blockb from the table to blockc.
3) Moving blocka from the table to blockb.

Figure B.1: A plan printout with number_plan_sequentially set to true

[ ] start-1 [4]
[ ] unstack-4 [5]
[ ] stack-5 [3]
[ ] stack-3 [2]
[ ] end-2 []

[4] Unstacking blockc from blocka to the table.

Figure B.2: A plan printout with number_plan_sequentially set to false

the appropriate action numbers. This can help when debugging a model.

B.1.4 Purge model

As discussed in section A.3.3, the model of a purge allows the cleaning of short side branches from the purge path. The control variable ‘max_purge_branch_depth’ determines the number of ports along a side branch that the planner will consider before it assumes that purge no longer takes place.

B.2 The debugger

CEP provides a debugger to help find errors in new models. The debugger is invoked with the -t command line switch. The debugger commands are given in table B.2.

Each command that is given to the debugger is also recorded in a log. This log can be saved to a file using the ‘s’ command. To replay a debugging session the command line option ‘-r file’ can be used where file is a saved log file. Saving and
Search Control:

c CONTINUE from this point
f FAIL the most recent decision
b call the BreakHere function then continue
j JUMP to the next goal failure
q QUIT

State Information:

a # Print ACTION number ‘#’ (eg ‘a1’ = action-1)
p Show the current state of the PLAN
o Show the ORDER of action in the plan
g Show the current GOALS (flaws) of the planner
l # Show the OPERATOR for action number ‘#’
?var # Print the VARIABLE ‘?var’ of action ‘#’

Saving Your Place:

n Add a note to the trace logfile.
s file Save the trace logfile (see the -r command line switch).
0 From now on, read trace input from stdin.

Miscellaneous:

d # Set the external DEBUG level
v # Set the current VERBOS level
h Show this HELP page
r REPORT the state of the control variables
= var # Equate a control variable with a value

Table B.2: Debugger commands
then replaying a log files is a convenient way to jump back to an interesting part of a program trace. This feature has proved invaluable in debugging CEP and the CEP models that have been developed during this project.

The ‘n’ command can be used to add a note to the log. It is good practice to add some form of comment immediately before saving the log to a file.

The ‘b’ option calls the ‘BreakHere ()’ C function inside CEP. When running CEP under gdb or some other source code debugger, ‘BreakHere ()’ should be set up as a breakpoint. If this is done, the ‘b’ will activate the debugger. When tracking down an error in some new enhancement to CEP, it is common to use CEP’s debugger and logfile mechanism to jump to the region of the problem and then use gdb to trace through the program.

The remaining options are self explanatory.

B.3 The CEP command line options

```
plan -f filename [ -l value ] [ -t ] [ -v value ]
[ -r filename ] [ -h ]

-f file          Run the CEP model stored in file.
-l value         Solve the given task value times to test planning speed.

-t               Activate the debugger.
-r file          Rerun the debugging commands listed in file.
-v level         Set the verbosity level to control tracing information.

-h               Print a table of command line options.

-ab              Find all bindings for each plan created.
-ap              Find all sequences of steps to solve a problem.
```
CEP has very few command line options because most features are activated either through the program file or through the debugger. The \( -v \) option is described in section B.1.3. The options \( -t \) and \( -r \) are described in section B.2. The remaining options are assumed to be self explanatory.

The invocation of CEP shown below runs the model 'dom.compressor' with maximally verbose output and with the tracer activated.

\[
\text{plan} -f \text{dom.compressor} -v 999 -t
\]
Appendix C

CEP Example: The Compressor Problem

This chapter describes one of the first OPS tasks that was modelled in CEP. The task is to recreate a procedure that appears in Sutton (1992), a paper on writing operating instructions. The procedure, shown in figure C.1, involves shutting down a simple compressor.

This task is interesting because it demonstrates CEP’s ability to solve high level problems using lower level actions.

C.1 The task

Our objective in this example is to recreate the procedure shown in figure C.1 using CEP. The procedure involves shutting down the compressor in figure C.2.

The title of the procedure is ‘How to shutdown C-206’ and so the goal should be something like ‘shutdown of C-206 is true’. The start state of the procedure represents the actual state of the plant and so will be low level as shown in figure C.3.

The objective ‘shutdown of C-206 is true’ cannot be achieved directly by a single action. Instead, expand operators are provided in order to describe the meaning of shutdown to CEP. The first expand operator translates the goal ‘shutdown’ into three lower level objectives, ‘depressure’, ‘isolate’ and ‘turn off’. The objective ‘isolate’ is
TITLE How to Shut Down C-206
The Off-Gas Recycle Compressor

CHECKLIST

Close XV-2067.
Wait until V-206 pressure (PI-2065) is less than 35 psig (normally 2 minutes).
Stop compressor (Switch SW-2066).
Confirm current meter 2066 is zero.
Close discharge valve XV-2068.
Open XV-2069 until V-206 (PI-2065) pressure is zero.
Close XV-2069.

DETAILED PROCEDURE

200.24.13 Isolate the Suction Line

CAUTION: The Isolation valve, XV-2067, may be partially plugged with corrosion products.
Ensure that the valve is completely closed before stopping the compressor.

(1) Close valve XV-2067.
(2) Wait until the pressure in V-206 (local pressure gauge PI-2065) is less than 35 psig (this
will usually take about 2 minutes). (PI-2065 is shown in Module ABC-200-126).

200.24.14 Stop the Compressor

(1) Stop the compressor by pressing switch SW-2066 on the outside control panel.
(2) Confirm that the current through meter 2066 is zero.
(3) Close the discharge valve, XV-2068 (Shown in Module ABC-200-121).

200.24.15 Vent the Compressor Casing to the Flare

(1) Open bleed valve XV-2069 until the pressure in V-206 is zero (indicated by PI-2065).
(2) Close Bleed Valve XV-2069.

Figure C.1: How to shutdown V-206
suction (XV-2067) outlet (XV-2068)
bleed (XV-2069)
C-206

Figure C.2: The plant section for the Compressor Problem.

START STATE
aperture of 'XV-2067' is open;
aperture of 'XV-2068' is open;
aperture of 'XV-2069' is closed;
pressure of 'C-206' is high;
state of 'C-206' is on;
checkedToBeOff of 'C-206' is false

GOAL STATE
shutdown of C-206 is true;

Figure C.3: The start and goal states.

Figure C.4: The breakdown of the goals in the Compressor Problem.
1) Closed suction valve XV-2067.
2) Wait until the pressure in V-206 (local pressure gauge is PI-2065) is less than 35 psig (this will usually take about 2 minutes).
3) Stop the compressor by pressing switch SW-2066 on the outside control panel.
4) Confirm the current through meter M-2066 is zero.
5) Closed discharge valve XV-2068.
6) Open bleed valve XV-2069.
7) Wait until the pressure in V-206 (local pressure gauge is PI-2065) is zero.
8) Closed bleed valve XV-2069.

Figure C.5: The procedure generated by CEP.

then further broken down. This complete decomposition tree is shown in figure C.4.

There are three goals of prevention for this problem. The suction valve and bleed valve must not be open at the same time in order to prevent back-flow. Similarly, the discharge valve and the bleed valve should not be open together. Finally, the discharge valve must be open if the compressor is on so as to prevent the compressor working against itself.

It is intended that CEP will create the sequences of actions that make up the procedure. However, it is not intended that CEP should create the layout of the procedure in figure C.1. Procedure layout is viewed a user interface problem and beyond the scope of this project.

C.2 The solution

When CEP is run on this problem, the planner finds the solution in about a fifth of a second on a Sun SPARCstation IPX.

The procedure generated by CEP is shown in figure C.5.

C.3 Plan development in CEP

CEP has a debugging mode to allow the user to trace through the development of a plan. This debugging mode can be used to highlight the three different methods CEP uses to develop a plan. The three methods are goal expansion, action addition and
global constraint resolution. The way in which CEP performs these steps highlights significant differences between our approach and earlier systems.

The first planning cycle expands the high level 'shutdown' goal into three lower level goals which can solved more directly.

Unachieved goals at:

shutdown of C-206 is true at end

The precondition 'shutdown of C-206 is true' of the action 'end', was EXPANDED using the operator 'DoShutdown'.

Unachieved goals:

pressure of C-206 is atmospheric at end
isolation of C-206 is true at end
off of C-206 is true at end

The next planning cycle adds a new action to solve the goal 'pressure of C-206 is atmospheric'. The action chosen is from the operator 'Depressure'. The function of this operator is to open the bleed valve and wait for most of the trapped gas to vent away. The action has the effect 'pressure of C-206 is atmospheric' and the preconditions 'pressure of C-206 is medium' and 'bleed of C-206 is open'. These preconditions are added to the goal agenda.

The precondition 'pressure of C-206 is atmospheric' of the action 'end', was solved by PROPOSING 'Depressure'.

The plan is now:

start
Depressure - Wait until the pressure of C-206 is zero.
end

Unachieved goals:

pressure of C-206 is medium at Depressure
bleed of C-206 is open at Depressure
isolation of C-206 is true at end
off of C-206 is true at end
CEP can use action addition to solve any goal so long as an operator has been
defined that will achieve that goal. Previous operator based OPS systems rely on
less general action addition strategies. In Fusillo and Powers (1987), actions can only
be added to solve the end goals of the procedure. In Tomita et al. (1989b), actions
can only added to solve the end goals and their subgoals. What these systems ignore
is that global constraint violations may create new goals for the planner and new
actions may be required to solve these goals and their subgoals.

At the end of this planning cycle, three safety problems are discovered, all caused
by the addition of the new action. In each case, the planner adds a new goal to the
agenda to resolve these violations.

Resolving the goal of prevention violation (attempt 1):
start-1 achieves checkedToBe0ff of C-206 is false
Depressure-3 achieves aperture of XV-2069 is open

Added the goal 'checkedToBeOff of C-206 is true' to
the latest action 'Depressure-3'.

Resolving the goal of prevention violation (attempt 1):
start-1 achieves aperture of XV-2067 is open
Depressure-3 achieves aperture of XV-2069 is open

Added the goal 'aperture of XV-2067 is closed' to
the latest action 'Depressure-3'.

Resolving the goal of prevention violation (attempt 1):
start-1 achieves aperture of XV-2068 is open
Depressure-3 achieves aperture of XV-2069 is open

Added the goal 'aperture of XV-2068 is closed' to
the latest action 'Depressure-3'.

C.4 Implementation

/* The plant model */
property SuctionValve/1, BleedValve/1, DischargeValve/1;
property PressureGuage, Vessel, Switch, Meter;
property aperture/1, checkedToBeOff/1, pressure/1, state/1;
property off/1, shutdown/1, isolated/1;
property name/1, type/1;

frame(pressure).
instance(atmospheric isa pressure).
instance(medium isa pressure).
instance(high isa pressure).

frame(boolean).
instance(true isa boolean).
instance(false isa boolean).

frame(state).
instance(on isa state).
instance(off isa state).

frame(aperture).
instance(open isa aperture, [ name tis Open ]).
instance(closed isa aperture, [ name tis Closed ]).

frame(PressureGuage).
instance('PI-2065' isa PressureGuage).
frame(Switch).
instance('SW-2066' isa Switch).
frame(Meter).
instance('M-2066' isa Meter).

frame(valve).
instance('XV-2067' isa valve, [ type tis suction ]).
instance('XV-2068' isa valve, [ type tis discharge ]).
instance('XV-2069' isa valve, [ type tis bleed ]).

frame(Vessel).
instance('V-206' isa Vessel, [ PressureGuage is 'PI-2065' ]).

frame(machine).
frame(compressor isa machine).
frame(typeXcomp isa compressor).
instance('C-206' isa typeXcomp, [ Vessel is 'V-206',
                                Switch is 'SW-2066',
                                Meter is 'M-2066',
                                SuctionValve is 'XV-2067',
                                BleedValve is 'XV-2069',
                                DischargeValve is 'XV-2068' ]).

/* General Operators. */

operator StopMachine {
machine ?m;

    expand
    * off(?m) is true;
}
using
  state of ?m is off;
  checkedToBeOff of ?m is true;
end }

operator OperateValve {
  valve ?v; aperture ?a;

  achieve
    * aperture of ?v is ?a;
  end

  print (?n)[name of ?a is ?n;] ,
    (?t)[type of ?v is ?t;] ' valve ',
    ?v; }

/* Operators on compressors. */

operator DoShutdown {
  compressor ?c;

  expand
    * shutdown of ?c is true;
  using
    pressure of ?c is atmospheric;
    isolated of ?c is true;
    off of ?c is true;
  end }

/* Operators for typeX compressors. */

operator Enclose {
  typeXcomp ?t;
  valve ?b; valve ?s; valve ?d;

  expand
    * isolated of ?t is true;
  using
    BleedValve of ?t is ?b;
    SuctionValve of ?t is ?s;
    DischargeValve of ?t is ?d;
    aperture of ?b is closed;
    aperture of ?s is closed;
    aperture of ?d is closed;
  end }

operator ReducePressure {
  typeXcomp ?t; valve ?s;

  achieve
    * pressure of ?t is medium;
  using
pressure of ?t is high;
SuctionValve of ?t is ?s;
aperture of ?s is closed;
end

print
'Wait until the pressure in ' (?v) [Vessel of ?t is ?v; ]
' (local pressure gauge is ' (?m) [Vessel of ?t is ?v; PressureGuage of ?v is ?m; ] ')'
is less than 35 psig (this will usually take about 2 minutes); }

operator Depressure {
typeXcomp ?t; valve ?b;

achieve
  * pressure of ?t is atmospheric;
using
  BleedValve of ?t is ?b;

  pressure of ?t is medium;
aperture of ?b is open;
end

print
'Wait until the pressure in ' (?v) [Vessel of ?t is ?v; ]
' (local pressure gauge is ' (?m) [Vessel of ?t is ?v; PressureGuage of ?v is ?m; ] ')'
is zero'; }

/* Operators specific to C-206 */

operator TurnOffXComp {
typeXcomp ?t;

achieve
  * state of ?t is off;
using
  pressure of ?t is medium;
  state of ?t is on;
end

print
'Stop the compressor by pressing switch '
(?s)[ Switch of ?t is ?s; ]
'on the outside control panel'; }

operator CheckXcompIsOff {
typeXcomp ?t;

achieve
  * checkedToBeOff of ?t is true;
using
state of ?t is off;
end

print
   'Confirm the current through meter ' (?m)[ Meter of ?t is ?m; ]
   ' is zero'; }

/* Safety considerations */

restrictions {
  prevent
    checkedToBeOff of C-206 is false;
    aperture of XV-2068 /* discharge */ is closed;
  end

  prevent
    aperture of XV-2067 /* suction */ is open;
    aperture of XV-2069 /* bleed */ is open;
  end

  prevent
    aperture of XV-2068 /* discharge */ is open;
    aperture of XV-2069 /* bleed */ is open;
  end
}

/* Task definition */

goals { require
  shutdown of 'C-206' is true;
end }

start { aperture of 'XV-2067' is open;
  aperture of 'XV-2068' is open;
  aperture of 'XV-2069' is closed;
  pressure of 'C-206' is high;
  state of 'C-206' is on;
  checkedToBeOff of 'C-206' is false;
}

C.5 Conclusion

The example in this section demonstrates three points:

- CEP can achieve high level objects like shutdown using low level actions like turning off a switch and reading a meter.

- CEP is able to solve tasks that have not simply been designed to show the
system off.

- CEP can solve problems which involve both subgoaling and conflict resolution.
Appendix D

CEP Example: Computer-Aided Planning of Purge Operations

This chapter provides a CEP model for an OPS problem described in 'Computer-Aided Planning for Purge Operations' by Fusillo and Powers (1988a). The task involves starting up the section of plant shown in figure D.1. Initially the section is full of air and water. The objective is to move the plant to its normal running state. The difficulty is finding the operations necessary to form a safe procedure.

Fusillo and Powers solved the task described here using an OPS system that has been specifically adapted to reason about purge operations. There system is able to select two of the three purge operations needed in the startup.

CEP is powerful enough to solve this task without resorting to problem specific techniques. Further, CEP is able to generate a complete solution to the problem. This solution produced by CEP consists of the three purge operations as well as nine other unit operations.

CEP's valve sequencing engine is not used in the model developed in this appendix. The plant is simple enough that the valve sequencing engine is not needed to decide how to route each flow of chemical.
Can't contain HCl, chlorine or methane.

Vent-1

Can't contain oxygen or chlorine.

Vent-2

Methane

Chlorine

HCl

Nitrogen

Feed

Feed

Feed

Feed

Mixer

Mixer

Can't contain chlorine, methane or HCl.

Drain-1

Separator

Compressor

Heater

Reactor

Figure D.1: Simplified chlorination flowsheet

<table>
<thead>
<tr>
<th>INITIAL STATE</th>
<th>FINAL STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>temperature = low</td>
<td>temperature = high</td>
</tr>
<tr>
<td>pressure = low</td>
<td>pressure = high</td>
</tr>
<tr>
<td>contains(system, nitrogen)</td>
<td>contains(system, HCl)</td>
</tr>
<tr>
<td>contains(system, oxygen)</td>
<td>contains(system, methane)</td>
</tr>
<tr>
<td>contains(system, water)</td>
<td>contains(system, chlorine)</td>
</tr>
</tbody>
</table>

Figure D.2: Initial and final states for the procedure
not { contains(chlorine) and contains(methane) and temperature 1= high }
not { contains(water) and contains(HCl) }
not { contains(oxygen) and temperature 1= low }
not { contains(methane) and contains(oxygen) }
not { vent1 = open and contains(any_toxic) }
not { drain1 = open and contains(any_toxic) }
not { vent2 = open and contains(oxygen) }
not { vent2 = open and contains(chlorine) }
not { separator_line = open and contains(any_non_condensable) }

?any_non_condensable = { oxygen, nitrogen }
?any_toxic = { chlorine, methane, HCl, chloronated_hydrocarbon }

Figure D.3: Global constraints for the procedure

D.1 The task

The task is the startup of the simplified chlorination plant shown in figure D.1. The section will start of in the state shown in figure D.2 and the goals for the procedure are also shown in this figure.

The problem is difficult because of the safety constraints on the plant (see figure D.3). Fusillo and Powers (1988a) divide these safety considerations into 'local constraints' and 'global constraints'. Local constraints behave like preconditions in CEP and global constraints behave like goals of prevention.

The paper models some safety constraints as local constraints when they would be better represented as global constraints. A good example is the constraint to prevent oxygen entering vent-2 so as to avoid causing an explosion. In the paper models this constraint as 'do not open vent-2 if oxygen is present in the system'. In this interpretation, the constraint fails to fault the procedure 'open vent-2 then open the oxygen inlet'. The constraint would be better modelled as 'oxygen should not be present in the system whenever valve-2 is open'. The earlier model describes the constraint as a precondition while the later model describes the constraint as a goal of prevention.

In the model of the chlorination plant used here, local constraints from Fusillo and Powers (1988a) have been represented as goals of prevention when this has seemed appropriate. If anything, this makes the problem harder to solve. The unmodified
1) Purge oxygen by pressure up blow down with purgative nitrogen to vent vent1.
2) Start compressor comp-1.
3) Start heater htr-1.
4) Close vent1.
5) Purge water by hot-sweep with nitrogen to vent drain1.
6) Close drain1.
7) Purge nitrogen by pressure up blow down with purgative methane to vent vent2.
8) Open separator_line.
9) Open methane-inlet-1.
10) Open HCl-inlet-1.
11) Close vent2.
12) Open chlorine-inlet-1.

Figure D.4: The procedure generated by CEP

problem has been solved using CEP but the model and solution are not shown here. In the original paper, a number of different purge methods were available to the synthesis system. In the model shown here, only two of these methods have been represented ‘pressure up blow down’ for gases and ‘hot sweep’ for liquids.

D.2 The solution

The procedure produced by CEP is shown in figure D.4. The order of some of the actions in this procedure can be changed without affecting the procedure’s correctness. A PERT chart of the actions in the procedure is presented in figure D.5.

CEP requires about a fifth of a second to solve this problem using a SPARCstation IPX.

D.3 Implementation

The chlorination plant is modelled here without using the valve sequencing component of CEP. The model was written before the valve sequencing was added to CEP. Valve sequencing is not required because the plant is simple enough that the choice of flow routes is obvious in each case.

The CEP model used here is coded as follows:
property section/1, supply/1, contents/2, temperature/1;
property aperture/1, state/1, bulk_flow/1, pressure/1;

/* The plant model */
frame(temperature).
frame(bulk_flow).
frame(value isa [temperature, bulk_flow]).
instance(low isa value).
instance(medium isa value).
instance(high isa value).
frame(boolean).
instance(true isa boolean).
instance(false isa boolean).
frame(state).
instance(on isa state).
instance(off isa state).
frame(aperture).
instance(closed isa aperture).
instance(open isa aperture).
frame(section).
instance(section1 isa section).
frame(supply).
frame(gas).
frame(toxic).
frame(volatile).
frame(nonCompress).
frame(chemical isa supply).

instance(chlorine isa [chemical, gas, toxic]).
instance(methane isa [chemical, gas, toxic]).
instance(HCl isa [chemical, gas, toxic]).
instance(CHmClN isa [chemical, toxic]).
instance(oxygen isa [chemical, gas, nonCompress]).
instance(nitrogen isa [chemical, gas, nonCompress]).
instance(water isa [chemical, volatile]).

frame(inlet isa valve).
instance(chlorine-inlet-1 isa inlet,
  [ section is section1,  
  supply is chlorine ]).
instance(methane-inlet-1 isa inlet,
  [ section is section1,  
  supply is methane ]).
instance(HCl-inlet-1 isa inlet,
  [ section is section1,  
  supply is HCl ]).
instance(nitrogen-inlet-1 isa inlet,
  [ section is section1,  
  supply is nitrogen ]).

frame(vent isa valve).
frame(gasVent isa vent).
frame(drain isa vent).
instance(drain1 isa drain,
  [ section is section1 ]).
instance(vent2 isa gasVent,
  [ section is section1 ]).
instance(separator_line isa gasVent,
  [ section is section1 ]).
frame(unscrubbed_vent isa gasVent).
instance(ventl isa gasVent,
  [ section is section1 ]).

frame(heater).
instance(htr-1 isa heater,
  [ section is section1 ]).

frame(compressor).
instance(comp-1 isa compressor,
  [ section is section1 ]).

/* The safety considerations */

restrictions {
  section ?s;  value ?notHigh;
  toxic ?t;  value ?notLow;
  nonCompress ?n;
}
?notHigh != high;
?notLow != low;

prevent
  contents(?s, chlorine) is true;
  contents(?s, methane) is true;
  temperature(?s) is ?notHigh;
end

prevent
  contents(?s, water) is true;
  contents(?s, HCl) is true;
end

prevent
  contents(?s, oxygen) is true;
  temperature(?s) is ?notLow;
end

prevent
  contents(?s, methane) is true;
  contents(?s, oxygen) is true;
end

prevent
  aperture of vent1 is open;
  contents (section1, ?t) is true;
end

prevent
  aperture of drain1 is open;
  contents (section1, ?t) is true;
end

prevent
  aperture of vent2 is open;
  contents (section1, oxygen) is true;
end

prevent
  aperture of vent2 is open;
  contents (section1, chlorine) is true;
end

prevent
  aperture of separator_line is open;
  contents (section1, ?n) is true;
end

/* The operations model */

operator OpenInlet {
section ?s; chemical ?c;
inlet ?v;

achieve
* contents(?s, ?c) is true;
* aperture of ?v is open;
using
  supply of ?v is ?c;
  section of ?v is ?s;
end

print 'Open ' ?v ; }

operator OpenVent {
  vent ?v;
  achieve * aperture of ?v is open; end
  print 'Open ' ?v ; }

operator CloseValve {
  valve ?v;
  achieve * aperture of ?v is closed; end
  print 'Close ' ?v ; }

operator StartHeater {
  heater ?h; section ?s;

  achieve
    * temperature of ?s is high;
    * state of ?h is on;
  using
    section of ?h is ?s;
  end

  print 'Start heater ' ?h ; }

operator StartCompressor {
  section ?s; compressor ?c;

  achieve
    * pressure of ?s is high;
    * state of ?c is on;
  using
    section of ?c is ?s;
  end

  print 'Start compressor ' ?c ; }

operator PUBD {
  gas ?g; gas ?p;
  section ?s; gasVent ?d;
inlet ?v;

  ?g != ?p; /* to avoid loops */

  achieve
* contents(?s, ?g) is false;
  contents(?s, ?p) is true;
  < aperture of ?d is open;
end

print 'Purge ' ?g ' by pressure up blow down with ' 'purgative ' ?p ' to vent ' ?d ; }

operator HotSweep {
  volatile ?1;   gas ?g;
  section ?s;   drain ?d;
  inlet ?v;     inlet ?i;

  achieve
    * contents(?s, ?1) is false;
    contents(?s, ?g) is true;
    < aperture of ?d is open;
end

print 'Purge ' ?1 ' by hot-sweep with ' ?g ' to vent ' ?d ; }

default {
  valve ?v;
  section ?s;
  chemical ?c;

  achieve
    aperture of ?v is closed;
    contents(?s, ?c) is false;
end }

/* The problem description */

start {
  temperature of section1 is low;
  pressure of section1 is low;
  bulk_flow of section1 is low;

  contents(section1, nitrogen) is true;
  contents(section1, oxygen) is true;
  contents(section1, water) is true; }

goals {
  require
    aperture of HCl-inlet-1 is open;
    aperture of methane-inlet-1 is open;
    aperture of chlorine-inlet-1 is open;
    aperture of separator_line is open;
    temperature of section1 is high;
    pressure of section1 is high;
end }
D.4 Conclusion

The example in this section demonstrates three points:

- CEP is general enough to solve problems designed for earlier and very different OPS systems.

- CEP is able to solve problems involving a reasonable number of interacting safety considerations.

- CEP is able to generate procedures that are not fully ordered. The plant designer has the freedom to complete the ordering of the procedure to meet requirements unknown to CEP.
Appendix E

CEP Example: The Sales Gas Filter

This Chapter examines the operation of a sales gas filter described and analysed in Nolan (1993).

This example is interesting because it involves recreating a real operating procedure. At the same time, the example demonstrates the power of the valve sequencing engine in CEP. Three different types of valve are used in this example, ball valves, hose connectors and block and bleed valves. In contrast to earlier systems, CEP is able to procedure different instructions for the operation each different types of valve.

E.1 The task

A simplified model of the sales gas filter is given in section E.1. In the normal running state of the plant, natural gas flows through the filter via two large block and bleed

![Figure E.1: Sales gas filter F-1101A and surrounding pipework](image-url)
valves W1140 and W1141. Periodically, the flow of gas must be stopped so that the filter can be cleaned out. Cleaning the filter involves having a person enter filter and so before the filter can be cleaned, the atmosphere there must be breathable and non-flammable.

In this example, we consider the task of moving the filter from its normal running state to a state where the equipment can be cleaned. The reverse task of restarting the filter after cleaning is not considered here.

E.2 The solution

When CEP is run on this problem, the planner finds the solution in about a fifth of a second on a Sun SPARCstation IPX. The procedure that is generated by CEP is shown in figure E.2.

E.3 Implementation

/* The plant model */

property aperture/1, bleed/1, clean/1, closed/1, contains/2, description/1;
property hatch/1, hose/1, isolated/1, lpVent/1, noFlowUpstream/2;
property noFlowDownstream/2, open/1, pressure/1, pressureStand/2;
property pressureUp/2, purge/7, supply/1, venting/1, vessel/1;

property flow/-1; /* 5 or 7 arguments */

frame(boolean).
instance(true isa boolean).
instance(false isa boolean).

frame(aperture).
instance(open isa aperture, [ description tis Open ]).
instance(closed isa aperture, [ description tis Close ]).

frame(pressure).
instance(high isa pressure).
instance(atmospheric isa pressure).

frame(contents).
frame(chemical).
frame(gas isa chemical).
frame(flammable isa chemical).
1) Close inlet valve W1140 to filter.
2) Turn the bleed screw on inlet valve W1140 to filter.
3) Tape open bleed screw on inlet valve W1140 to filter.
4) Close outlet valve W1141 from filter.
5) Turn the bleed screw on outlet valve W1141 from filter.
6) Tape open bleed screw on outlet valve W1141 from filter.
7) Close valve VV11900.
8) Disconnect the nitrogen-hose from nitrogen_purge_point.
9) Close the door of F-1101A.
10) [Close point for Flow of none from <F-1101A, in> to <lp-stack, in>].
11) Open valve VV11900.
12) Wait until the pressure of F-1101A is atmospheric.
13) Close valve VV11900.
14) [Close point for Flow of nitrogen from <nitrogen-hose, out> to <F-1101A, in>].
15) Connect the nitrogen-hose to the nitrogen_purge_point.
16) Wait until the pressure of F-1101A reaches 60psi.
17) Disconnect the nitrogen-hose from nitrogen_purge_point.
18) Wait 10 minutes to allow the nitrogen to settle in F-1101A.
19) [Close point for Purge of nitrogen from <F-1101A, in> to <lp-stack in>].
20) Open valve VV11900.
21) Wait five minutes for the gas to discharge fully. Repeat the pressure up / blow down until the concentration of "natural-gas" is less than 5 percent.
22) Open the door of F-1101A and place a "NO ENTRY" notice on it.
23) Enter F-1101A and remove one of the draws for cleaning.

Figure E.2: The procedure generate by CEP.
instance(nitrogen isa gas).
instance(air isa gas).
instance(natural-gas isa [gas, flamable]).
instance(none isa chemical).

/* Models */
frame(unit).
frame(inlet isa unit). /* Has a provides slot */
frame(outlet isa unit).
frame(valve isa unit, [ propLinks info [ arc([in, composition], 1, [out, composition]]) ]).
frame(header isa unit, [ propLinks info [ arc([in1, composition], 1, [out1, composition]), arc([in2, composition], 1, [out2, composition]) ]]).
frame(divider isa unit, [ propLinks info [ arc([in, composition], 1, [out1, composition]), arc([in, composition], 1, [out2, composition]) ]]).
frame(vessle isa unit, [ propLinks info [ arc([in, composition], 1, [out, composition]) ]]).
frame(filter isa vessle).
frame(hatch isa valve).

/* Inlets */
instance(gas-inlet isa inlet, [ outports info [out is [F-1101A, in]], supply is natural-gas, pressure is high ]).
frame(hose isa inlet).
instance(nitrogen-hose isa hose, [ outports info [out is [nitrogen_purge_point, in]], supply is nitrogen, pressure is high ]).

/* Valves */
frame(blockBleedValve isa valve). /* Has a description */
frame(hose-join isa valve). /* Has a hose */
frame(ball-valve isa valve). /* "normal" valve */
instance(W1140 isa blockBleedValve, 
  description tis 'inlet valve W1140 to filter',
  outports info [out is [F-1101A, in]]).

instance(W1141 isa blockBleedValve, 
  description tis 'outlet valve W1141 from filter',
  outports info [out is [outlet-a, in]]).

instance(VV11900 isa ball-valve, 
  outports info [out is [lp-stack, in]]).

instance(nitrogen_purge_point isa hose-join, 
  hose is nitrogen-hose,
  description tis 'nitrogen purge point',
  outports info [out is [F-1101A, in]]).

/\* Vessles */
instance(F-1101A isa filter,% Real name unknown.
    [ outports info [out is [W1141, in]],
     outports info [out is [VV11900, in]],
     outports info [out is [F-1101A-hatch, in]],
     hatch is F-1101A-hatch,
     lpVent is lp-stack ]).

instance(F-1101A-hatch isa hatch,
    [ vessel is F-1101A ]).

/\* Outlets */
instance(outlet-a isa outlet).
instance(lp-stack isa outlet).

/\* Operations */
operator IsolateFilter {
  filter ?filter;
  valve ?inlet;
  valve ?outlet;

  expand
    * isolated of ?filter is true;
  using
    noFlowUpstream(?filter, in) is true;
    noFlowDownstream(?filter, in) is true;
end }

/\* Operations on BlockBleedValves */
operator CloseBlockBleedValve {
  blockBleedValve ?valve;
expand
  * aperture of ?valve is closed;
using
  closed of ?valve is true;
  venting of ?valve is true;
  bleed of ?valve is true;
end

operator DoCloseBlockBleedValve {
  blockBleedValve ?valve;

  achieve
    * closed of ?valve is true;
  end

  print 'Close ' (?d) [description of ?valve is ?d;];
}

operator BleedBlockBleedValve {
  blockBleedValve ?valve;

  achieve
    * bleed of ?valve is true;
  using
    closed of ?valve is true;
  end

  print 'Turn the bleed screw on ' (?d) [description of ?valve is ?d;];
}

operator VentBlockBleedValve {
  blockBleedValve ?valve;

  achieve
    * venting of ?valve is true;
  using
    bleed of ?valve is true;
    closed of ?valve is true;
  end

  print 'Tape open bleed screw on ' (?d) [description of ?valve is ?d;];
}

/* Operations on Ball-Valves */

operator OperateBallValve {
  ball-valve ?valve;
  aperture ?aperture;

  achieve
    * aperture of ?valve is ?aperture;
  end

  print (?d)[ description of ?aperture is ?d; ] ' valve ' ?valve ;
}
/* Operations on Hoses */

operator OpenHoseJoin {
hose-join ?join;
hose ?hose;

achieve
  * aperture of ?join is open;
using
  hose of ?join is ?hose;
end

print 'Connect the ' ?hose ' to the ' ?join ; }

operator CloseHoseJoin {
hose-join ?join;
hose ?hose;

achieve
  * aperture of ?join is closed;
using
  hose of ?join is ?hose;
end

print 'Disconnect the ' ?hose ' from ' ?join ; }

/* Operating the hatch */

operator OpenHatch {
hatch ?hatch;

achieve
  * aperture of ?hatch is open;
end

print 'Open the door of ' (?vessle)[ vessle of ?hatch is ?vessle; ] ' and place a "NO ENTRY" notice on it'; }

operator CloseHatch {
hatch ?hatch;

achieve
  * aperture of ?hatch is closed;
end

print 'Close the door of ' (?vessle)[ vessle of ?hatch is ?vessle; ] ; }

/* How to pressure-up blow-down */

operator CleanFilter {
filter ?filter;
flamable ?flamable;
chemical ?chem;
hatch ?hatch;

?chem != ?flamable;

achieve
  * clean of ?filter is true;
using
  hatch of ?filter is ?hatch;
  aperture of ?hatch is open;
end

print 'Enter ' ?filter ' and remove one of the draws for cleaning'; }

operator BlowDown {
  filter ?filter;
  outlet ?vent;
  chemical ?purgant;
  chemical ?waste;

  ?purgant != ?waste;

  achieve
    * contains(?filter, ?waste) is false;
    pressureStand(?filter, ?purgant) is false;
    pressure of ?filter is atmospheric;
  using
    lpVent of ?filter is ?vent;
    pressureStand(?filter, ?purgant) is true; /* Cheat :) */
    purge(?filter, in, ?filter, in, ?vent, in, ?purgant) is true;
end

print 'Wait five minutes for the gas to discharge fully. Repeat' ' the pressure up / blow down until the concentration of "' ?waste " is less than 5 percent'; }

operator Stand {
  filter ?filter;
  chemical ?chem;

  achieve
    * pressureStand(?filter, ?chem) is true;
    pressureUp(?filter, ?chem) is false;
  using
    pressureUp(?filter, ?chem) is true;
    noFlowUpstream(?filter, in) is true;
    noFlowDownstream(?filter, in) is true;
end

print 'Wait 10 minutes to allow the ' ?chem ' to settle in ' ?filter; }

operator PressureUp {
filter ?filter;
inlet ?inlet;
chemical ?chem;

achieve
  * pressureUp(?filter, ?chem) is true;
  contains(?filter, ?chem) is true;
  pressure of ?filter is high;
using
  pressure of ?filter is atmospheric;
  pressure of ?inlet is high;
  supply of ?inlet is ?chem;
  flow (?inlet, out, ?filter, in, ?chem, false, true) is true;
  noFlowDownstream (?filter, in) is true;
end

print 'Wait until the pressure of ' ?filter ' reaches 60psi'; }

operator Depressure {
  filter ?filter;
  outlet ?vent;

  achieve
    * pressure of ?filter is atmospheric;
using
   lpVent of ?filter is ?vent;
    flow (?filter, in, ?vent, in, none, true, false) is true;
    pressure of ?filter is high;
end

print 'Wait until the pressure of ' ?filter ' is atmospheric'; }

priority 1000 {
  /* Conditions with the following format are easy to achieve because 
     they are constants */
supply of <inlet> is <chemical>;
pressure of <inlet> is <pressure>;
hatch of <vessle> is <hatch>;
vessle of <hatch> is <vessle>;
lpVent of <filter> is <valve>;
aperture of nitrogen-hose is closed; }

  /* Safety considerations */

restrictions {
vessle ?vessle;
hatch ?hatch;

prevent
  aperture of ?hatch is open;
  vessle of ?hatch is ?vessle;
  contains (?vessle, natural-gas) is true;
end }
/* The problem */

default {
    valve ?v;

    achieve
        aperture of ?v is closed;
    
    end }

start { aperture of W1140 is open;
    aperture of W1141 is open;
    contains (F-1101A, natural-gas) is true;
    pressure of F-1101A is high;
    clean of F-1101A is false; }

goals { require
    clean of F-1101A is true;
end }

E.4 Conclusion

The example in this section demonstrates two points about CEP:

- CEP is able to solve complex problems because valve sequencing and planning are only loosely coupled. In this example, the valve sequencing engine generates requests to open valves and the planning tool is able to interpret these requests so that different instructions and different numbers of steps are used to open different types of valves.

- CEP is able to recreate apparently real procedures.